Is computer programming beneficial to architects and architecture students for complex modeling and informed performative design decisions?

Rania Labib

College of Architecture, Architecture department, Texas A&M University, USA

rlabib@tamu.edu

Abstract

For the last few decades, digital tools have become an important part of architectural design. Architects and architecture students create their designs using one or more digital applications. Although most of these applications are advanced and can be used for various tasks – such as parametric design and building performance simulations –, they lack some capabilities that are crucial to solving problems that might arise during the design and simulation process. Therefore, programming knowledge is invaluable to those in the field of architecture for customizing digital applications to perform and automate tasks that are out of the scope of built-in functions.

In this research study, detailed examination of various case studies of custom-coded design programs is discussed. The programs written in Python proved to be crucial to organizing visual scripts by making them less confusing and more efficient. In addition to using computer programming within visual scripting environments, this paper presents a case study of computer programs written in Python, JavaScript, and HTML that were used to organize an enormous amount of data produced by hundreds of glare simulations. The custom programs performed the process of organizing, analyzing and visualizing large amounts of data in a time-efficient manner, thereby facilitating informed decisions for better performative design.

Keywords: computer programming, Python for Grasshopper, data visualization, programming for architecture students, programming for architects, programming for high performance building design

1. Introduction

Computational tools and techniques have been a fundamental part of architectural design in recent decades. They have enabled architects to deal with forms that previously could not have been drawn or built and that require non-standard engineering methods for their fabrication. For example, there has been elevated attention toward parametric modeling in the last few years. This is mainly due to the emergence of visual parametric modeling tools, which hide the algorithmic complexity of a model behind an easy-to-use visual programming interface that can be manipulated without programming skills. These programs decrease the technical skills required to compute and contribute to their widespread adoption in architectural design. Parametric tools facilitate the exploration of alternative designs within a single model by using parameters and formulae to control the geometric and constructive aspects of architectural models [1].

The integration of digital and computational tools in architectural design has become a necessity. In recent years, digital computational methods – such as those found in parametric design, energy simulation, daylight calculations, and environmental analysis – have been adapted by both architectural firms and architecture schools. Consequently, many educators and researchers have identified the need for architecture students to become familiar with the rapidly-developing tools known as parametric [2] [3], computational modeling tools[4] [5], or algorithmic programs [6] [1]. As such, architects and architectural students are increasingly adapting to the use of these tools. Although these software have proven to be crucial in completing difficult tasks – be that modeling complex geometries or simulating performance –, some problems may be beyond their reach. Computer coding and scripting can be invaluable in addressing these challenges by customizing the built-in functions within various digital tools, thereby allowing users to perform difficult tasks that were not possible before. Additionally, computer coding can be used to automate repetitive tasks, reducing the time required to perform these large, time-consuming processes. Furthermore, computer scripting can be used to solve design problems and can even analyze and visualize huge datasets such as those data required in energy modeling.

While it is clear that computer programming and scripting are considered essential tools in architectural practice, few universities across the US introduce computer coding to their undergraduate and graduate architecture programs. Although there are many books, primers, and online video tutorials to teach architects and designers computer programming and scripting [7] [8] [9] [10], there has been an extreme lack of computer programming and scripting classes offered in university settings within architecture schools. This is a significant problem as it becomes very difficult for architects to navigate the ample resources found both in print and online to teach themselves programming in non-school setting.

This paper discusses the benefits of teaching computer programming and scripting to architectural design students within their program's curriculum; it would enable these future architects to customize their digital tools for a better design process and to prepare students for a computing era that will ultimately facilitate a new generation of performative, responsive, and smart buildings and cities.

2. Extending visual scripting capabilities

Visual programming dates back to the 1960s, when the GRAIL system was introduced to the world. The GRAIL system allowed computer users to input data using a graphic interface and a tablet that was very similar to modern graphic tablets [12]. Interest in parametric and generative design has increased over the last two decades due to the introduction of digital fabrication machines, which allow the fabrication of complex, free-form architectural geometries that were not possible to model before. Parametric and generative modeling have been adopted by many architects and architecture students due to the recent availability of visual parametric modeling tools, which disguise the algorithmic complexity of models with a clear visual programming interface. These changes decrease the technical skills required to use these programs and contribute to their universal adoption in architectural design. The integration of such tools in architectural practice has been expanding since 2003, when Bentley Systems introduced Generative Components, a graphic-based generative modeling software used within MicroStation [13]. Many universities across the United States adopted generative design tools into their architecture courses after the release of Grasshopper, a visual programming plugin for Rhinoceros [14]. Generative modeling tools

help designers explore various model configurations within one design by changing parameters that are input by the user. On many occasions, however, students end up with a geometry that differs from their desired model for a number of reasons, such as the complexity or number of design parameters. When this occurs, it can be very challenging for students to edit the script to achieve the desired model. In many cases, students abandon their initial design ideas and use the resulting model instead. To prevent this, it is crucial to explore ways of overcoming such challenges.

