# Automatic Parameterisation of Semantic 3D City Models for Urban Design Optimisation

Kian Wee Chen [1], Patrick Janssen [2], Leslie Norford [3]

[1] CENSAM, Singapore-MIT Alliance for Research and Technology, Singapore
[2] National University of Singapore, Singapore
[2] Department of Architecture, Massachusetts Institute of Technology, USA

chenkianwee@gmail.com, patrick@janssen.name, lnorford@mit.edu

**Abstract.** We present an auto-parameterisation tool, implemented in Python, that takes in a semantic model, in CityGML format, and outputs a parametric model. The parametric model is then used for design optimisation of solar availability and urban ventilation potential. We demonstrate the tool by parameterising a CityGML model regarding building height, orientation and position and then integrate the parametric model into an optimisation process. For example, the tool parameterises the orientation of a design by assigning each building an orientation parameter. The parameter takes in a normalised value from an optimisation algorithm, maps the normalised value to a rotation value and rotates the buildings. The solar and ventilation performances of the rotated design is then evaluated. Based on the evaluation results, the optimisation algorithm then searches through the parameter values to achieve the optimal performances. The demonstrations show that the tool eliminates the need to set up a parametric model manually, thus making optimisation more accessible to designers.

**Keywords:** City Information Modelling, Conceptual Urban Design, Parametric Modelling, Performance-Based Urban Design

## 1 Introduction

In the early stages of urban design, designers will explore the impact that building layouts and building forms have on various urban performances. For example, building layouts and forms strongly influence solar availability [1] and urban ventilation [2, 3]. One powerful way of facilitating this exploration is using optimisation algorithms. In the optimisation process, designers create parametric models that allow design variants to be easily generated and evaluated according to performance objectives. An optimisation algorithm is used to search for a series of optimal design instances automatically. The advantage is that huge numbers of design variants can be explored and the design can be optimised.

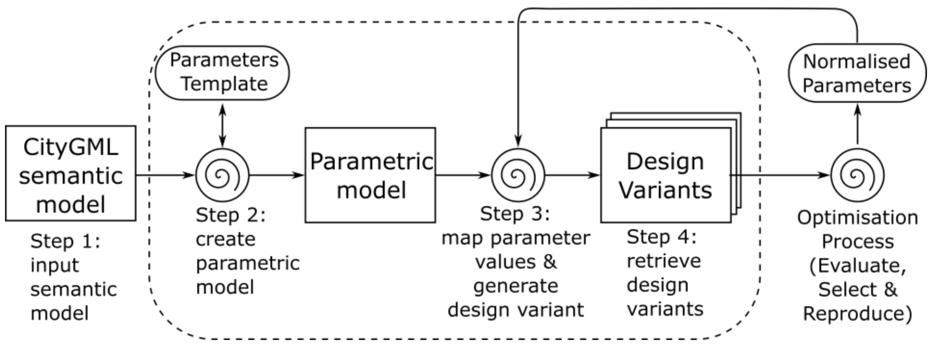
At the building scale, Attia et al. have highlighted how parametric modelling is one of the main technical hurdles for applying such optimisation algorithms [4]. The authors foresee the same technical difficulty for urban design. Currently, designers who do urban design optimisation use parametric modelling applications with a Visual Programming Language (VPL) interface to encode their design as a parametric model [5, 6]. Although it has been shown that design students are able to learn parametric modelling faster through VPL as compared to textual programming, VPL quickly becomes inadequate when applied to complex design task such as generative designs and large scale designs [7, 8]. In addition, VPL network easily becomes unmanageable with too many links, nodes and confusing iteration [9]. Thus, VPL is less than ideal for parameterising an urban scale design. Instead of using VPL, we propose an automated method to facilitate the parameterising of urban models from 3D semantic models.

In this paper, we considered a scenario where the designer is planning a cluster of buildings on a large urban plot. The shape of each building plan is assumed to be fixed, but the position, orientation, and height of the buildings may be varied subject to constraints. One key constraint is the Floor Area Ratio (FAR) of the plot, which results in a maximum total Gross Floor Area (GFA) for all the buildings on the plot. The heights of the building need to be constrained to ensure that this GFA is not exceeded. Other constraints are that the buildings must stay within the plot boundary and must not intersect one another.

