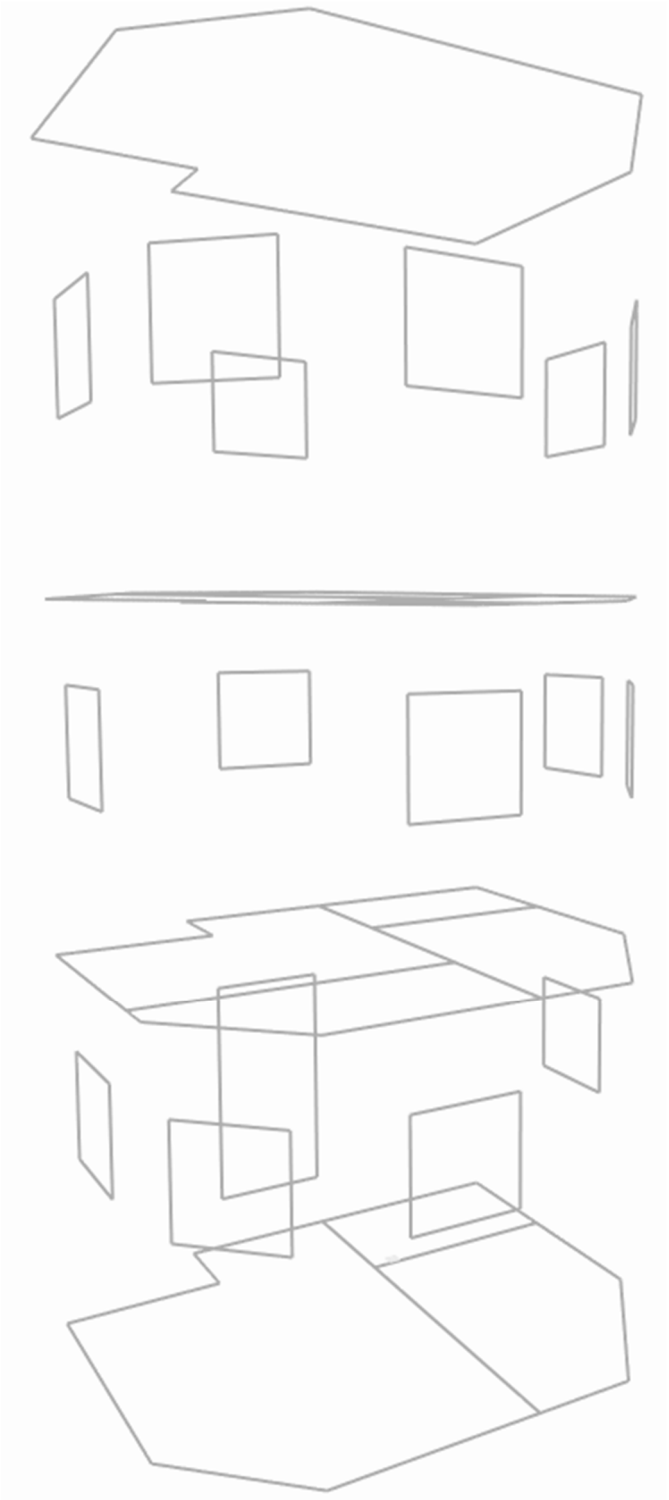
AUTOMATION OF INFORMATION FLOW FROM REVIT TO BSIM USING DYNAMO



MANAGEMENT IN THE BUILDING INDUSTRY

4TH SEMESTER, MASTER THESIS

Pavel Pavlov
ppavlo14@student.aau.dk

AALBORG UNIVERSITET
STUDENTERRAPPORT

Education

Fourth semester of M.Sc. in

Management in the Building Industry

Title

Automation of Information Flow from Revit to BSim Using Dynamo

Report Type

Semester Report – Autumn, 2015

Author

Pavel Pavlov

Project period

Sep 2015 – Apr 2015

Supervisor

Kjeld Svidt

Number of pages 70

Synopsis

This paper is based on the work in collaboration with a student group from Indoor Environmental and Energy Engineering programme in aim for automation of the connection between Revit and BSim software.

The automation process consists of two steps: exporting the Revit model geometry and relevant information to Excel file, and modifying the excel file to prepare for the BSim analysis.

The work presented in this paper describes the first step of the automation process – the design of the Dynamo script used to export the relevant information from Revit to Excel and the development process.

Signature

Pavel Pavlov

Table of Contents

01. Preface

This dissertation is based on the work of developing a Dynamo script, as a part of an automation process for information transfer from Revit to BSim. It is written as graduation requirement for Management in Building Industry programme in Aalborg University (AAU) by me, Pavel Pavlov. The work has been carried out in collaboration with a group of students in Indoor Environmental and Energy Engineering programme in AAU, consisting of Frederik Søndergaard Mikkelsen, Emil Greve Bonde, and Mia Buhrkal-Donau, which is working on the second part of the automation process, scheduled to be finished later this year.

In this paper I am presenting the process of development and describing the inner works of the script, as well as guidance to the users of the script and anyone intending to continue the work on the script and improving or optimising. As such, the paper is intended to be read by anyone interested in using the script or further developing the project, as well as anyone interested in application of Dynamo in Revit projects, or getting higher level of proficiency with Dynamo and new ideas on methods of work within the software. The thesis is written in a way that allows reading only the relevant to the user chapters without having read everything else. In the places where additional information is available elsewhere in the paper, such references are leaved.

As Revit enthusiast, it was a great experience for me to work with Dynamo, even though its still early stage of development and respective quirks and limitations, and am proud of the result of my efforts. I would like to express my gratitude to Frederik, Emil, and Mia for their cooperation throughout the semester and help in difficult times when feeling stuck with the progress. I would also like to thank my supervisor Kjeld Svidt for his guidance and support. His similar enthusiasm for tools like Dynamo was very contagious. Also thank you to Rasmus Jensen, supervisor of Frederik, Emil, and Mia, for his great input in our common discussions.

I hope you enjoy reading.

02. Introduction

In today's era in the construction industry, there is a rapid change happening. More and more countries and companies are transitioning from conventional workflow to BIM – a modern way of managing project information in an intelligent way that has many benefits over older methods.

What is BIM?

BIM (Building Information Modelling) is a process. A process of managing information from the idea of a construction project, to potentially decades after it has been executed and handed over. It is widely considered to be much more efficient in many areas than conventional construction processes in variety of ways by providing integration and thus enabling different disciplines in the process to communicate with much less effort and errors, along with new tools for the professionals, which can be used to do their work better and effectively. The benefits of BIM are many but could be summarised in several points:

- Faster design process
- Fewer mistakes in cross-disciplinary communication and fewer errors on construction site
- Enhanced building energy performance, due to availability of relevant information from earlier on in the design
- Improved life cycle and waste management
- Better overall client satisfaction, due to being able to make more informed decisions earlier in the process

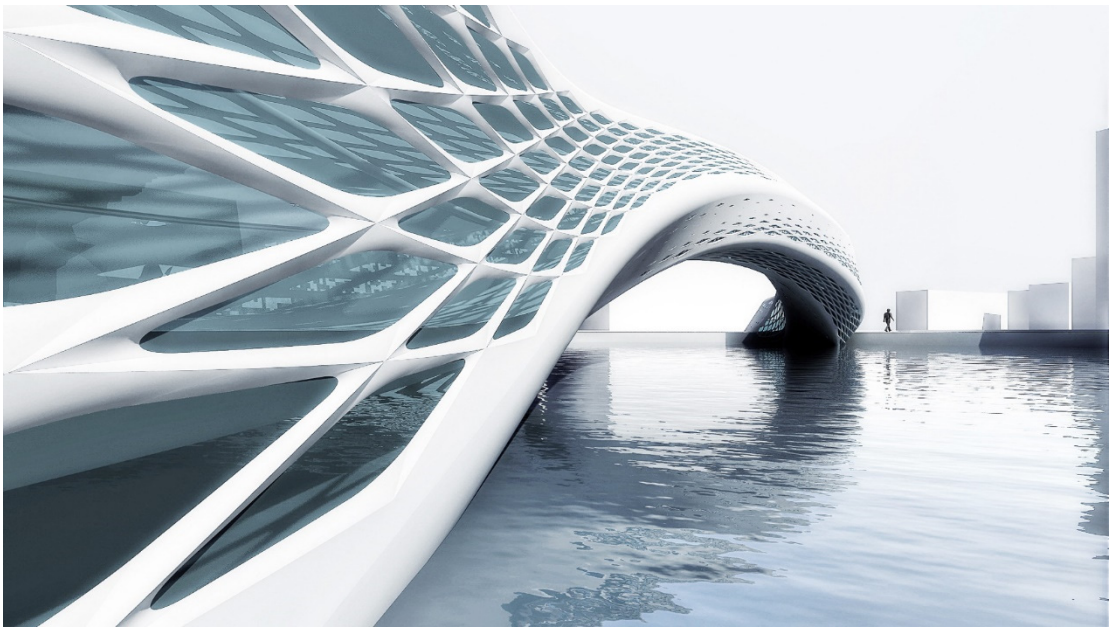


Figure 02-1 Parametric Architecture (Gillen, 2015)

Building Information Modeling (BIM) is an intelligent 3D model-based process that equips architecture, engineering, and construction professionals with the insight and tools to more efficiently plan, design, construct, and manage buildings and infrastructure. (Autodesk, 2016)

At the core of the BIM process lies an *intelligent* 3D model. Intelligent, because it contains information that is accessible and manageable by compatible software. That allows architects, structural and energy engineers, contractors, owners and clients, to have access to the same centralised information.