2.1 Computer coding for visual scripting organization and simplification

Although visual programming can simplify complex programming with drag and drop components, many challenges can arise during the modeling process. For example, when designers run into modeling problems due to missing or inappropriate input, they may not be able to fix the model due to the complexity of the visual script file. Such files often contain hundreds of components that are connected to each other leading to creating a visual tangle [15]. Moreover, it can be difficult to understand the structure of the file when the person editing it is not the original author, as is common practice in the collaborative work environment of architecture education.

Visual scripting environments such as Grasshopper usually contain built-in components that are used for modelling, where each component contains hidden code that cannot be edited by the user. Those components usually receive several user inputs that are pre-set by the hidden code and produce a specific output. A variety of modeling tasks are required to produce medium or complex geometries within visual scripting environments. However, many of the visual components in Grasshopper perform only one task and, therefore, an enormous number of components are often needed to complete the desired model. This, in turn, leads to a visual tangle that can cause confusion, particularly for those inexperienced with programming.

To overcome these challenges in a visual scripting file, custom computer code can be used to create visual components that perform multiple tasks at a time, thus reducing the number of components needed for modeling. This makes the script organized and simple to understand by not only the original author but also other people who are working with him or her.

An example is illustrated in Figures 1, where a pyramid is imported into a visual scripting environment. To perform simple transformation tasks on the pyramid – such as copy, morph, or mirror –, the program needs information about the individual surfaces that make the pyramid or the point coordinates that define each surface. A script that contains 24 Grasshopper components and 16 panels for user inputs is used to extract the 4 surfaces and the vertices of theses surfaces (Figure 1). To simplify the script and better organize it, a custom component was created with Python to extract the information obtained using the visual script illustrated in Figure 1. The information extracted included the coordinates of one vertex, one individual surface, three vertices of one surface, and all surfaces of the pyramid. Figure 2 show this custom component and the Python code inside it. The code successively explodes the pyramid into surfaces, edges, and vertices, then stores all resulting data into organized lists. These lists can be called as component outputs that can then be used in the modeling process. Figure 2b shows that all four outputs are identical to those obtained in Figure 1. In this example, the author combined 40 components and panels into a single component with multiple outputs, thus creating a simplified, organized and easy to understand script.

2.2 Computer coding for complex tasks within visual scripting

In addition to simplifying and organizing a visual script, computer coding can also be used to perform tasks that are otherwise impossible to do in a visual scripting environment. One example is the process of recursive modeling. Recursion in computer science is “a process where the final result of a solution to a specific problem depends on the solution of smaller instances of the same problem” [16]. A Recursive function is a function that can be used to call itself, this type of function is impossible to call within visual scripting software packages because it is impossible to get the result of the function and feed it back into the same function. By using a programming language, however, one can create such functions by having them define their output based on outputs from previous versions.

Figure 3 shows an example of a script written in Python that contains a recursive function. This function is used to model smaller, modular, three-sided pyramids through various iterations. The code is written inside a custom-made Grasshopper block that takes user input and produces an output defined by the code, which in turn contains two functions: mirrorBrep and recursiveBrep. The function mirrorBrep defines a process where a simple geometry – in this case, a simple three-sided pyramid – is exploded and reflected around one of its surfaces. The function recursiveBrep defines a process where the pyramid is reflected repeatedly around the newest pyramid created to form a more complex geometry. Figure 4 shows the resulting geometry at different iterations, where the structure becomes more complex as the number of iterations increases. Although the process defined in the first function can be accomplished using drag and drop components, the amount of components needed to perform such a process are excessive and would lead to a confusing visual script. Additionally, the Python code in the second function completes a task that is impossible using drag and drop components. Therefore, custom coded components are crucial in order to organize and simplify visual scripts, as well as to extend the capabilities of off-the-shelf visual scripting environments.