To demonstrate this method, we implemented a prototype auto-parameterisation tool. The tool imports a semantic urban model in the CityGML format [10] and automatically generates a parametric model by parameterising the position, orientation, and height of building objects. These parametric models can then be used to optimise the urban layout and forms of these building objects. The usefulness of the tool is that it allows designers to bypass the complex and time-consuming step of constructing parametric models, thereby removing a key hurdle in the application of optimisation algorithms within the urban design process.

## 2 Method

The proposed method consists of four steps as shown in **Fig. 1**: 1) inputting the semantic model, 2) creating the parametric model, 3) mapping parameter values and generating designs 4) retrieve design variant models.



Parameters Template
CityGML semantic model
Step 1: input semantic model
Step 2: create parametric model
Parametric model
Step 3: map parameter values & generate design variant
Design Variants
Step 4: retrieve design variants
Normalised Parameters
Optimisation Process (Evaluate, Select & Reproduce)

**Fig. 1.** Proposed method for urban optimisation. The dotted box indicates the steps performed by the auto-parameterisation tool.

In step 1, the method requires a CityGML model as input. In step 2, the tool first analyses the semantic CityGML model and identifies all the building objects that are on the plot. It then creates a parametric model by applying a parameter template to each of these objects. The prototype parameter template defines three parameters: building position, building orientation, and building height. Additional templates can be created and customised by designers for different urban and building typologies. For example, for a carpark, a simpler parameter template may be created that does not have a height parameter, while for a podium block with a tower above, a more complex parameter template may be created that has separate height parameters for the podium and the tower.

The mapping and generate design variants process in step 3 and optimisation process are linked in a cyclical loop. In each iteration, the mapping process generates a population of design variants and the optimisation process evaluates them, and then perform reproduction and selection. The reproduction and selection will generate normalised parameter values for a new and fitter population. The mapping process takes in the normalised values and maps them to model specific values for generation of design variants. For the three parameters; building position, height and orientation, the procedure for mapping the normalised values to the model-specific values are as follows:

1. For the building position parameter, a grid is overlaid on the plot, and each position in the grid is assigned a numeric index. The normalised position value is then mapped to an index value in the range {0, N-1}, where N is the number of grid points. The default number of grid points is 100, but the designer can specify the grid density.
2. For the building orientation parameter, *O*, the normalised orientation value is mapped to a rotation angle in the range {0.0, 360.0}.
3. For the building height parameter, the calculation of the actual height values needs to take into account the FAR constraint. First, the plot area and base floor area for each building are calculated. The number of floors for each building is then calculated as follows:

$$F = (h \cdot a_p \cdot r_p)/(a_b \cdot \sum_{i=0}^{n-1}(h_i))$$

where $F$ is the number of floors, $h$ is the normalised height, $a_p$ is the area of the plot, $r_p$ is the FAR for the plot, and $a_b$ is the area of the building base. The height of the building is then $F$ multiplied by the floor to floor height for the building, for which either a default value of 3m can be used, or a value can be specified by the designers.

The parameters for position and orientation may result in design variants where buildings either intersect one another or intersect the plot boundary. The buildings are placed in sequence to make sure these intersections do not happen, and when constraints are broken, the parameter values are iteratively adjusted until a valid model is generated.

## 3 Implementation

The proposed method is implemented in a Python library called Pyliburo [11] (*https://github.com/chenkianwee/pyliburo*) as two Python classes, `Parameterise` and `BaseParm`. The two classes use the modelling kernel and the CityGML reader/writer from Pyliburo for their geometrical operations and for reading and writing CityGML files. The `Parameterise` class represents a parametric model. To parameterise a CityGML model, one needs first to configure a series of `BaseParm` classes and append them to the `Parameterise` class. Each `BaseParm` class specifies the parameterisation procedure for a parameter. We implemented the `BaseParm` class as a Python abstract class to facilitate reuse and extensibility. The `Parameterise` class then reads and parameterises the CityGML model according to the appended `BaseParm` classes. A combination of multiple `BaseParm` classes forms the parameter template as mentioned in section 2. Thus, the `Parametrise` class has a one-to-many (1-N) relationship with the `BaseParm` classes as shown in **Fig. 2**, a Unified Modelling Language (UML) class diagram. **Fig. 2** also illustrates an implementation of the `BaseParm` class, `BldgOrientationParm`, which will be discussed in detail below.