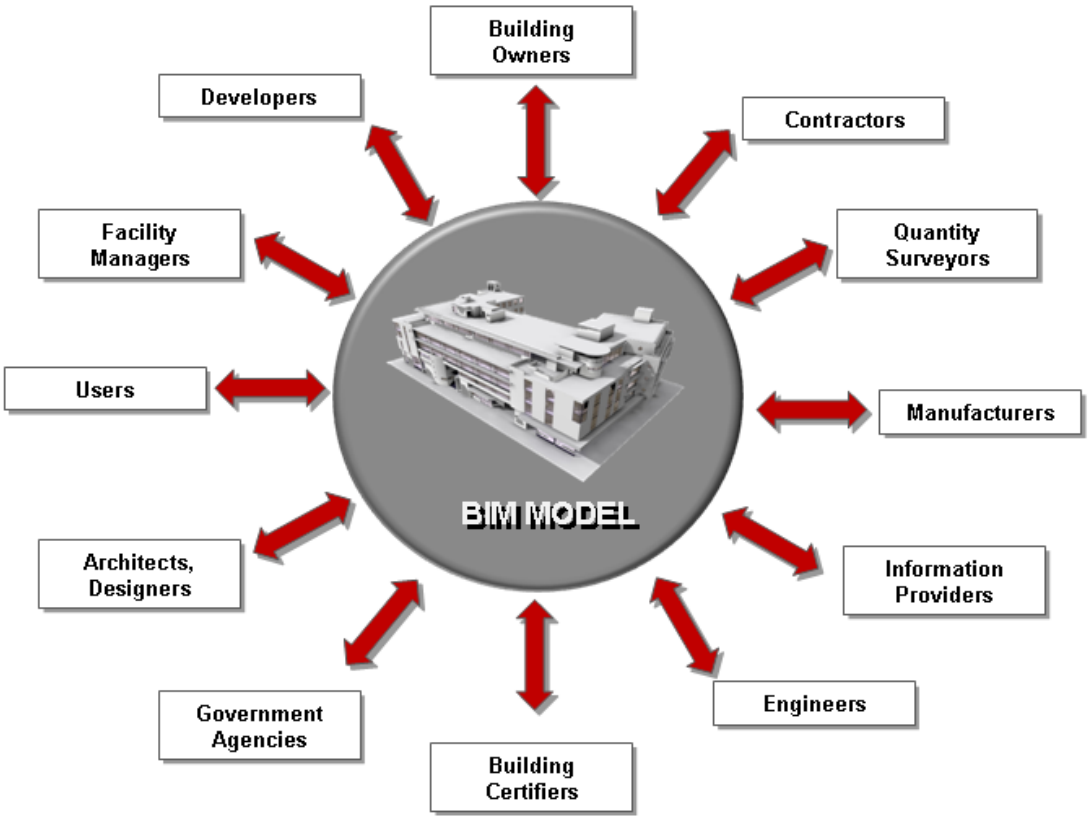


Figure 02-2 BIM as a Platform for Communication (Graphisoft, 2016)

As the needs for more energy efficient construction raises, so does the focus on more accurate energy analysis of the building in the early stages of the project. Therefore the link between the modelling and analysis software is very important. As the software develops and new ones often comes to the scene, the question of managing the communication between these software (interoperability) is of constant importance. As Epstein puts it, “Interoperability is the ability of different systems (hardware or software) to work together” (Epstein, 2012). Software developer companies have been making great progress, but there is still more to be desired. While link between many modelling and energy analysis software is available, there is one missing between two of the most popular tools for modelling and energy analysis in Denmark – Revit and BSim. This lack of communication issue is the root of the problem that this student projects is aiming to solve.

03. Problem Formulation

Typically the energy analysis of a building is performed by the engineering company, but it could be a hired consultant specifically for that job – that would be more feasible for larger scale projects. The workflow of the engineer performing the energy analysis usually involves taking the architect's model of the building, creating an energy model that will be used for the simulations, and performing analysis on the results of the simulations, creating suggestion on changes on the model such as used materials, size and location of windows, shading and others. It is evident that a link or at least import/export communication between the architectural model and the energy analysis model is important. A big problem in Denmark is that two very popular software – Revit and BSim cannot do this communication. So typically energy engineers need to manually remake the geometry and information from the architectural model they need for the simulation, and that requires a lot of resources. These restrictions cause negative effects on the final outcome for reasons such as not being able to analyse wider variety of geometrical options due to the need to manually recreate that geometry every time. Ideally the Revit model would be exportable to BSim and these changes could be made in the model, instead of manually. If such link existed, feedback from the simulation could be fed back into the original model in the form of a schedule or colour-coded floor plans with legends.

The author of this thesis and a group of students from Indoor Environmental and Energy Engineering programme in AAU have decided to work together on creating this missing link. The chosen method is to use visual programming extension for Revit called Dynamo to extract the needed information to an Excel spreadsheet, then via VBA script in Excel to process that information to include data, related to the energy analysis, to an XML file, which in the end is used for performing the simulations in BSim. The goal of the flow is to automate the manual work that usually accompanies these tasks. The overall work has been split in two parts:

- Export model geometry, room, and construction information from Revit to Excel via Dynamo
- Processing that Excel file and automating BSim simulations, which involves including information regarding HVAC systems, people load, equipment load, construction properties and weather information.

The first part of this process is taken by the author of this paper and thus the main focus and problem of this report is as follows:

Problem Formulation:

> *How to export information from a Revit building model to Excel file, formatted in a BSim compatible schema, to allow automation of energy simulations?*

As mentioned above, the software used for the export is Dynamo, which is presented in more details in the next chapter. This raises several sub-questions to the main problem statement:

- What are the limitations of the software?
- What are the requirements for the output file?
- What is the general approach and structure of the script?
- Are there any specific guidance going to be needed for the users?

These sub-questions are answered in the following chapters, along with description of the script development process, discussions on some major decisions that have taken place, balance between functionality and computational time and hardware demands, and creator's recommendations for future development.

Project Limitations

While the interface of Dynamo allows for visual programming, there are some limitations to what is possible with the current version of the software, which are described in chapter 05.3. Software Limitations. There is an option to include blocks of code, written in Python language, in the visual environment to supplement the overall script. That allows to overcome some limitations, however, the script is created without the use of coding in Python language, due to the lack of author's skills in that language. This also serves as a good representation of the abilities of the software to develop algorithms without much textual scripting. The few exceptions of this are several places where "Design Script" language is used for clarity and saving canvas space. Design script is a scripting language developed specifically for Dynamo and following a syntax very similar to the visual programming nodes. All code written in Design Script in this project can be translated to visual nodes. More details on visual programming and nodes can be found in chapter 05. Dynamo – Software Possibilities and Current Limits.

04. Methodology

The methodology this thesis is based on is called Contextual Design. It is developed by Karen Holtzblatt, psychologist, and Hugh Beyer, developer, in the early 1980s (Interaction Design Foundation, 2015). They created the methodology incorporating practices from their fields of work. Contextual design is a design process that focuses on finding the needs of the users by establishing strong communication with them, analysing the collected information, use the interpretation of the data to develop prototypes, and iteratively test and improve on those prototypes by continuing the communication with the users. Contextual design is primarily used in computer information and IT systems, however parts of the methodology is used in various different fields (Interaction Design Foundation, 2015).

The Contextual Design methodology comprises of two major parts: Finding requirements and solutions, and defining and validating concepts. Refer to Figure 04-1 Contextual Design Process :

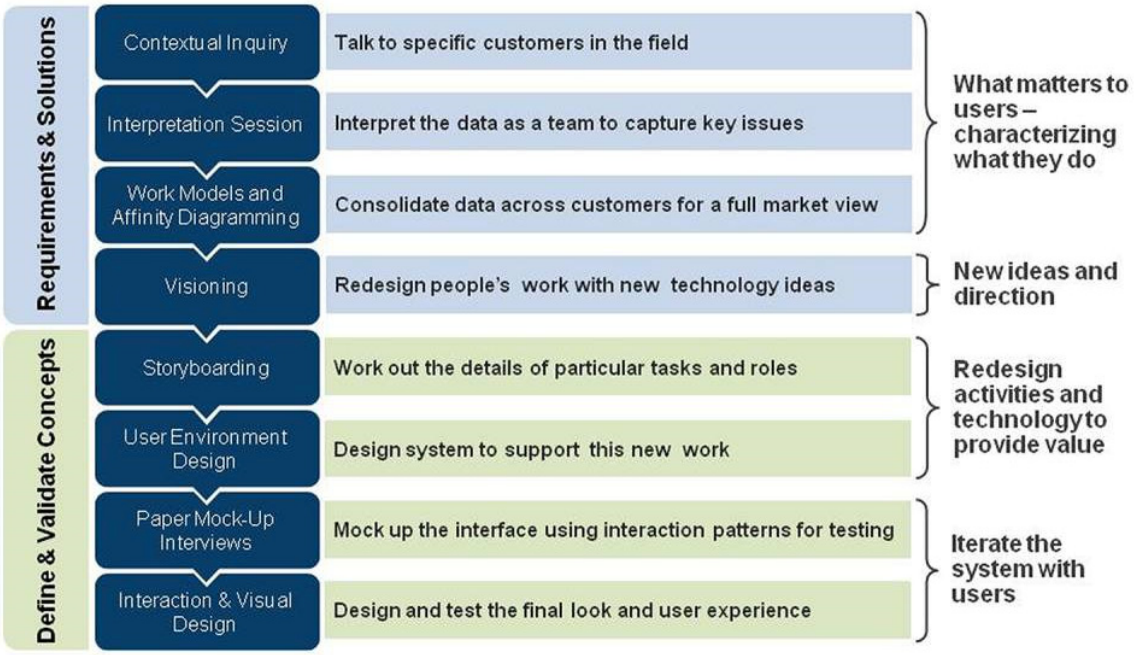


Figure 04-1 Contextual Design Process (Interaction Design Foundation, 2015)

Requirements and Solutions

The first part of the process is defining the requirements of the people involved in the process – end users, indirect users, managers – by establishing communication. First in the process is defining what they do and what their needs are by analysing the present work practices. The goal is to attain a good understanding of the users as they are. This comprises of the first three tasks on Figure 04-1 above. Second in the process is, based on the attained understanding of the work practices, to create a plan of transforming some tasks to make the work practices more efficient.