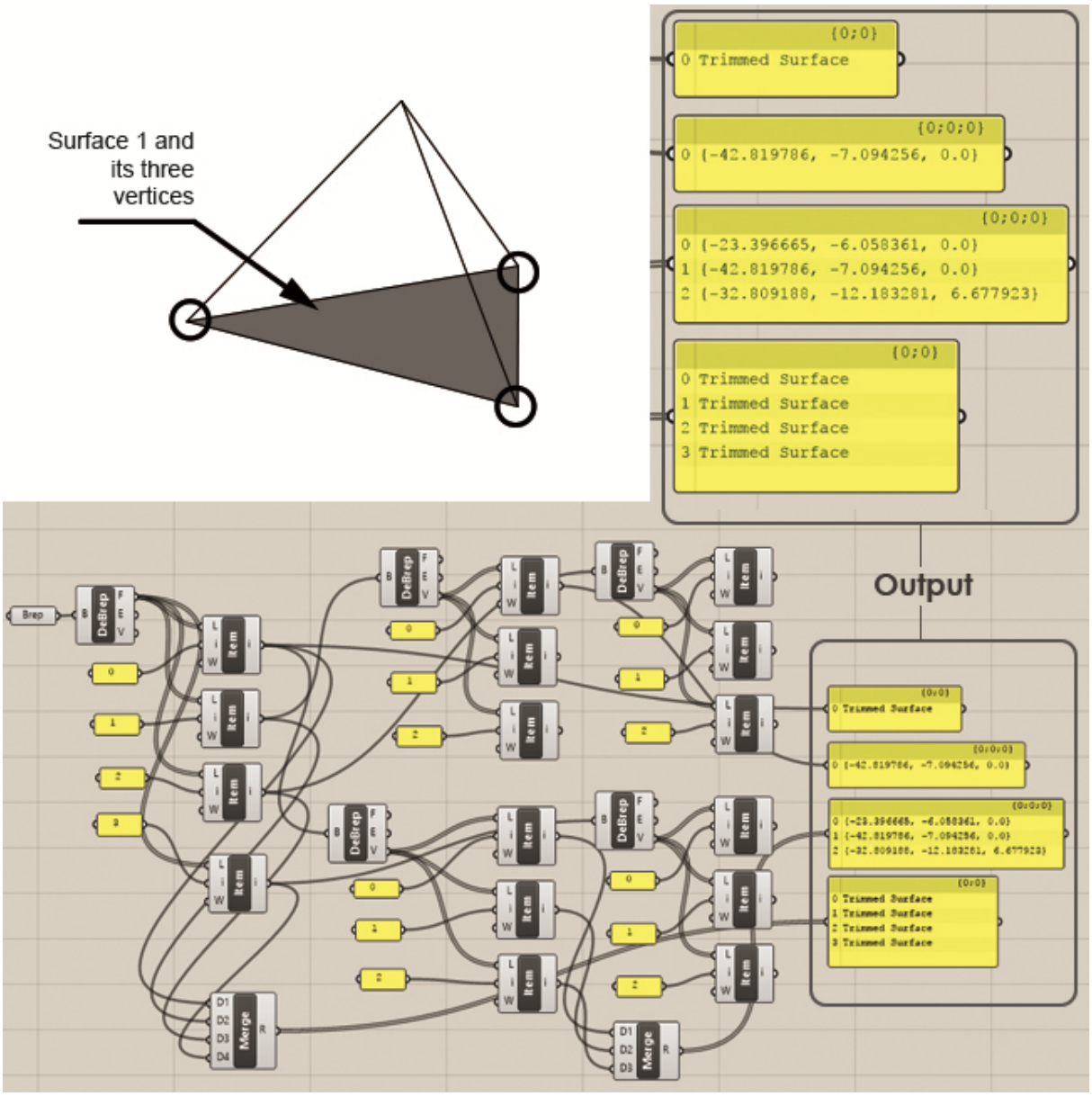
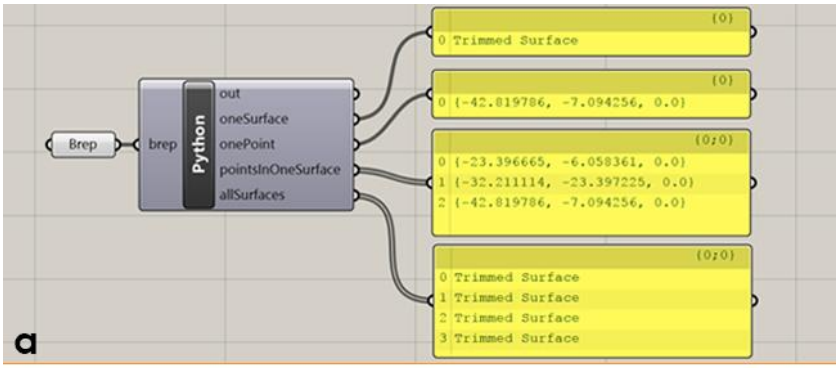


Figure 1: Grasshopper file to explode a pyramid into three surfaces and 4 points.