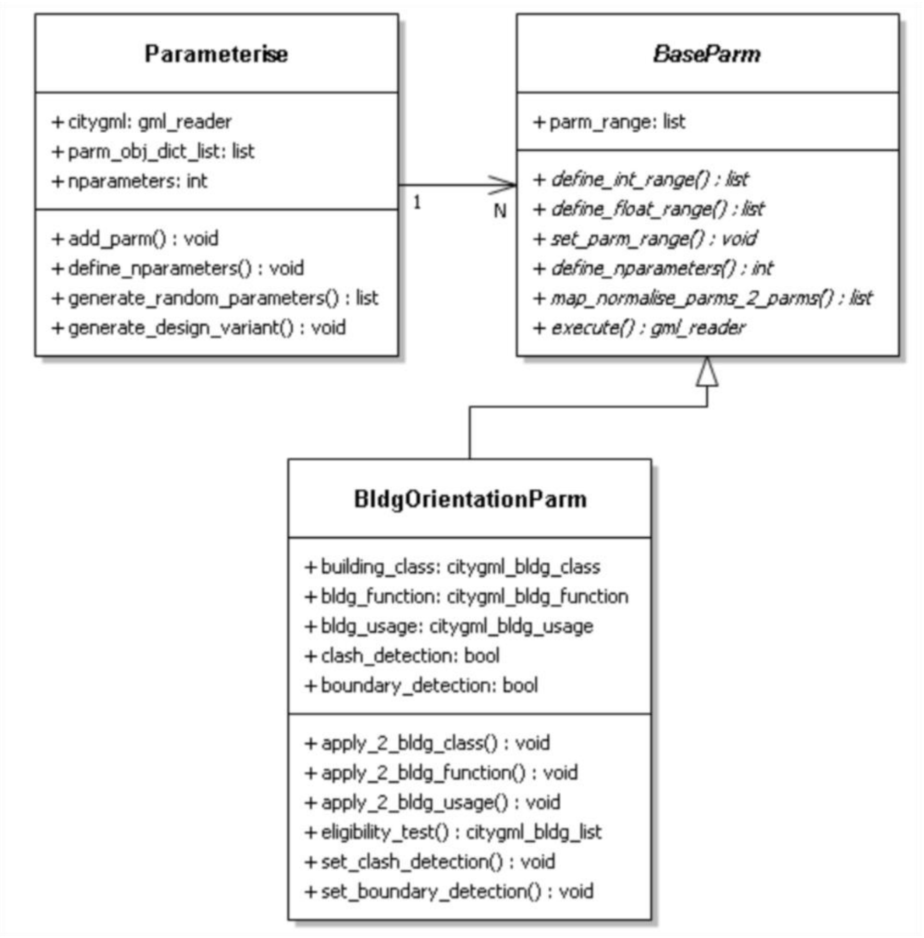

Parameterise
+ citygml: gml_reader
+ parm_obj_dict_list: list
+ nparameters: int
+ add_parm() : void
+ define_nparameters() : void
+ generate_random_parameters() : list
+ generate_design_variant() : void
1
N
BaseParm
+ parm_range: list
+ define_int_range() : list
+ define_float_range() : list
+ set_parm_range() : void
+ define_nparameters() : int
+ map_normalise_parms_2_parms() : list
+ execute() : gml_reader
BldgOrientationParm
+ building_class: citygml_bldg_class
+ bldg_function: citygml_bldg_function
+ bldg_usage: citygml_bldg_usage
+ clash_detection: bool
+ boundary_detection: bool
+ apply_2_bldg_class() : void
+ apply_2_bldg_function() : void
+ apply_2_bldg_usage() : void
+ eligibility_test() : citygml_bldg_list
+ set_clash_detection() : void
+ set_boundary_detection() : void

**Fig. 2.** UML class diagram of the two classes for the auto-parameterisation tool

## 3.1 Parameterise Class

The `Parameterise` class parameterises a CityGML model according to the added `BaseParm` classes. The `Parameterise` class stores the CityGML data as a pycitygml `Reader` class from Pyliburo. It has two other attributes. First, `parm_obj_dict_list` is a list of dictionaries documenting the appended `BaseParm` classes and the corresponding number of parameters for this parameter. `BaseParm` classes are added to the `parm_obj_dict_list` using the `add_parm` method. Second, the `nparameters` attribute is an integer indicating the number of parameters of the parametric model calculated by the `define_nparameters` method. The `Parameterise` class generates a design variant using the `generate_design_variant` method; a random design variant can be generated by using the random parameters generated by the `generate_random_parameters` method as inputs.