Defining and Validating Concepts

The second part of the Contextual Design process is developing and testing a system to implement the changes, defined in the plan for transforming the tasks. That involves the four steps, shown in green in Figure 04-1 Contextual Design Process , and namely: defining details of tasks and roles, designing new system for the changes in the work, developing a prototype, and testing the prototype against the set criteria.

Project Methodology

Most points of the Contextual Design process can be successfully applied to the project, and thus it has been chosen as basis for the used methodology. Since the topic of the project is more specific than the common projects the Contextual Design steps would be used for, the main focus of the project is on the second part of the Contextual Design process, and namely – the development of a working system to improve the efficiency of the working practices. The exact steps of the process are as follows:

1. Defining Requirements of the engineers performing the energy analysis
2. Creating a Plan for improving the current work practice
3. Inventing a System to implement the plan
4. Developing Prototype of the system
5. Testing the prototype against the requirements and refining

As the problem of the project can be defined early in the process, the majority of the work on the project is spent on step 3 to 5 from the described above – inventing a system, development of prototype, and testing.

The requirements of the engineers performing the energy analysis are improvement on the tedious process of manual work to remodel the Revit model to BSim. And, as main subject of the project, the plan for improving the work practice is automating that process. The system for automating the process is the workflow between Revit and BSim using Dynamo and Excel. As mentioned in chapter 03 Problem Formulation, this project is based on the development of the first part of the system (exporting data from Revit to Excel), thus the prototype for the current thesis is the Dynamo script, which has been tested on multiple Revit input files.

The details of the development of the system, prototype, and testing are described in the following chapters.

05. Dynamo – Software Possibilities and Current Limits

This chapter introduces the software used to develop the system of automating the workflow – Dynamo. It discusses the possibilities and current limits of the software and gives an inside of how Dynamo is useful for the process.

Dynamo is a relatively new software. At the time of writing, it is just now reaching version 1.0, being in very early stage of development. Seeing as the script is created using an older version (0.9.1), the chapter does not focus on any changes with the latest or future versions but rather the state, in which Dynamo was used to develop the script. This chapter is taking a look at the foundations behind Dynamo and its possibilities and current limitations. For simplicity, when referring to “visual programming” it is meant visual programming in the environment of Dynamo. There are other software providing similar visual interface of textual coding that do not necessarily share all the points discussed in this chapter.

05.1. Visual Programming

Dynamo is a visual programming extension for Revit and Maya, currently developed by Autodesk. The official learning centre for Dynamo (Dynamo Primer, 2015) gives a good overview of what visual programming is. Conventional computer programming and visual programming essentially have similar goal – to allow the user to define a set of algorithms via means of either text, formatted in a certain syntax, or in a visual, and for some more intuitive, way by connecting pre-packaged “nodes”, which represent various commands. Nodes are boxes with inputs and outputs, which provided the right arguments, act as commands in a code. As an example, Figure 05-1 and Figure 05-2 show the algorithm "draw a circle through a point", defined in textual and visual programming.

Visual programming:

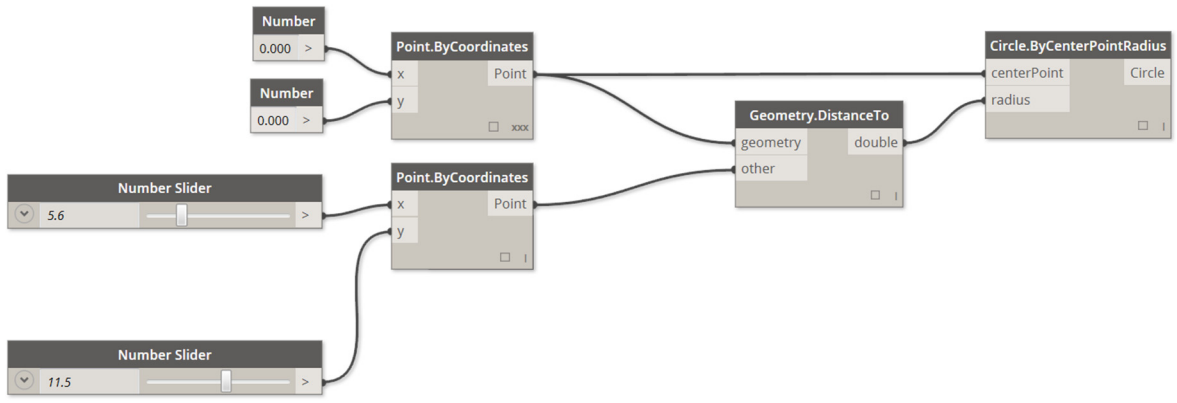


Figure 05-1 Example of Visual Programming (Dynamo Primer, 2015)

Textual programming:

```
myPoint = Point.ByCoordinates(0.0,0.0,0.0);
x = 5.6;
y = 11.5;
attractorPoint = Point.ByCoordinates(x,y,0.0);
dist = myPoint.DistanceTo(attractorPoint);
myCircle = Circle.ByCenterPointRadius(myPoint,dist);
```

Figure 05-2 Example of Textual Programming (Dynamo Primer, 2015)

Both examples do the exact same thing and have the same input and output. Generally visual programming is best suited for people with no or limited textual programming knowledge. For example architects who have never worked with coding, can use visual programming to achieve similar results, as with coding. There are a few major differences between the two methods of programming:

Possibilities. Textual programming has been around for as long as software existed. Visual programming (at least in the scene of substituting textual programming in construction software) is much newer. As such, it is not as developed and can never come in front of its "parent". In situations where it is not sufficient, blocks of textual code can be inserted within the visual environment for the specific needs. Given this fact visual programming is not as rich and enabling as textual, but does allow developing algorithms without coding skills.

Flexibility. After executing a written code, there is no way to go back and see exactly what happened, where. If the code is working as expected, that is not a problem, but a major time of any programmer is spent in finding errors in code. A solution is to write additional lines of "debugging" code, to help track the changes in the information flowing through the code. Working with visual programming allows things to be simpler in that sense, as after executing a "visual script", one can go back to every command and see its results at the time of calculation. This makes debugging much simpler than going through many lines of code without such aid.

Real time execution. Understandably, a code cannot be executed at the time of its creation. Visual programming removes this limitation, as placing a node, as a single action, acts as a complete and finished block of code. After connecting its inputs appropriately, the code can be executed automatically and changing any of the inputs is reflected on the result in real life. Taking the example above – if executed automatically, moving the sliders changes the radius of the produced circle:

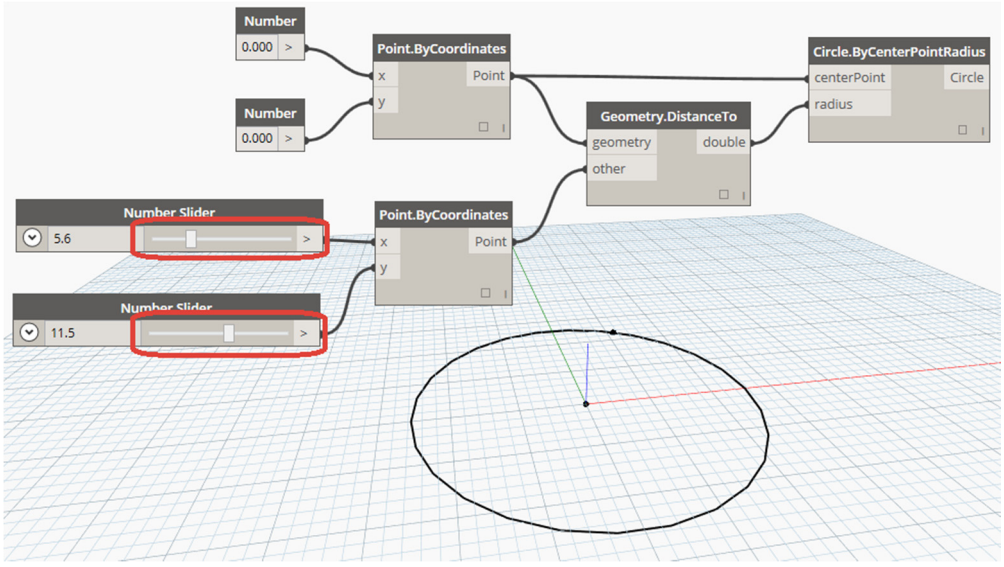


Figure 05-3 Real Life Execution of Code

Visual programming is in the core of Dynamo and understanding it is essential to understanding Dynamo.

05.2. Software Possibilities

Being able to manipulate Revit data in such a powerful and easily accessible way brings many possibilities. Understanding these allows for a better grasp of the possibilities of Dynamo. In a presentation for Autodesk University (Keough, 2014), Ian Keough, the father of Dynamo, presents several potential uses for Dynamo. Here are some of them:

Software Communication: Traditionally creating a complex geometry is often done in more competent software for the task, and then imported in Revit. Dynamo allows for creation of very complex geometry directly in Revit without the need of external tools.