Grasshopper Python Script Editor

File Help

```
1  import rhinoscriptsyntax as rs
2  pts=[[],[],[],[]]
3  allSrf = rs.ExplodePolysurfaces(brep)
4  for y in range(0,4):
5      border = rs.DuplicateSurfaceBorder(allSrf[y])
6      lines = rs.ExplodeCurves(border)
7      for line in lines:
8          points = rs.CurveEndPoint(line)
9          pts[y].append(points)
10 onePoint = pts[0][0]
11 oneSurface = allSrf[0]
12 pointsInOneSurface = pts[3]
13 allSurfaces = allSrf
```

b

Figure 2a: Custom-coded Grasshopper component used to explode the pyramid, the output obtained is the same output obtained in figure 1.
Figure 2b: The Python code inside the custom-coded component

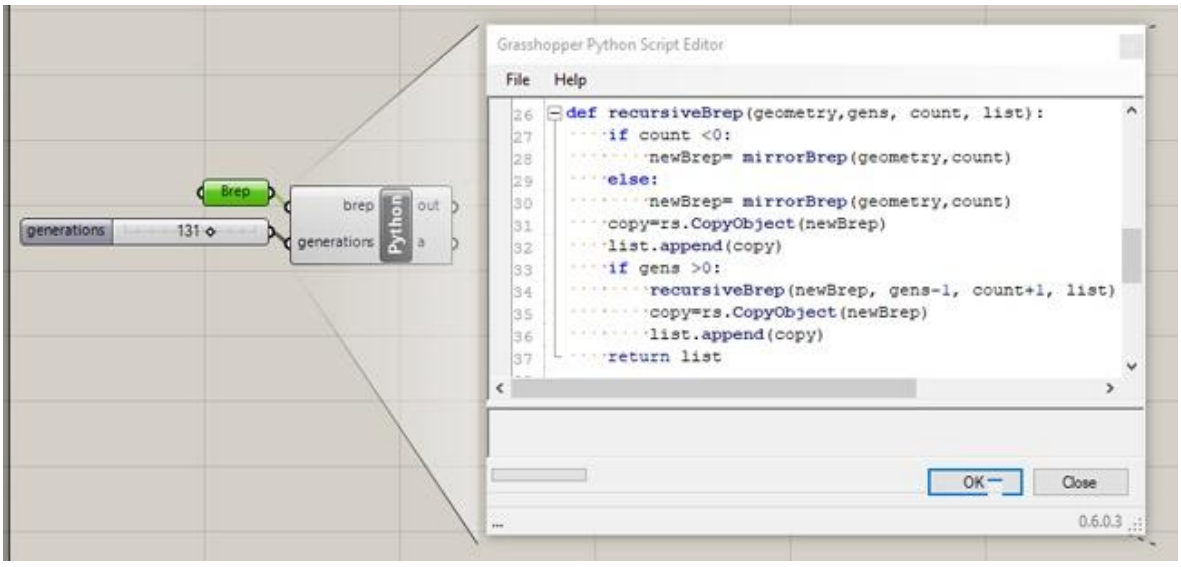


Figure 3: Custom-coded Grasshopper component used for recursion, the code shown illustrates the function recursiveBrep that is used for recursive modeling