## 3.2 BaseParm Class

The `BaseParm` abstract class defines the parameterisation procedure for any parameter. Any implementation of the `BaseParm` abstract class has the attribute `parm_range`, which is a list of all the possible parameter values. The `parm_range` attribute can be defined by any of three methods: the `define_int_range` method, by specifying the starting integer, the last integer and the step between each integer; the `define_float_range` method, by specifying the starting float, the last float and the step between each float; and the `set_parm_range` method, by specifying all the possible parameter values. With the `parm_range` set, the `map_normalise_parms_2_parms` method maps the normalised parameter values received to the defined parameter range. The `define_nparameters` method calculates and returns the number of parameters generated by this `BaseParm`. Eventually, with everything configured, the `execute` method executes the parameterisation procedure for this parameter.

We have implemented three `BaseParm` classes: `BldgFlrAreaHeightParm`, `BldgOrientationParm` and `BldgPositionParm`. We will describe the `BldgOrientationParm` implementation to illustrate the abstract class. The `BldgOrientationParm` class has all the methods specified by the `BaseParm` abstract class. In addition, we have implemented five additional attributes and six additional methods as shown in **Fig. 2**.

By configuring the attributes and methods, users of the library are defining the constraints of the `BldgOrientationParm` class. The `execute` method requires two inputs, the CityGML `Reader` object from Pyliburo that carries all the information from the CityGML model and the list of normalised parameters. First, the normalised parameters are mapped to the model-specific parameters using the `map_normalise_parms_2_parms` method. The `eligibility_test` method then filters the buildings as specified by the `bldg_class`, `bldg_function` and `bldg_usage` attributes to find the eligible buildings. Last, the `execute` method loops through all the eligible buildings and rotates them counter-clockwise from their original orientation. If clash detection or boundary detection is set to True, and the rotated building clashes with other buildings or is outside the land-use boundary, the building will not be rotated. The method then documents and returns the rotated design in the CityGML `Reader` object. The CityGML `Reader` object is then passed on to the next `BaseParm` classes. After being parameterised by all the `BaseParm` classes in the `Parameterise` class, a design variant is successfully generated.

## 4 Examples

We demonstrate the proposed method and tool on two examples. One simple example shows the operation of the auto-parameterisation tool and the second example shows the integration of the tool into an optimisation design process. For the demonstrations, we develop the tool by writing two Python scripts with the auto-parameterisation Python classes. Both scripts require only one main input: the CityGML file to be parameterised and optimised. The source code is available on GitHub (*https://github.com/chenkianwee/pyliburo_example_files*).

### 4.1 Example 1

The first example is a simple case of a land-use with 13 residential buildings and two multi-storey carpark buildings (**Fig. 3**a). The example was modelled in the CityGML format and imported into the auto-parameterisation tool.

The automatic parameterisation process is illustrated as follows:

1. Each residential building is assigned the parameters building position, orientation and height, while carparks are only assigned position and orientation.
2. A parametric model with 43 parameters is automatically created.
3. The three parameters are as described in section 2. The ranges of their parameters are as follows:
    a. The density of the grid for the position parameter is 10m by 10m.
    b. The rotation angle for the orientation parameter is $0 <= O <= 350$, $10 \mid O$.
    c. The normalised height for the height parameter is $3 <= h <= 10$, $1 \mid h$
4. **Fig. 3**b shows four design variants generated from the automatic parameterisation process.

Fig. 4 shows a snippet of the script. The script requires one input; the CityGML file path to be parameterised. The `BldgFlrAreaHeightParm` class is first initialised. The `define_int_range` method then defines lower bound, upper bound and the step between each value of the normalised height. As specified by `apply_2_bldg_function("1000")`, only residential buildings have the height parameter. The parameter “1000” is the residential building function code in the CityGML schema. The `BldgFlrAreaHeightParm` is then added to the `Parameterise` class. Random normalised parameters are generated by the class accordingly and used to generate a design variant.