Parametric Dependencies: Every element in Revit has parameters. In some instances Revit allows for some dependencies between parameters and simple formulas. Dynamo allows for *anything* with parameters to drive anything else, it allows elements to "talk" to each other. Furthermore it can take parameters from other files, like Excel, the Internet, etc, and drive data in and out of Revit.

Design and Analysis at the same time: With Dynamo analysis can be done directly in Revit, at the same time, using the same model. Having this analysis being performed at the same time as the design is taking place, removes the need of back and forth communication with analysers and thus the chance of miscommunication and mistakes. Also allows the designer to produce a higher quality design, being able to see the direct effect of his decisions immediately. Ian Keough shows an example of visibility analysis that writes a parameter in the model file and allows adaptive components to be sized according to that parameter.

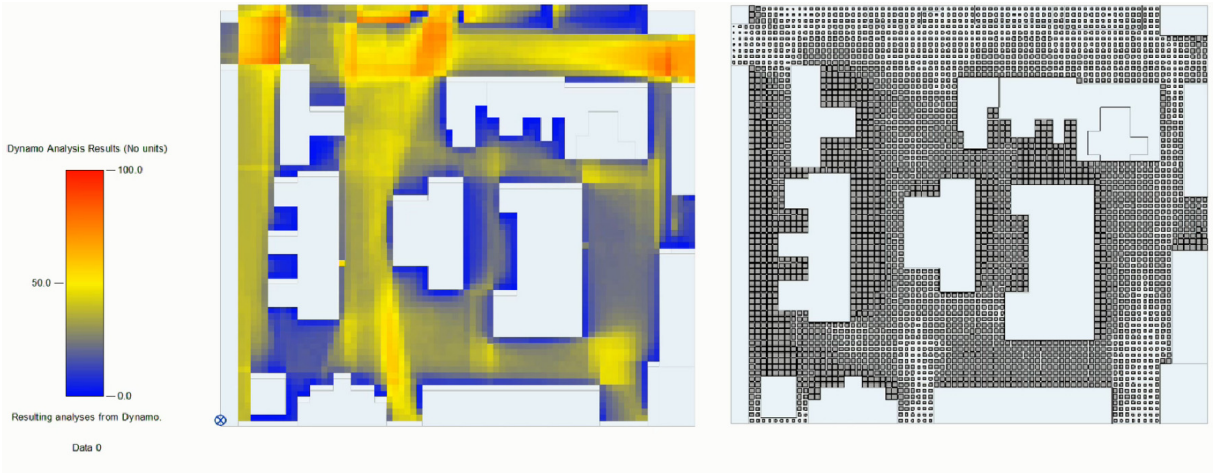


Figure 05-4 Visibility Analysis Done with Dynamo (Keough, 2014)

Figure 05-4 Visibility Analysis Done with Dynamo represents that analysis. To the left can be seen the data from the Dynamo analysis, represented as colours-coded plan view. The view on the right indicates the dynamic sizing of Revit elements (adaptive components) in the model..

In a presentation for Autodesk University (Hudson, 2014) called *Dynamo Hero: Using Revit Scripting Tools to Optimize Real-World Projects*, Michael Hudson (architect) and Andrea Vannini (expert in Adaptive Architecture and Computation) talk about two real project examples of the power of Dynamo, allowing design and analysis to happen at the same time. The first project is development of a bandstand for the seaside town of Littlehampton, UK, which does not require sound amplification. Regardless of the tight budget, Dynamo has made the project possible by allowing the acoustic analysis and the geometry creation to happen at the same time, and thus achieving the best possible result.

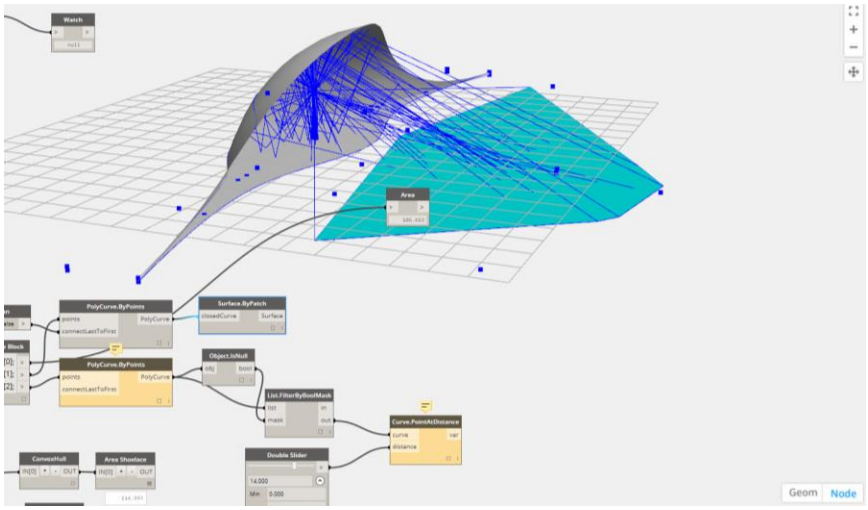


Figure 05-5 Acoustic Analysis with Dynamo (Hudson, 2014)

Figure 05-5 Acoustic Analysis with Dynamo above shows that analysis. The light blue area represents the area that would receive, which would receive a good sound without too large interval between direct and bounced sound (echo).

The second example project consists of two buildings, with the goals of maximising floor area while mitigating a windy microclimate (Figure 05-6 Prefabrication Optimisation with Dynamo below).

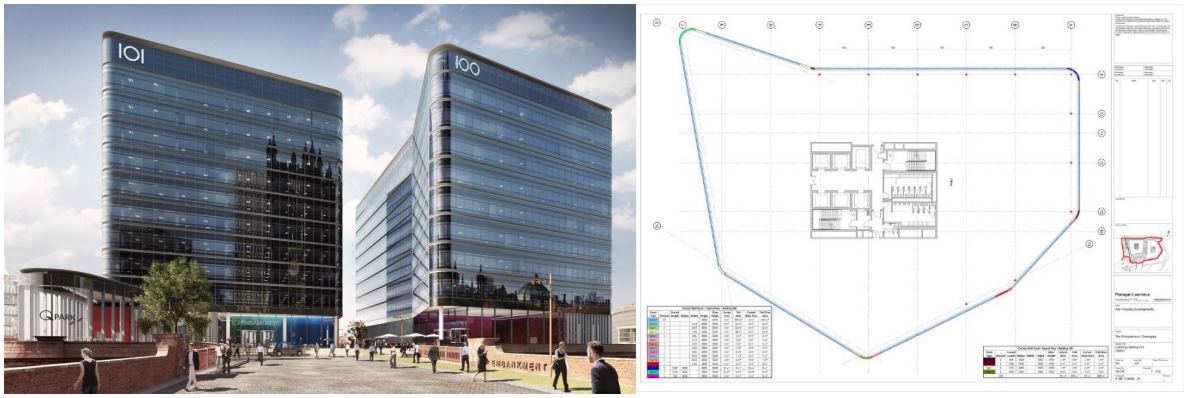


Figure 05-6 Prefabrication Optimisation with Dynamo

While building 101, on the left, could maintain typical 4-side façade, building 100 required more complex 7-side façade (refer to the floor plan on the right of Figure 05-6). Initial analysis of the façade of building 100 revealed that it would require 13 unique elements per floor. Development of optimisation script in Dynamo allowed for dynamic analysis while working on the building design in the Revit model. The results were improving the cost-effectiveness of the façade significantly, due to the decrease of unique elements per floor from 13 to 3 – refer to Figure 05-7 Facade Quantity Take-off below.

Curtain Wall Panel - Typical Floor - Building 100

Panel Type	Amount	Curved Length	Radius	Width	Height	Glass Height	Curved Area	Flat Area	Curved Glass Area	Flat Glass Area
Type A	101				3890	3590	0 m²	589 m²	0 m²	544 m²
Type B	2			1514	3890	3590	0 m²	12 m²	0 m²	11 m²
Type C	2			1691	3890	3590	0 m²	13 m²	0 m²	12 m²
Type D	2			1794	3890	3590	0 m²	14 m²	0 m²	13 m²
Type E	1			696	3890	3590	0 m²	3 m²	0 m²	2 m²
Type F	1			1212	3890	3590	0 m²	5 m²	0 m²	4 m²
Type G	1			1256	3890	3590	0 m²	5 m²	0 m²	5 m²
Type H	1			1262	3890	3590	0 m²	5 m²	0 m²	5 m²
Type I	1			1550	3890	3590	0 m²	6 m²	0 m²	6 m²
Type L	1			1916	3890	3590	0 m²	7 m²	0 m²	7 m²
Type M	1			2038	3890	3590	0 m²	8 m²	0 m²	7 m²

Curtain Wall Panel - Typical Floor - Building 100 - Option 2f

Panel Type	Amount	Curved Length	Radius	Width	Height	Glass Height	Curved Area	Flat Area	Curved Glass Area	Flat Glass Area
Type A	117				3890	3590	0 m²	683 m²	0 m²	630 m²
Type B	1			1626	3890	3590	0 m²	6 m²	0 m²	6 m²
Type C	1			1894	3890	3590	0 m²	7 m²	0 m²	7 m²
Type D	1			2301	3890	3590	0 m²	9 m²	0 m²	8 m²

Figure 05-7 Facade Quantity Take-off

The value of performing analysis while designing can be easily understood from these examples.

Flexibility from Software Constrains: Like all software, Revit has many constrains on the elements, such as floors are supposed to be flat, beams – horizontal, columns – vertical, etc. Dynamo has none of these constrains and can create geometry directly in Revit. With that freedom, parametrically driven roof construction like the one shown on Figure 05-8 Parametric Roof Construction below can be created directly in the model.