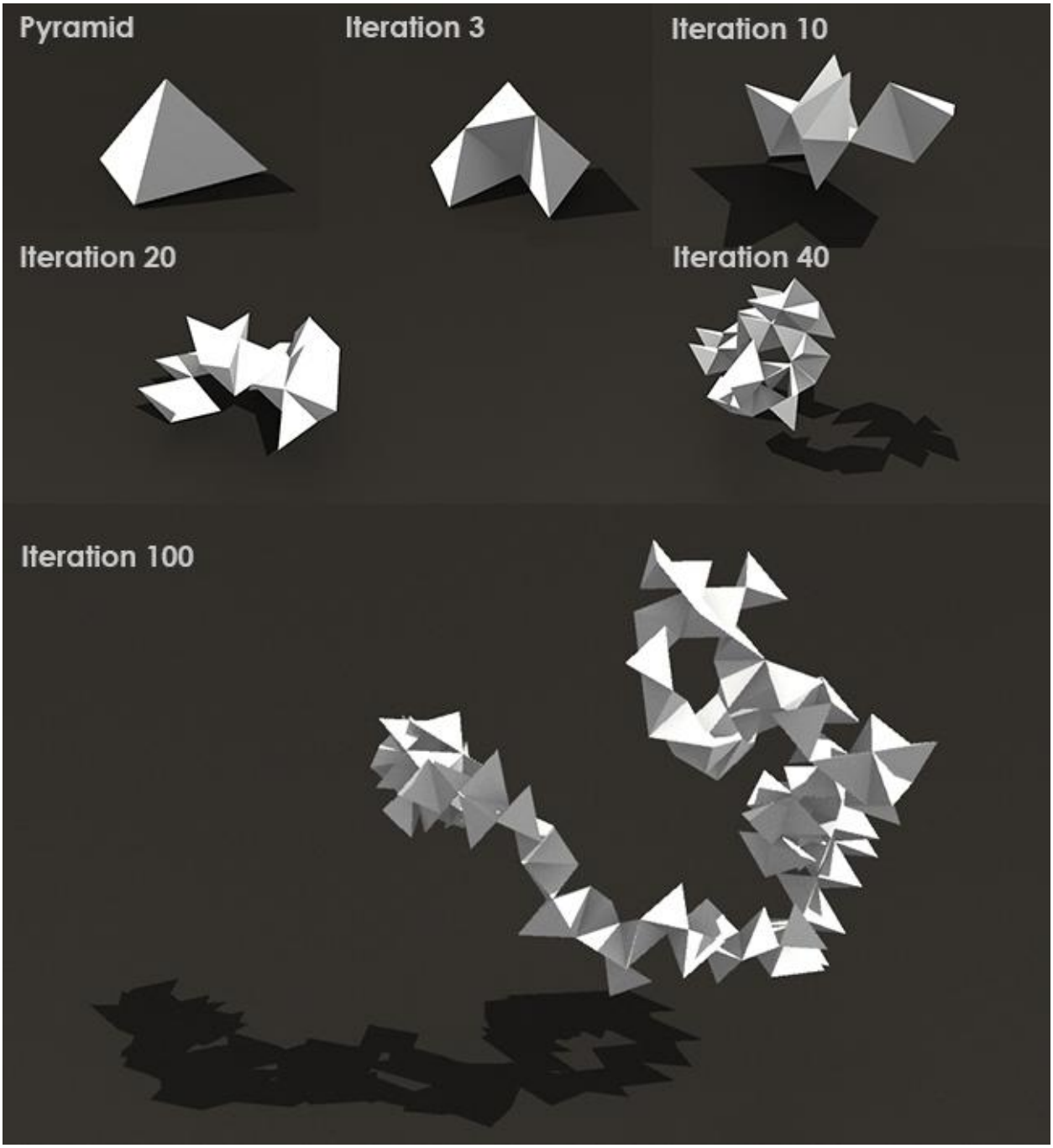


Figure 4: pyramid recursion modeling at different iterations

3. Extending the capabilities of building performance simulation tools

Building energy consumption accounts for 30% of global energy consumption, and this provides designers with a great opportunity for energy conservation [17]. To address this issue, architectural students and architects have started to embrace a new set of increasingly popular building simulation tools to support energy-efficient design. Programs that simulate building performance are powerful tools for examining a number of aspects such as energy consumption, thermal comfort and daylight performance. The number of these simulation tools available to architects has been rapidly increasing in

recent years. They differ in many ways, from their behind-the-scene engine and user interfaces, to their output format and ability to communicate with modeling software. Architecture students and architects are faced with a steep learning curve when using new simulation tools. Therefore, they tend to select tools that offer a clear graphic interface to run building performance simulations. However, these interfaces do not make all the features of building analysis available to everyone. Therefore, it is crucial to understand how these tools are designed and how they communicate with their back engine in order to have accurate and fully-realized building simulations [18]. Most of the tools currently available are open source tools, meaning that users with programming skills can modify the underlying source code to tailor the software to their needs. Architecture students in integrated design studios that combine design with building performance are often tasked with a design problem that has an emphasis on one or more building performance criteria. Students tend to use simulation tools that are widely available as plugins for commonly used modeling environments. By using these plugins, architects and architecture students can find themselves with huge amounts of data and files, making it very difficult to examine the performance of a proposed design. The following section will examine a case study in which programming knowledge helped the author visualize the data produced by a simulation tool.