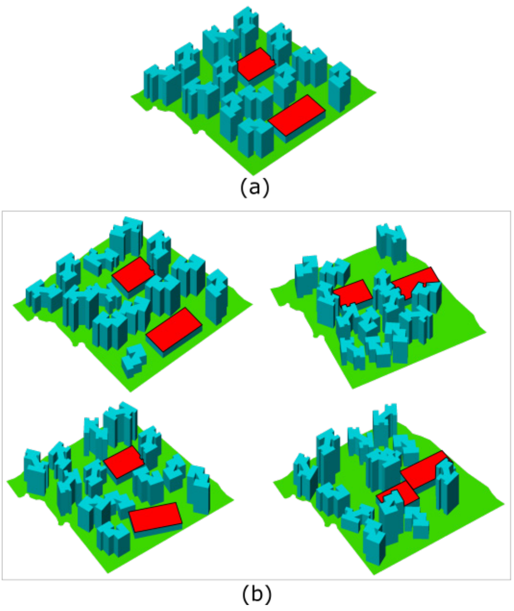


**Fig. 3.** (a) Example 1 has a land-use with 13 residential buildings (blue) and two multi-storey carpark buildings (red) (b) four design variants randomly generated by the auto-parameterisation tool.

```
input = CityGML_filepath
height_parm = BldgFlrAreaHeightParm ()
height_parm.define_int_range(3,10,1)
height_parm.apply_2_bldg_function("1000")
parameterise = Parameterise(input)
parameterise.add_parm(height_parm)
parameters = parameterise.generate_random_parameters()
parameterise.generate_design_variant(parameters)
```

**Fig. 4.** Snippet of the auto-parameterisation script with the input CityGML file highlighted in bold

## 4.2 Example 2

The second example is a land-use with 25 residential buildings in the tropical climate arranged in a five-by-five grid as shown in **Fig. 5**a. The example is parameterised in height, orientation and position as described in section 2; the same parameter ranges are used as in example 1 for the auto-parameterisation process. Through the parametric model, we want to explore the impact of different configurations of height, orientation and position on the solar irradiation and urban ventilation performances. **Fig. 5**b shows two randomly generated design variants.

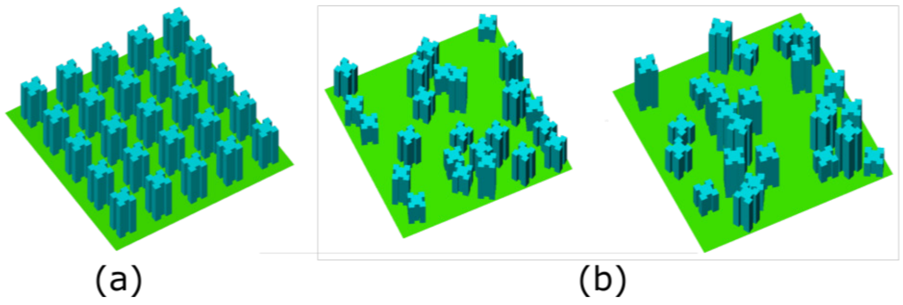


**Fig. 5 (a)** Example 2 with 25 residential buildings arranged in five by five grid (b) two design variants randomly generated by the auto-parameterisation tool.

We measure the solar irradiation performance using the Non-Solar Heated Façade Area Index (NSHFAI), which is the fraction of the façade area receiving annual solar irradiation equal to or below a threshold value. The threshold value is 227.45 kWh/m$^2$ and is based on the Singapore Green Mark's maximal permissible Envelope Thermal Transfer Value (ETTV) of 50W/m$^2$ [12], multiplied by the 4549 annual daylight hours in Singapore [13]. NSHFAI is to be maximised in the optimisation process. We measured the urban ventilation performance using the Frontal Area Index (FAI) [14, 15], which is the ratio of the total façade area projected to a vertical plane facing the user-defined wind direction to the horizontal plane area. The FAI is calculated using the northeast prevailing wind direction of Singapore, and the site is gridded 100mx100m. The average FAI is calculated and used as a performance objective. The average FAI is to be minimised in the optimisation process. Py2radiance [16] is used to execute Radiance [17] for simulating the annual solar irradiation, and Pyliburo is used to calculate the FAI.

The parametric model is used for running an optimisation process with NSHFAI and FAI as the performance objectives. We used The Non-dominated Sorting Genetic Algorithm II (NSGAII) [18] as implemented in the Pyliburo library for the optimisation process. The default values of 0.8 and 0.01 are used for the crossover rate and mutation rate. **Fig. 6** shows a snippet of the script illustrating the integration of the auto-parameterisation library with the optimisation process.