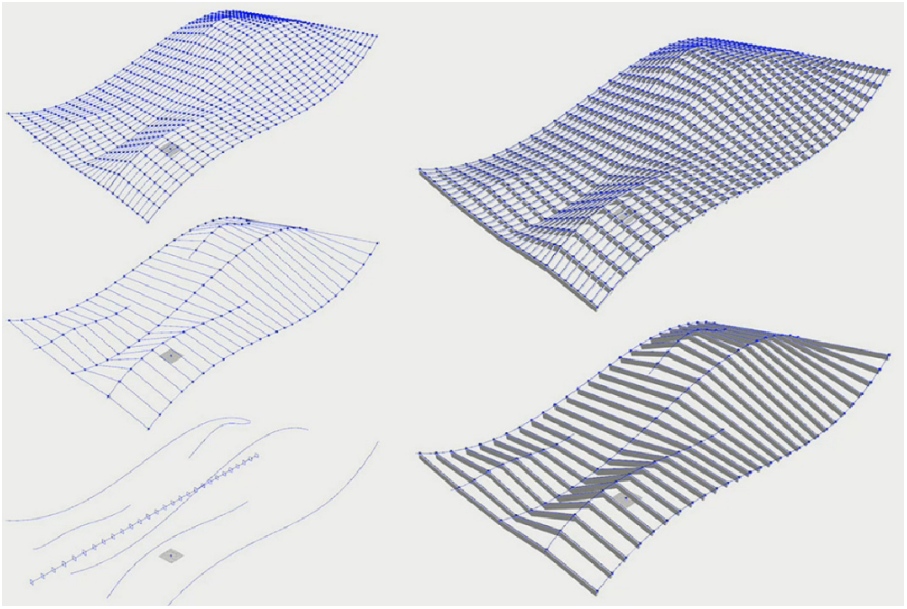


Figure 05-8 Parametric Roof Construction (Keough, 2014)

As shown in Figure 05-8, curved beam elements are supporting a roof structure and the rafters are created based on the variation of supports elevations.

All these possibilities paint a good picture for the software's abilities, but there are limitations that become visible only after one starts using it on daily basis.

05.3. Software Limitations

At the current state of development (version 0.9.1, as used for the development of the script, described in this thesis) there are limitations of Dynamo that affect the developed script. Some of them can be lifted with future versions but the current limitations are as described below. Most of them may originate from the assumption that Dynamo is not made to handle scripts of the scale of this project, but rather smaller and simpler ones. For such scripts, most of the listed points are not of any significant concern.

Hardware Requirements and Memory Management

The hardware requirements of Dynamo are seemingly quite substantial. Author's testing shows that even a relatively small Revit model requires significant amount of processing power and available RAM. On the test computer, the final script, used on the Revit test model (made specifically to include potentially troublesome to calculate scenarios) uses the following amount of memory, reported by Windows Task Manager: fresh start of Revit and opening test file: 400MB; opening Dynamo script: 1.5GB, running dynamo script: maximum of 2.3GB for the execution duration of 1:20 min. That is by no means an objective study but is meant to indicate that the script is best ran on a computer with better hardware, if different options are available. Moreover, memory leaks have been observed, increasing with the time of use of the program and keeping RAM used until Revit is closed and reopened.

Array Handling and Crashing

Related to the point above, it has been observed that combining larger arrays of data into fewer, bigger arrays, causes the programme to use significantly more memory and processing time, and at times crash during execution. It is thus proved necessary to monitor memory consumption of the script during development and avoid using too big arrays. That has been the reason for the decision to write the data to the output file several times instead of all at once – along with other considerations of handling of lists of arrays.

Execution Time

Managing a script of this size requires significant amount of time to execute compared to smaller scripts often found on the internet. Execution of the tested files is usually more than 1 minute. For one time execution that is not much but for continuous development, when the script needs to be executed very often for testing, it builds up. Dynamo is optimising this time to an extent, recalculating only parts of the script that have been changed, but as the file has been opened for longer time and ran multiple times, this time gradually increases, perhaps due to the poor memory management being observed and described in the previous point. In the later versions of the software a feature was implemented, allowing to freeze a point in the script and everything following would not be executed.

Looping and Logical Formulas

Familiar to anyone, who has worked with code or some form of logical programming, looping is an extremely helpful asset when creating algorithms such as the ones in this project. The algorithms used in the script are described in chapter 08. Dynamo Script Design. Dynamo does have a node called “Loop While”, which is a good solution in principal, however in practice it is complex to use, offers very limited functionality, and in the case of the script getting stuck in the loop, there is no way to stop it without killing the whole programme and losing the work since the last save point. Considering that saving takes significant amount of time, it is quite impractical to use “Loop While” in the few situations where applicable. Instead the script developer needs to find a way to manage the lists of data by performing actions to the whole lists, without the ability to go through the items one by one and testing a condition. Usually workarounds are needed to achieve the desired result. An option is to write the loop code in Python,

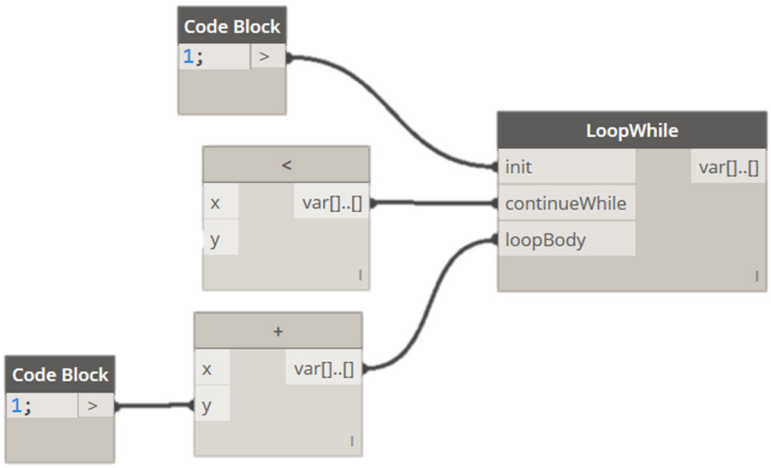


Figure 05-9 Loop While Node

using standard coding syntax, but that is not possible for people without such skills and is beyond the project constrains set out in chapter 03. Problem Formulation.

The workaround used in many places throughout the script is using a combination between the more complicated list structure in Dynamo and the so called “lacing”, together with creative “cycling” of some of the lists. “Lacing” and “cycling” are described below:

The term “lacing” in Dynamo defines how two or more inputs (lists) of different amount of items are processed by a given command. There are three lacing options:

1. Shortest: Each of the items of the shortest list is matched with each of the first X items of the longer list(s), where X is the amount of items in the shorter list.
2. Longest: Same as Shortest lacing, however the last of the items in the shortest list is *also* matched with each of the remaining items of the longer lists.
3. Cross Product: each of the items of each list is matched with each of the items in all other lists. This creates all possible matches, structured as “list of lists”.

The above-referred “cycling” of lists refers to the node List.Cycle in Dynamo. It allows all items of a list to be repeated X amount of times, where, for the purpose of the workaround, usually X is the amount of items in another list.

The lack of proper looping restricts the possibilities of the Dynamo script without the use of Python and for that reason the output file cannot meet one of the desired criteria, outlined in chapter 07. Output Requirements. It has been agreed with the collaborating group that the problem will be solved in the following VBA script, after the Dynamo export.

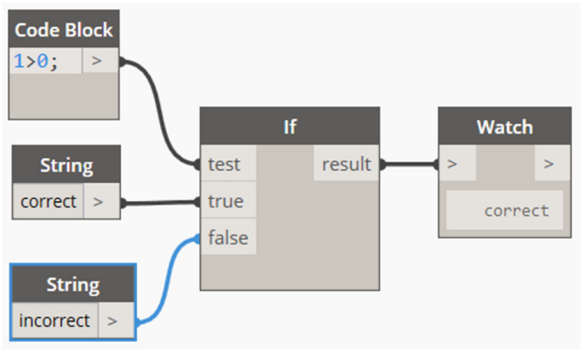


Figure 05-10 IF Node

Another limitation of Dynamo is the way it handles logical formulas, such as standard “IF”. Typically in programming, a condition would be checked, and depending on whether it is true or false, one of two things is being done. However in Dynamo, both “true” and “false” are being calculated and the condition decides which of the two results to use. That requires additional processing time that is not used for anything useful.

Besides the more major limitations to the development, there are more minor limitations of the software that make the development process more difficult – such as the lack of more sophisticated grouping of nodes and ability to have nested groups in one another. Although these would make the clarity of the canvas and navigation significantly easier, they are not of such importance to the outcome of the project and are therefore not being discussed in detail.

06. Development Process and Decisions

The project has been developed in constant communication with the colleagues from the Indoor Environmental and Energy Engineering group, especially in the earlier stages of development. There have been numerous discussions and decisions taken with the group and some more technical decisions taken by the author while creating the script. Some of these are being described in this chapter with the goal of giving the reader a better inside to the development process and general approach toward the problem. Additionally, as part of the development process, learning of the software has been described, along with testing of the script and final refinements.