3.1 Performance simulation results analysis

As previously noted, simulation tools produce data files that may contain an overwhelming amount of numbers. Although these files hold the results of potentially helpful simulations, it is difficult to read the data contained within them, let alone visualize it. To address this issue, computer programming languages such as Python and MATLAB can be used to organize, analyze, and even visualize data for better understanding of the results produced by simulation software. This ultimately leads to better assessment of the proposed design's performance. In a case study where the author of this paper needed to investigate the effects of glare on the visual comfort of a building's occupants. This building had 20 floors with 10 offices each, totaling 200 offices affected by the refection of the sun off another building's highly reflective façade. To better address the visual comfort of the workers stationed in these 200 rooms, it was necessary to perform a daylight glare analysis for each of these views. This analysis was repeated for each view every time the façade optical material is changed. Glare results were produced by the simulation tools Honeybee and Ladybug [19] for each room, and these results were subsequently stored in individual files that had nearly 8,640 hourly daylight glare probability (DGP) values. Glare simulations were performed 4 times, producing 800 files or a total of 6,912,000 DGP values.

To understand the data, various boxplot graphs for each view were needed to show the maximum, minimum, and mean values for each specific hour of the day through each month for an entire year. It was clear that manually extracting the information needed for these graphs from 800 files would be a very difficult and time-consuming process. Therefore, the author wrote a set of computer programs using Python to efficiently perform the following processes:

1- Check and validate the files to prepare the results for analysis. This step was needed due to unexpected interruptions in the simulations that produced files with missing data.
2- Extract the minimum, maximum, and mean DGP value for each view at 15:00 every day for the entire year. It later stored the extracted values in a custom Excel sheet that allows the automatic creation of boxplot graphs. Figure 5 shows some of the lines of the code that was used to complete this process.
3- Extract the minimum, maximum, and the mean DGP value for each view across an entire month, and then store these values in an Excel sheet that automatically creates boxplot graphs. Figure 6 shows a sample boxplot.

As shown, the custom Python code was crucial to extracting, organizing and analyzing the data required. This data helped the author to better assess occupants' visual comfort and the façade performance for an informed facade design decision.

```
for root, subFolders, files in os.walk(rootdirl):
    for file in files:
        if file.endswith('dgp'):
            with open(os.path.join(root, file), 'r') as fin:
                fileNameSplit= file.split('_')
                viewNumber = fileNameSplit[0]
                for line in fin:
                    line_text= line.split(" ")
                    month = line_text[0]
                    month_int=int(month)
                    hour= line_text[2]
                    hour_int=int(float(hour))
                    dgp_text= line_text[4]
                    dgp= float(dgp_text)
                    apply_dgp_to_month_col()

for b in range (1,13):
    s=numpy.percentile(m_list[b], 75, interpolation='higher')
    t=numpy.percentile(m_list[b], 25, interpolation='lower')
    m=statistics.median(m_list[b])
    med_list.append(m)
    s_list.append(s)
    t_list.append(t)
```

Figure 5: Partial view of the code used to iterate over 800 files to extract the information needed to create boxplots

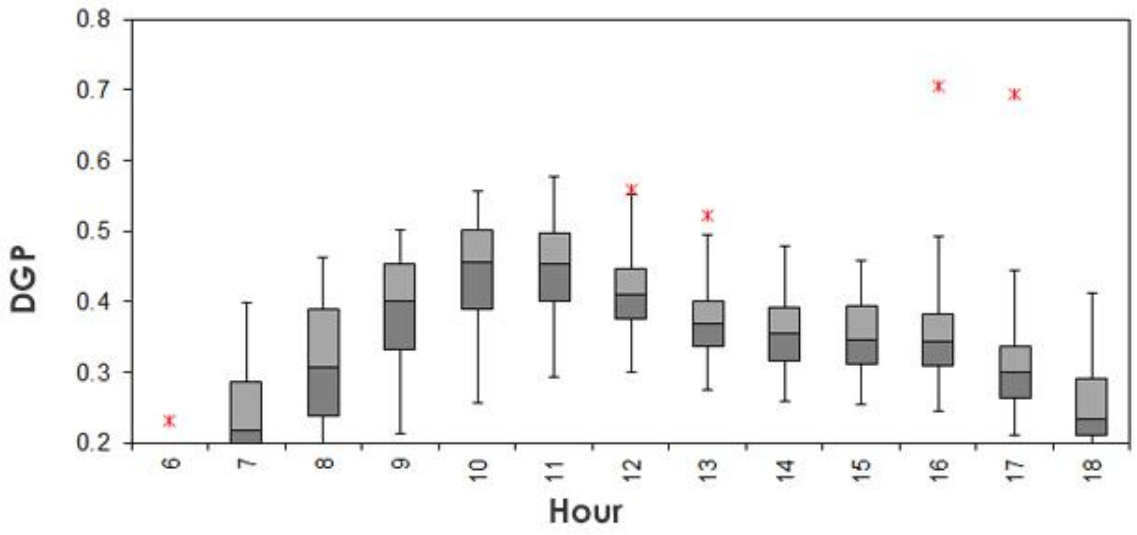


Figure 6: An example of the boxplots easily created by the Python program

3.2 Simulation results visualization

As shown above, Python is not only a great tool for data analysis but also for data visualization. Its abilities can be greatly improved when used in conjunction with JavaScript to produce easy-to-understand plots such as interactive pie charts, polar charts, wind roses, and heatmaps.