The script initialises the NSGAII optimisation class by defining genes (parameters), and scores (performance objectives) as a list of Python dictionaries. Then for each individual in each generation, a design variant is generated using the genotype (parameters) of an individual through the `Parameterise` class, `generate_design_variant` method. The generated design variant is evaluated in terms of NSHFAI and FAI. The two performances are then used for the feedback procedure, crossover and mutation, in NSGAII. The feedback procedure generates a new generation of individuals, and the cycle repeats until as specified by the user. The main input for the script is the CityGML model of the design, while default values are used for the other inputs for the optimisation algorithm and solar simulations.

```
#define the genes
gene_dict_list = []
for _ in range(number_of_genes):
    gene_dict={"type": "float_range", "range":(0.0,1.0)}
    gene_dict_list.append(gene_dict)
#define the scores
shgfai_dict = {"name": "nshfai", "minmax": "min"}
dfai_dict = {"name": "fai", "minmax": "max"}
score_dict_list = [nshfai_dict, fai_dict]

population=initialise_nsga2(gene_dict_list, score_dict_list)

for generation in range(ngeneration):
    individuals = population.individuals
    for individual in individuals:
          parameters = individual.genotype.values
          design_variant = parameterise.generate_design_variant
          (parameters)
          nshfai = eval_solar(design_variant)
          fai = eval_fai(design_variant)
          individual.set_score(0, nshfai)
          individual.set_score(1, fai)
    feedback_nsga2(population)
```

**Fig. 6.** A snippet of the optimisation python script

**Results.**

We ran the optimisation for 40 generations with an initial population of 25 design variants. There are seven design variants on the Pareto-front (red dots) in reference to the base case performance (blue dot) as shown in **Fig. 7**. All the design variants' performances improve significantly in comparison with the base case. Three design variants are visualised in 3D (**Fig. 7**). Two design variants are on the two ends of the Pareto front: one with the lowest NSHFAI and FAI (design variant 853), the other with the highest NSHFAI and FAI (design variant 993). The third design variant is a negative example with low NSHFAI and high FAI (design variant 153).

Design variant 853 has similar NSHFAI performance as design variant 153 with only a 0.06 difference in NSHFAI. In comparison, design variant 993 performs significantly better in terms of NSHFAI by a 0.15-1.21 difference. This is due to the inter-shading between buildings in design variant 993 as shown in **Fig. 8**. However, the close-packing of the buildings increases its average FAI as compared to design variant 853 by 0.04 as shown in **Fig. 9**. The congregation of buildings in the centre of the plot contributes to the increase in FAI. The buildings in design variant 853 are better spaced apart and as a result has lesser inter-shading but better FAI. Although design variant 153 is also well-spaced like design variant 853, its bad building positions and orientations, with their vertical facade facing the wind direction, are blocking the wind flow.

By visualising both the Pareto front design variants and bad performing design variants performances in 3D, designers are able to quickly assess how the building arrangement are affecting the NSHFAI and FAI.

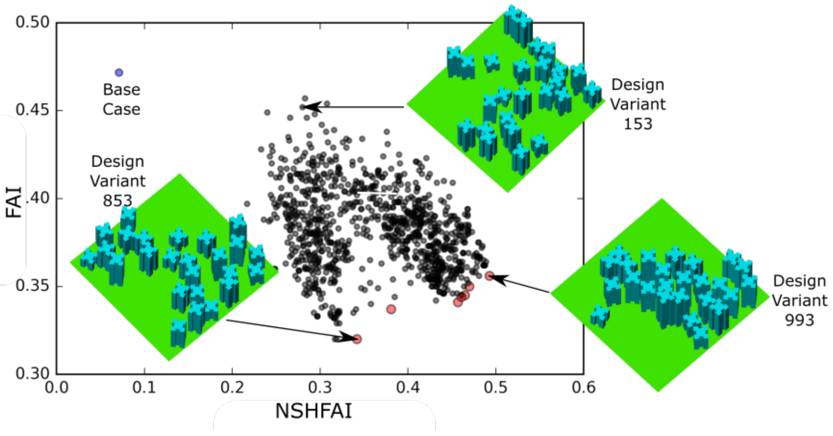


**Fig. 7.** The optimisation generated 1000 design variants with seven on the Pareto front (red dots) and the base case (blue dot)

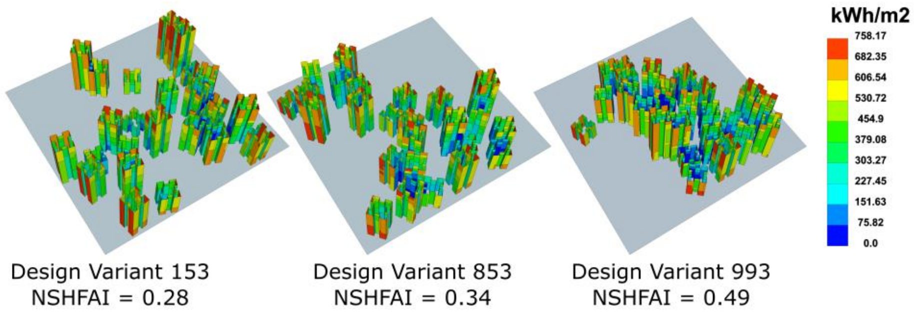


**Fig. 8.** The NSHFAI result of the three design variants. The blue surfaces are surfaces receiving irradiation less than the threshold value.