06.1. At Which Point to Introduce New Information?

Creating the automated process has the benefits of allowing the creators to choose at which point in the flow new information is being introduced. Information that needs to be added is information regarding which rooms are being analysed in detail in the simulation, which thermal zone the rooms belong to, HVAC systems, people load, equipment load construction specifications. There are two points in the flow, at which new information can be introduced: in the Revit Model, by editing the model in Revit; or in the XLSX file. Moreover, there are several approaches – either all information could be added in one of the places, or the information can be split, and some of it can be added in the Revit file, and the rest – in the Excel file. Given the fact that the Dynamo script has some limitations and requires some editing of the Revit file in preparation of the execution of the script, avoiding adding the information in Revit provides no benefits. Having information on what is needed for the simulation can greatly help the resource requirements of the script, thus it is very logical that information regarding which room(s) are going to be analysed is added in the Revit model. On the other hand adding too much information in the model (that does not help the script in being less resource demanding) can slow the script down. So it is logical that the rest of the information would be added after the export from the script. Considering all these factors, the creators have decided to add room information, regarding whether a room should be analysed, and the thermal zone it belongs to, to the Revit file, and all additional required information, in the excel file – such as HVAC systems, people load, equipment load, and also the properties of the constructions. It is decided that the script exports the names of the elements, which are unique for the type of construction, and enough to look up the needed information from the Revit model. After the export from the script, these unique names are linked to BSim construction database, for the simulation to use.

06.2. Changes Throughout Development

The question of how big part of the flow the script can handle has been continuous throughout the earlier stages of the development with the answer varying as the script got more developed. Accepting a conservative stance on what can be done with Dynamo from the beginning proved to be a good approach, considering how new the software is and the lack of experience with it. Initially not

overly complicated task was expected to be not as easy as thought, and correctly so. While initially there were ideas of having the script to also provide suggestions on changes to the construction – for example location, size, shape, and amount of the windows – it was later decided that it is of higher importance to have a working flow from start to finish with a simple structure, to avoid potential problems. And after proven working well to develop further features such as changes suggestions. Another discussion was if the script would be able to feed back the result of the simulation to the model. While Dynamo is indeed capable of doing that, it makes more sense to have that task handled by a separate script, to make both lighter and less likely to crash. Thus it has been decided to be left out of the current project until future development is taken over. Further discussion of the idea of feeding back the simulation results to the Revit Model can be found in chapter 10. Future Development.

06.3. Efficiency Optimisation

There has been discussion on the topic of whether optimisations for the script should be made in the Revit model – in terms of the information being processed by the script. While the execution time of the script may be improved, if the model is cleaned up of information that is not relevant for the analysis, the script has been designed in a way to filter out information that is not relevant and not process data that is not needed, for the most part. There are however limitations to this, and that is the reason it may be improved by cleaning the model. However the decision was to have the script work on mostly unmodified Revit file, with a few exceptions, that are described in the User Guide.

06.4. What Data to Export?

While there is a lot of information that can be exported, an analysis of what information is needed and exporting *only* that information is critical to the speed and resource consumption of the script and to avoid crowding the output file with data that is not used later. While initially things seem clear, there are more grey areas, which are a matter of approach. One such is in areas with suspended ceiling. While the most accurate representation of the construction would be to have the floor slab and ceiling defined as two separate entities in the geometry and the air between them as separate “cell” (more details on the terminology in chapter 07. Output Requirements), that would make both the script and simulation more time consuming. Therefore a decision on which exact building elements to include in the export has been taken, and they are: walls, floors, ceilings, and roofs. The materials and specific composition of each element however is not pulled by Dynamo, but instead, the unique element identification is exported by means of its Family Name and Family Type. After the export, the different element constructions are included in a database, which links the family name and type to the specific detailed construction for BSim to use.

06.5. Exported Storey Partition Elevation?

As described in more detail in the next chapter (chapter 07. Output Requirements), whenever a storey partition is inside the building, it needs to be exported as a surface, located in the middle of the construction. For example if a storey partition has a total thickness of 400mm, the surface in the exported geometry would lie in the centre of the element (200mm from the top and the bottom). However, as described in detail in chapter 08. Dynamo Script Design, the script can handle only one elevation for the whole floor. If there are rooms/spaces on the floor with different ceiling height, they would ideally have different elevations of the storey partition geometry in the XML file. Seeing that the script can only export one elevation, a decision needs to be made on whether to take the highest, lowest, on anywhere in the middle.

A typical scenario in an office building is a case where a corridor has a lower ceiling and the offices have higher. Choosing to use the higher level would leave the air volume in the corridor bigger than the actual, while choosing the lower level, would leave the volume of the offices lower than desired. Taking into account the rest of the problems and questions discussed, it was decided that the best approach is to compare the heights of the ceilings of only analysed rooms for the given floor. Which rooms are being analysed is then defined in the Revit Model by adding a parameter to the Room Element, as described in the User Guide. That significantly lowers the negative effect of the limitation of the script, leaving the problem in consideration only in cases of two rooms with different ceiling levels being analysed on the same floor. In such case, a choice has been made to take the lower ceiling level between the two, as being more conservative.

06.6. Windows and Doors Dimensions

The script needs a way to find the dimensions of each window and door. In Revit, typically windows and doors have parameters for width and height. The cleanest approach would be for the script to find the location of each window and door and read their width and height from these parameters. However that requires that these parameters are both named exactly “Width” and “Height” (or whichever name is chosen in the script), and that they point to the same dimension on the window and door. It is not rare to see elements with several dimensions defining the width and height, for example “width” and “rough width”. That raises a problem of consistency and taking that approach in the script would require that all windows and doors in the model have the exact same names for both parameters and both parameters point to the exact same location within the family. It is very likely that that is not the case and going through the families in a big project to fix these parameters is a very tedious task, so a different approach is considered.

The alternative to that method is to consider the actual geometry of the window/door and choosing width and height of the exported element based on the location of their outermost points. That is a little more resource heavy task but deals with the above-described issues. The downsides of this

method is that not all families are created equally well and “clean” and sometimes it may happen that some geometry of the element is being created outside of the desired borders of the window/door. That may happen if a 2D representation of the window/door frame, meant to be shown in plan view, is made as 3D geometry, instead of 2D. That would create the “outermost points” further out than the actual dimensions of the element.

There are arguments for both methods, however the second method has been chosen, which takes the outermost 3D points of the element. The reason is that it is considered less likely that a family would be made in a way that would cause the script to calculate wrong values, than the parameters being named differently, or not pointing to the desired references in the family. A suggested fix to the cases when a window or a door *is* created wrong, and the calculated dimensions are bigger than the element is, after the script is executed, to visually examine the 3D model in Revit and see whether the calculated size of windows and doors matches the lines of the elements. The script is created in a way that visualises 3D lines in the Revit model so the calculations of the script can be compared to the structure of the model and any mistakes – visible. If any elements are detected as being wrongly calculated, they need to be edited in Revit and the geometry that is causing the issue to be deleted. Detailed steps of this process are described in the User Guide.

06.7. Resource Usage vs. Accuracy

Generally speaking, a lot of the current limitations of the script could be removed by using more elaborate algorithms. That would increase the accuracy and amount and quality of features of the script but would cause the script to use a lot more resources at the same time. In ideal situation, on a good hardware, increase in resource usage would only cost execution time, which compared to the time required by the BSim simulation is not very much. But in sub-optimal conditions, when the hardware may not be up to par with the task, and especially considering the lack of optimisation of Dynamo for bigger scripts like this yet, the script may simply crash and not be able to finish executing. Observation during the development show that the hardware requirements of the script surpasses the Revit requirements for a given Revit model. Considering that Revit is a very hardware-demanding software, complexity of the Dynamo script algorithms need to be taken seriously. That is the reason, at least with the current version of Dynamo, to make compromises between accuracy and resource demands. Chapter 08. Dynamo Script Design describes in more detail the specific compromises that have been made in the script. A few more major ones are:

Matching construction type

The algorithm used measures distance from the centre of the face of the exported geometry to the closest element and taking its construction type. There may be rare cases when the geometry is at equal distance between the actual element it represents and a different one. A more sophisticated

algorithm would be one that calculates the intersection of the geometry with each family and decides which is the best mach.

Storey partition elevation

As described above, the script only allows for one elevation of the storey partition for a whole floor. The reason is to not make it more complex than already needed, as incorporating multiple elevations (and connections between the different elevations) would require a lot of new calculations.

External walls need to be connected in one chain

While that restriction covers most of the buildings, it also allows the script to be very simple in the initial steps when extracting and processing the external walls from the model.

External walls need to be the same width

The script extracts the centre-lines of the external walls and in case of having different widths, it would need to do separate evaluation for each wall to decide how far to offset the line, to meet the face of the wall. Being the same width, it can evaluate only once and save a lot of time.

No sloped ceilings or sloped windows

Sloped ceilings or sloped windows would require separate algorithm entirely to manage the generated surfaces. Currently all surfaces created by the script are either horizontal or vertical.

06.8. Testing

Part of the development process is testing the prototype. As it is not possible to write a script like this without a Revit model to work over, an initial simple model has been created. As the script developed further, the same test model increased in complexity, with the goal to cause as much trouble for the script, as possible, and thus test the reliability of the script and reveal unexpected issues. The test model includes intentionally created complexity such as (among others):

- multiple storeys
- walls that face directions different than the common North, East, South, West, with openings
- multiple windows on the same wall, above one another
- different combinations of rooms being “analysed” and not
- different types of construction for storey partitions being used
- different combinations of presence of ceilings in the rooms and lack of such
- different techniques of creating window families

A 3D view of the test model, along with two room plans can be observed on Figure 06-1 Revit Test Model – 3D View and Room Plans below.