For the purpose of visualizing data produced by the simulation tool for the 200 views previously discussed, a custom Python program was written with embedded HTML and JavaScript code; it produced interactive heatmaps that visualized the DGP values of every hour between 6:00 and 18:00 for the whole year. A static image of one such interactive heatmap produced by the custom program is shown in Figure 7. Additionally, the code was incorporated into a Grasshopper component to allow immediate access to the code within a visual scripting environment (Figure 8)

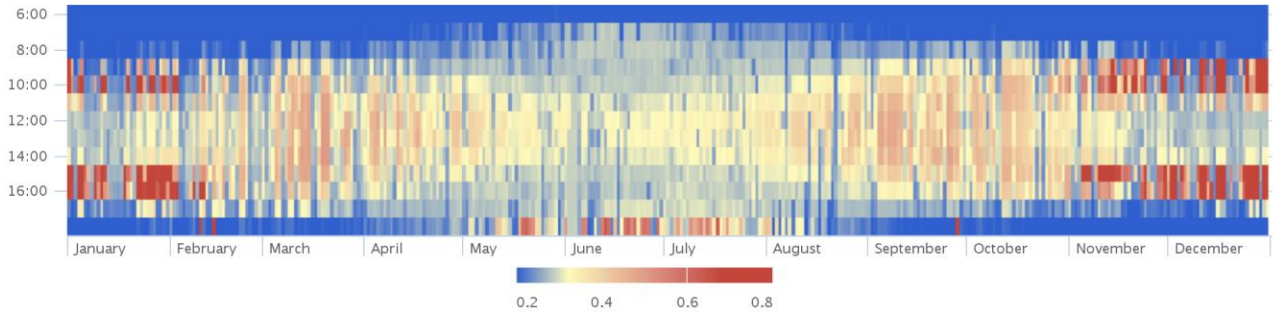


Figure 7: A heatmap produced by the custom-written Python program

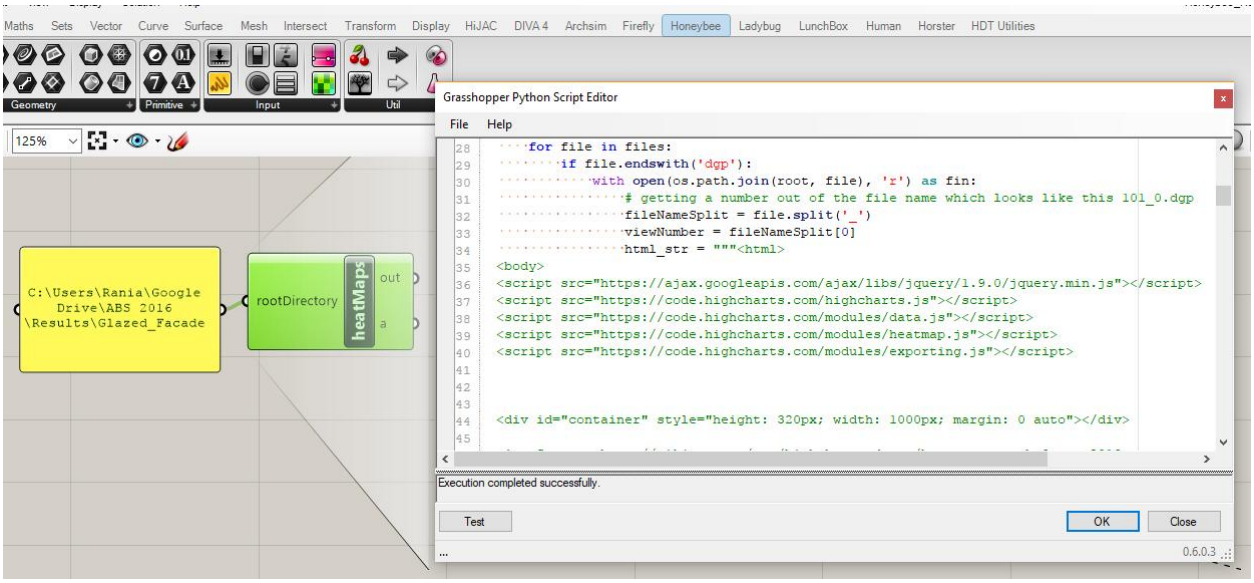


Figure 7: A custom coded component used in conjunction with Honeybee to create heatmaps for all analyzed views contained within the project's folder