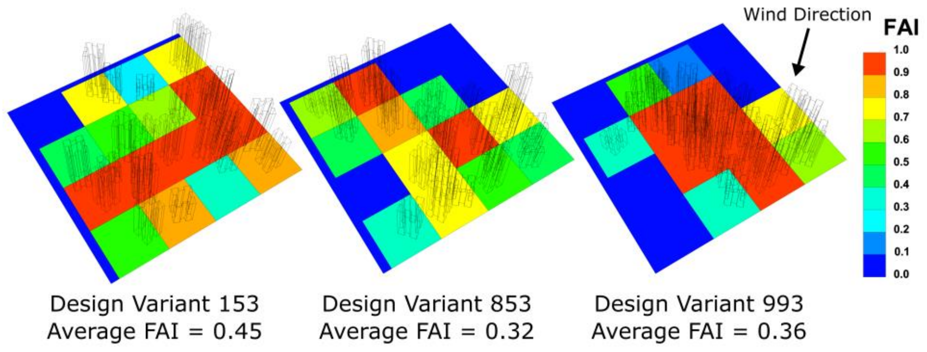


**Fig. 9.** The FAI false colour diagram of the three design variants.

## 4.3 Discussion

The two example scripts are able to auto-parameterise a CityGML model in terms of height, orientation and position without additional inputs from the designers. Example 2 demonstrates the capability of the auto-parameterisation tool to engage optimisation in the design process without the need to parameterise the design manually.

The Python library that is used to develop the auto-parameterisation tool offers flexibility for designers with a programming background. This is very useful for customising the parameterisation process for their own project. As shown in example 1, it is straightforward to customise the auto-parameterisation by appending different `BaseParm` classes to the `Parameterisation` class. For example, to parameterise the orientation and height of the buildings, one simply appends the `BldgFlrAreaHeightParm` and `BldgOrientationParm` to the `Parameterise` class and leaves out the `BldgPositionParm`.

While customising the parameter template, designers need to pay attention to the sequence of the appended `BaseParm` classes, as the sequence affects the parameterisation. If the clash detection is active for either `BldgOrientationParm` or `BldgPositionParm`, the same set of parameters will result in a different result depending on the sequence. For the sequence rotate-relocate, a building that is not allowed to rotate because it clashes with a neighbouring building will be relocated to a new position without being rotated. However, for the sequence relocate-rotate, the building will be relocated first and allow to be rotated at the new position as it will not clash with any neighbouring buildings.

We have only demonstrated a very narrow set of parameterisation classes. The auto-parameterisation tool is highly extensible; one can extend the `BaseParm` class to include an implementation to parameterise the building forms of a design. In contrast to the current classes where parameters are assigned to each building, we will parameterise the building forms of a design by assigning a parameter to each land-use. We will define the `execute` method depending on the form that we will explore. In the method, the parameter would influence the building forms on the land-use. While defining the `execute` method, it is important to note whether the implementation is compatible with the other `BaseParm` classes. For example, if the urban form parameter value changes the number of buildings on the land-use, the urban form `BaseParm` implementation must be appended after the `BldgFlrAreaHeightParm`, `BldgOrientationParm` and `BldgPositionParm`. This is because the three classes assign parameters according to the original number of buildings in the CityGML model; if the number of buildings is changed, there will be a mismatch of parameters to the buildings to be either rotated, relocated, shorten or heighten.

Eventually, it is up to the designers to customise their script. The Python library is open and available for any interested designer to develop, extend, explore and experiment.

## 5 Conclusion

The research demonstrates the feasibility of auto-parameterising semantic 3D city models. The proposed method is a viable solution in helping urban designers computationally encode their designs as a parametric model for optimisation. The auto-parameterisation tool has been integrated into an optimisation process by connecting it with an optimisation algorithm NSGA2. The integration simplifies the encoding process of the urban optimisation process and makes the process more accessible to designers.