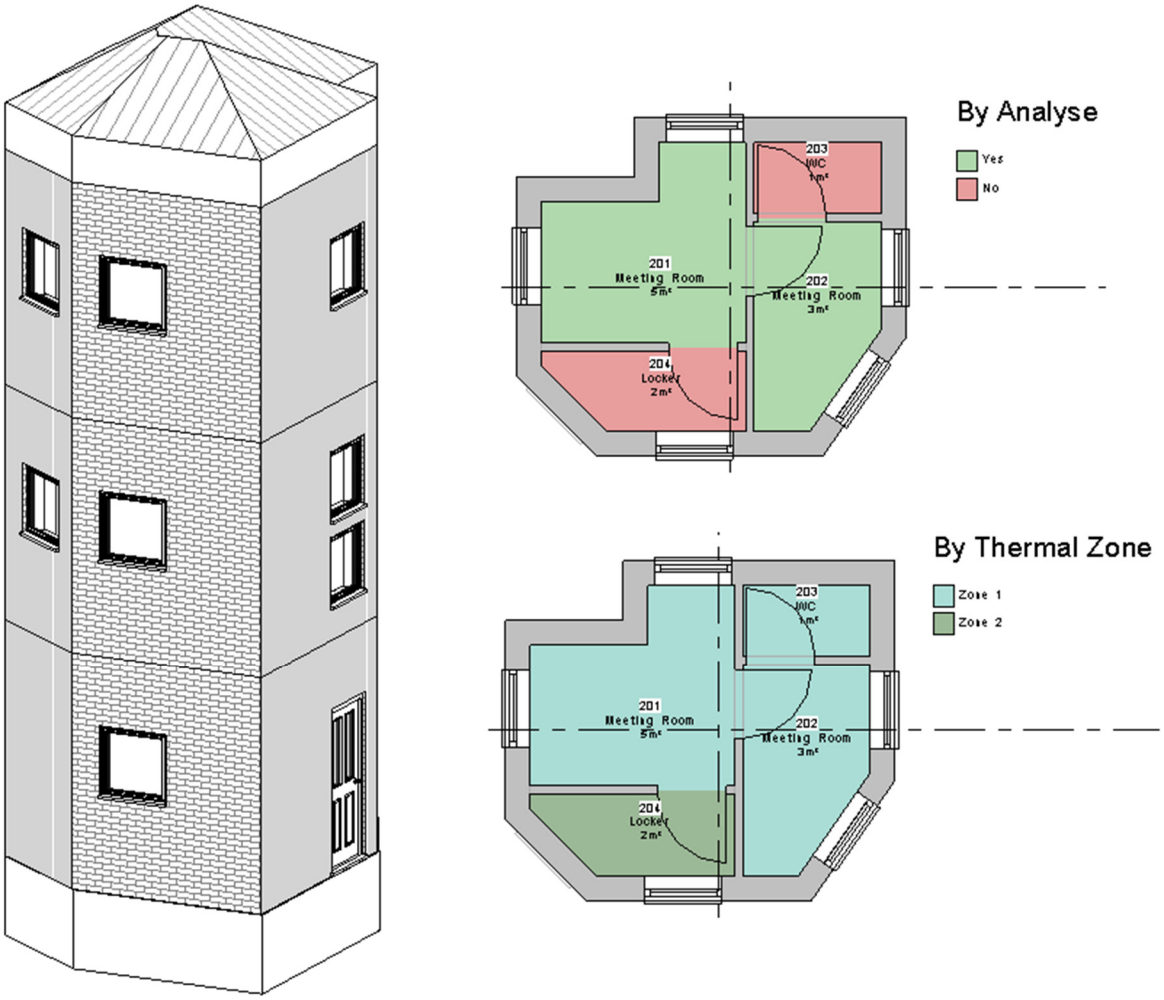


Figure 06-1 Revit Test Model – 3D View and Room Plans

After the prototype of the script has been thoroughly tested with the described test model, another test was performed with another, real project model. The new test presented several issues, which were not initially predicted. Some of the issues were with present algorithms, which were optimised to overcome the issues, while others required developing new algorithms to manage unforeseen situations. One example of a limitation to one of the algorithms, found during this test is the connection between internal and external wall in specific situation, described in subchapter 08.3. Script Constraints. Initially the issue with the algorithm required much bigger workaround, however, after optimisation without adding extra complexity, the workaround was reduced to an acceptable magnitude. As described in the above sub-chapter 06.7. Resource Usage vs. Accuracy, balance between the two is a goal, and this situation is one, where it was decided to optimise the current algorithm instead of developing a new, more complex one, that would be more accurate.

After finalising the refinements and final optimisations to the script, several tests were completed without any issues, and the version of the script was officially labelled as “stable” and final.

07. Output Requirements

As described in chapter 03. Problem Formulation, the goal of the Dynamo script is to output a file with very particular set of data from the Revit Model, structured in a very specific way. Due to the better possibilities of Dynamo to output data to Excel file (.xlsx) compared to XML, it has been agreed and chosen to export the data to XLSX.

The requirements for the output file have been discussed with the collaborating group from Indoor Environmental and Energy Engineering. The needs for the output file include specific information for the geometry of the building, and the type of the construction in the different exported elements. It has also been decided that the thermal zones shall be defined in Revit, instead of manually in excel later. Along with that information, general information about the extracted rooms is exported too – such as room name and room number. The output file requires the geometry of the building envelope and the analysed rooms, and does not need any other geometry that would make the file output more complex than needed. That has been used to lower the computational time of the script, by not analysing geometry that does not need to be exported. More information on that can be found in chapter 08. Dynamo Script Design.

BSim requires the input data to be structured in very specific manner. After discussion with the group, the required set of data for the automation of analysis has been chosen. That defines the structure of the output file as described in this chapter.

The output file contains 9 objects, exported in different sheet: CONSTRUCTION, ROOM, CELL, FACE_SIDE, FACE, WINDOOR, EDGE, VERTEX, VECTOR3D. Below is a short description of each of them, followed by a summary table, describing the whole structure of the output file:

VECTOR3D is a point, defined by X, Y, and Z coordinates. These are defining the location of the vertices (below).

VERTEX is an object that contains vector3d. Each vertex has exactly one vector3d. Vertices define all exported geometry.

EDGE is an object defined by two vertices.

WINDOOR is an object, defined by four edges. Windoor includes windows and doors, from where it get its name.

FACE is a surface object that is defined by 3 or more edges (typically 4). Walls and storey partitions are defined as a face. The specific location of the face object in relation to the building element depends on the type and location of the element:

- Walls: for external walls, a face is the external side of the wall. For internal walls, the face lies on the centreline of the wall.

- Storey partitions: for storey partitions, the face lies in the centreline of the cross section, except for the bottom and top floor/roof. The bottom floor has the face in the lowest surface of the floor (in touch with the soil) and the top has the face object on the top.

In other words the outermost faces of the building envelope are defining the external face of the construction element, and any internal faces, describe the centreline of the object – wall or storey partition element.

FACE_SIDE is one of the two sides of a face. Each face has two face_sides. A face can either face a cell (see below) or outside of the building. They also may contain windoor objects.

CELL is an object that is defined by several face_sides. A typical space with 4 walls, ceiling and floor would be a cell with 6 face_sides.

ROOM is a room object. It is being represented by a cell and has properties typical for a room, taken from Revit, such as name and number, along with a thermal zone, which it belongs to.

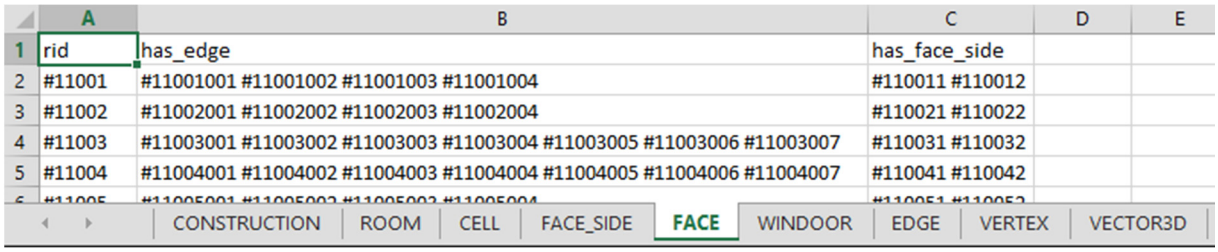
CONSTRUCTION is an object that describes an element. Each "construction" object is represented by a face. It has an ID, which is then linked to a specific construction type from BSim's database. In this case that ID is formed by the Revit model Family Name and Family Type – a combination, which is always unique in Revit, and also usually descriptive enough, to be understandable by the user.

Each of these objects has a unique reference number (RID), which is used when referring to the specific object. The RID is a # sign, followed by a string of numbers. Below is a table, describing in more detail the structure of the file:

object name	contains	notes
CONSTRUCTION	rid (unique reference id) id (describing the construction) represented_by (face rid) includes_segments (windoor rid)	If more than one windoors in includes_segments separate them with a space.
ROOM	rid id (name of room) represented_by_cell (cell rid) thermal_zone_id (defined as room parameter in Revit) revit_room_number (room number from Revit)	
CELL	rid bounded_by (face_side rid)	bounded_by is series of face_sides with spaces as separators
FACE_SIDE	rid faces_cell (cell rid) has_face (face rid)	faces_cell is $ if the face_side is external
FACE	rid has_edge (edge rid) has_face_side (face_side rid)	

WINDOOR	rid represented_by (face rid)	
EDGE	rid has_vertex (vertex rid)	Vertices are separated by spaces
VERTEX	rid has_geometry (vector3d rid)	
VECTOR3D	rid x y z	

Below is an example of the output file from the script:



	A	B	C	D	E
1	rid	has_edge	has_face_side		
2	#11001	#11001001 #11001002 #11001003 #11001004	#110011 #110012		
3	#11002	#11002001 #11002002 #11002003 #11002004	#110021 #110022		
4	#11003	#11003001 #11003002 #11003003 #11003004 #11003005 #11003006 #11003007	#110031 #110032		
5	#11004	#11004001 #11004002 #11004003 #11004004 #11004005 #11004006 #11004007	#110041 #110042		

CONSTRUCTION | ROOM | CELL | FACE_SIDE | FACE | WINDOOR | EDGE | VERTEX | VECTOR3D

Figure 07-1 Output File

As can be observed from the description above, in some cases different cells would be in contact with the same face – for example, when two rooms are divided by the same wall. Due to limitations of the algorithm, described in chapter 08. Dynamo Script Design, the output of the script actually exports different face for the different cells, which in turns creates more than one face on the same physical space. As described in sub-chapter 05.3. Software Limitations, after agreement with the Indoor Environmental and Energy Engineering group, it has been decided clean-up of the duplicate faces and correction of their references to be done in the VBA script rather than in the Dynamo script.