4. Conclusion

Architects and architecture students typically use off-the-shelf applications to do the majority of their modeling. However, computer programming can extend the abilities of this modeling software to perform tasks that are impossible to do with an off-the-shelf application. Additionally, computer programming may be the ultimate tool in organizing and simplifying visual script files, thereby leading to a speedy modeling process that helps students and professionals meet project deadlines. Computer programming skills are crucial to make modeling processes more time-efficient for those in the field of architecture. Moreover, computer programming also facilitates the process of organizing, analyzing, and visualizing complex simulation data for building performance that might be impossible to analyze manually. Therefore, custom programs are considered essential to make informed decisions for high performance building design.

In conclusion, computer programming can assist today's architects and architecture students in overcoming the challenges that result from using new digital tools. Integrating computer programming into architectural education could be instrumental to preparing future architects to better organize, simplify, and visualize heir design ideas in an efficient way, and most importantly make informed design decisions.

References:

[1] K. Terzidis, *Algorithmic architecture*, vol. 1. Architectural Press, 2006.

[2] I. G. Dino, “Creative design exploration by parametric generative systems in architecture,” *Metu J. Fac. Archit.*, vol. 29, no. 1, pp. 207–224, 2012.

[3] D. J. Gerber, “Parametric practices: Models for design exploration in architecture.” Harvard University, p. 511, 2007.

[4] A. Menges and S. Ahlquist, *Computational Design Thinking: Computation Design Thinking*. Wiley, 2011.

[5] M. Burry, *Scripting cultures: Architectural design and programming*. John Wiley & Sons, 2011.

[6] P. Coates, *Programming. architecture*. Routledge, 2010.

[7] W. Jabi, “Parametric Design for Architecture,” *Int. J. Archit. Comput.*, vol. 11, no. 4, pp. 465–468, 2013.

[8] M. Jezyk, “Python and Revit | The Dynamo Primer.” [Online]. Available: http://dynamoprimer.com/en/09_Custom-Nodes/9-5_Python-Revit.html. [Accessed: 19-Jun-2017].

[9] McNeel, “RhinoScript Wiki [McNeel Wiki],” *2005*. [Online]. Available: https://wiki.mcneel.com/developer/rhinoscript. [Accessed: 19-Jun-2017].

[10] W. J. Mitchell, R. S. Liggett, and T. Kvan, *The art of computer graphics programming: a structured introduction for architects and designers*. Van Nostrand Reinhold Company, 1987.

[11] M. McCracken, T. Wilusz, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, and I. Utting, “A multi-national, multi-institutional study of assessment of programming skills of first-year CS students,” *ACM SIGCSE Bull.*, vol. 33, no. 4, p. 125, 2001.

[12] T. O. Ellis, J. F. Heafner, and W. L. Sibley, “The GRAIL Project: An experiment in man-machine communications,” RAND CORP SANTA MONICA CA, 1969.

[13] R. Aish, “Extensible computational design tools for exploratory architecture,” *Archit. Digit. age Des. Manuf. Spon Press. New York, NY*, 2003.

[14] R. McNeel, “Grasshopper generative modeling for Rhino,” *Comput. Softw. (2011b), http//www. Grasshopp. com*, 2010.

[15] R. Woodbury, *Elements of parametric design*. Taylor and Francis, 2010.

[16] P. L. Pirolli and J. R. Anderson, “The role of learning from examples in the acquisition of recursive programming skills.,” *Can. J. Psychol. Can. Psychol.*, vol. 39, no. 2, p. 240, 1985.

[17] IEA, *Key World Energy Statistics 2014*. Organisation for Economic Co-operation and Development, 2014.

[18] T. Maile, M. Fischer, and V. Bazjanac, “Building energy performance simulation tools-a life-cycle and interoperable perspective,” *Cent. Integr. Facil. Eng. Work. Pap.*, vol. 107, pp. 1–49, 2007.

[19] M. P. Mostapha Sadeghipour Roudsari, U. S. A. Adrian Smith + Gordon Gill Architecture, Chicago, M. S. Roudsari, M. Pak, and A. Smith, “Ladybug: a Parametric Environmental Plugin for Grasshopper To Help Designers Create an Environmentally-Conscious Design,” *13th Conf. Int. Build. Perform. Simul. Assoc.*, pp. 3129–3135, 2013.