Ongoing research to improve the auto-parameterisation tool includes developing a Graphical User Interface (GUI) for designers with no programming background to customise the parameterisation process and integrating the Python library with a parallel computing framework to speed up the optimisation process. Firstly, a parameter tree GUI like a feature-based modelling application such as CATIA will be useful for designers to append and arrange the sequence of the `BaseParm` classes in the `Parameterise` class. Secondly, the optimisation process currently takes up to 155 hours to generate 1000 design variants. The process can be significantly sped up by integrating the optimisation process into a parallel computing framework. We envision an accessible and fast optimisation process will encourage adoption of optimisation algorithm in the urban design process.

**Acknowledgements.** This research was supported by the National Research Foundation Singapore through the Singapore MIT Alliance for Research and Technology's Center for Environmental Sensing and Modeling interdisciplinary research program.

## References

1. R. Compagnon, Solar and daylight availability in the urban fabric, Energy and Buildings, 36, 321–328 (2004)

2. E. Ng, C. Yuan, L. Chen, C. Ren, and J. C. H. Fung, Improving the wind environment in high-density cities by understanding urban morphology and surface roughness: A study in Hong Kong, Landscape and Urban Planning, 101, 59–74 (2011)
3. C. Yuan, L. Norford, R. Britter, and E. Ng, A modelling-mapping approach for fine-scale assessment of pedestrian-level wind in high-density cities, Building and Environment, 97, 152–165 (2016)
4. S. Attia, M. Hamdy, W. O'Brien, and S. Carlucci, Assessing gaps and needs for integrating building performance optimization tools in net zero energy buildings design, Energy and Buildings, 60, 110–124 (2013)
5. M. Bruno, K. Henderson, and H. M. Kim, Multi-objective Optimization in Urban Design, in Proceedings of the 2011 Symposium on Simulation for Architecture and Urban Design, San Diego, CA, USA (2011) at <http://dl.acm.org/citation.cfm?id=2048536.2048549>
6. H. Taleb and M. A. Musleh, Applying urban parametric design optimisation processes to a hot climate: Case study of the UAE, Sustainable Cities and Society, 14, 236–253 (2015)
7. A. Leitão, L. Santos, and J. Lopes, Programming Languages For Generative Design: A Comparative Study, International Journal of Architectural Computing, 10, 139–162 (2012)
8. G. Celani and C. Vaz, CAD Scripting And Visual Programming Languages For Implementing Computational Design Concepts: A Comparison From A Pedagogical Point Of View, International Journal of Architectural Computing, 10, 121–138 (2012)
9. P. Janssen, Visual Dataflow Modelling: Some thoughts on complexity, in Proceedings of the 32nd eCAADe Conference, Newcastle, UK (2014)
10. G. Gröger and L. Plümer, CityGML – Interoperable semantic 3D city models, ISPRS Journal of Photogrammetry and Remote Sensing, 71, 12–33 (2012)
11. K. W. Chen and L. Norford, Developing an Open Python Library for Urban Design Optimisation - Pyliburo, in Building Simuation 2017, San Francisco, USA (2017)
12. BCA Singapore, BCA Green Mark for New Non-Residential Buildings Version NRB/4.1, (2013). at <http://www.bca.gov.sg/greenmark/others/gm_nonresi_v4.1_revised.pdf>
13. EnergyPlus - Weather Data by Region, (2017). at <https://energyplus.net/weather-region/southwest_pacific_wmo_region_5>
14. C. S. B. Grimmond and T. R. Oke, Aerodynamic Properties of Urban Areas Derived from Analysis of Surface Form, Journal of Applied Meteorology, 38, 1262–1292 (1999)
15. S. J. Burian, M. J. Brown, and S. P. Linger, Morphological Analyses Using 3D Building Databases: Los Angeles, California, Los Alamos National Laboratory (2002)
16. P. Janssen, K. W. Chen, and C. Basol, Iterative Virtual Prototyping: Performance Based Design Exploration, in 29th eCAADe Conference Proceedings, Ljubljana, Slovenia (2011) at <http://cumincad.scix.net/cgi-bin/works/Show?ecaade2011_074>
17. G. J. Ward, The RADIANCE Lighting Simulation and Rendering System, in Computer Graphics 94 SIGGRAPH (1994)

18. K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II, in *Parallel Problem Solving from Nature PPSN VI*, Edited by M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. Merelo, and H.-P. Schwefel, Springer Berlin Heidelberg (2000)