08. Dynamo Script Design

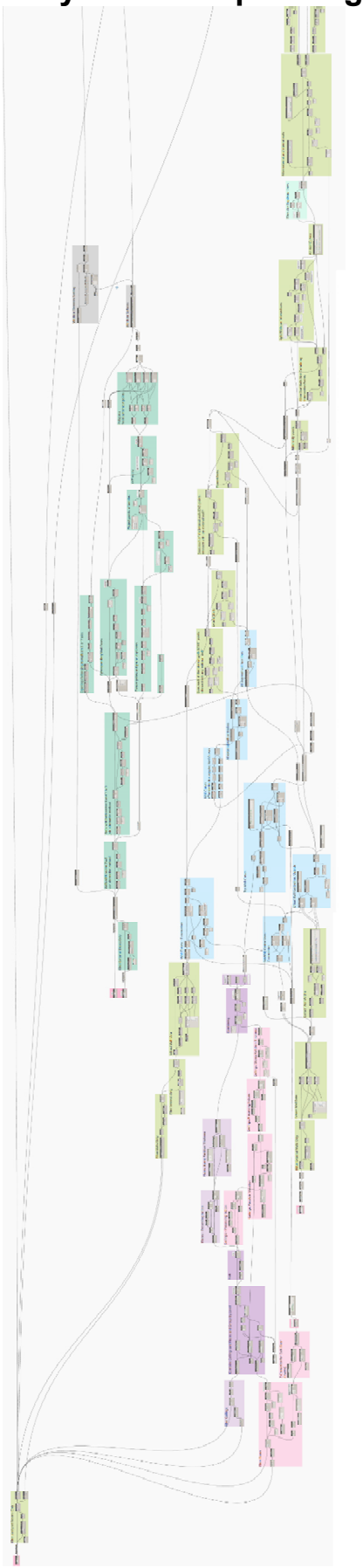


Figure 08-1 Dynamo Canvas – Left Half of the Script

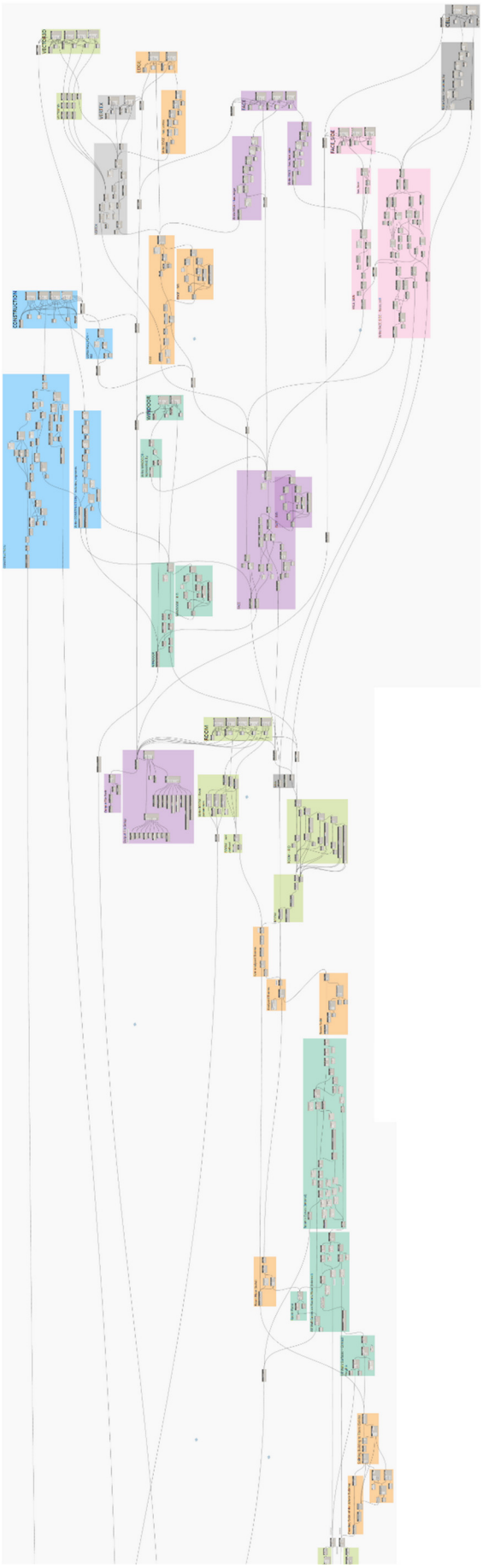


Figure 08-2 Dynamo Canvas – Right Half of the Script

This chapter describes the design of the script and in particular the details of how it works and what are its limitations. In the beginning there is an overall flow of the logic of the script, followed by more detailed description of the different parts. The latter part contains more technical language and is mainly aimed at people who are interested in taking over future development of the script or potentially fixing and/or optimising the script for a particular project.

08.1. General Structure

Figure 08-1 and Figure 08-2 Dynamo Script Canvas on the previous page represent the canvas of the script in Dynamo. Higher resolution can be found in Appendix A. The nodes are organised by groups. The colours represent common purpose of the different groups. The work that the script performs can be summarised as shown below:

1. Elements of analysed rooms – The scripts extracts the elements of the analysed rooms and processes them separately
2. Internal walls – The centre-lines of the internal walls are extracted, along with their levels.
3. Storey partitions – All floors and ceilings are extracted and the total thickness of each storey partition is defined
4. Z-correction – Calculates the elevation change (z-direction), by which to move the wall-lines, according to the thickness of the storey partitions. This way they align to the bottom, middle, or top of the storey partitions, according to the output requirements, stated in chapter 07. Output Requirements.
5. External Walls – Extracts the outer base lines of all external walls and aligns them to the proper elevations.
6. Windows and Doors – Extracts all windows and doors, finds which wall they are hosted on and creates their geometry on that wall face. The script uses two different methods of calculation for the windows and doors, depending on which direction they are facing. It uses the simpler (and quicker) algorithm for elements facing the project north, west, south or east, and a more complex algorithm for the rest.
7. Extending Internal Walls to External – Internal wall lines are initially exported to meet the centre-lines of each other wall. That includes external walls. Therefore each internal wall-line needs to be extended to the external face of the external walls.
8. Trimming Wall Lines – All the walls produce lines from the start of the wall to the end of the wall. These lines need to be split on every intersection in such way, that no line is being intersected by another line in any point of its length except the start or end point.
9. Creating Building Solid – Creates one solid form for the whole building envelope.
10. Splitting Solid to Floors – Splits the building solid to floors
11. Creating Room Solids – Creates solid forms for each of the *analysed* rooms.

12. Subtracting Rooms from Floors – Removing the room solids from the respective floor solid. This leaves the rest of the space of the floor as one unit that is later defined as a Cell and Room for the output file.
13. Writing to Output File – Processes all the needed information and writes to the output file, which is located in the same directory as the Revit file and named the same way including " - output" string. The writing to the file happens several times throughout the execution of the script. Each writing opens the file in Excel after finishing. This may be confusing, letting the user think the script has finished, while it may not have. A good practice is to minimise the Excel window and to wait for the confirmation "Run Completed" in the Dynamo window.

This simplified representation of the actions of the script include 13 steps. It is worth noting that they do not execute in the exact order shown above and many of them run partially simultaneously, but this summary serves as a good representation of the general logic and structure of the script.

08.2. Building Blocks

This sub-chapter goes in further detail in the major parts of the script. It is meant to be read by anyone interested in the little details of the script, and anyone interested in continuing development of the script. Anyone who is already working with Dynamo may also find these details interesting and perhaps educating. The script, which can be found in Appendix A, is schematised in Figure 08-3 and Figure 08-4 on page 33 below, which can also be found in higher resolution in Appendix B. The scheme is made in a way that keeps the same relative position of the script elements, and preserves visually similar colours. The goal is to serve as a simplified guide to understanding the way the script works in better detail.

Each of the coloured boxes on the scheme represents a *group* of nodes from the script. These coloured boxes are referred to as *building blocks* in this paper. Different sets of building blocks are bounded in the scheme by a thick coloured line. This line indicates group of blocks with similar purpose, and combines them under a common name. These sets of boxes with common purpose are referred to as *groups of blocks* in this chapter. Each group is described in more detail below.

Windows and Doors

This group's purpose is importing and processing window and door elements. It consists of five building blocks, as seen the scheme in Figure 08-3 Scheme of Canvas (Left Side). The block called Import Windoor performs the actions of importing all windows and all doors, filtering only external doors, and then combining them in one list. It is important to note that that list is ordered in the way the elements are being imported by the Dynamo nodes and has not got a link to which floor, wall or room they belong to.

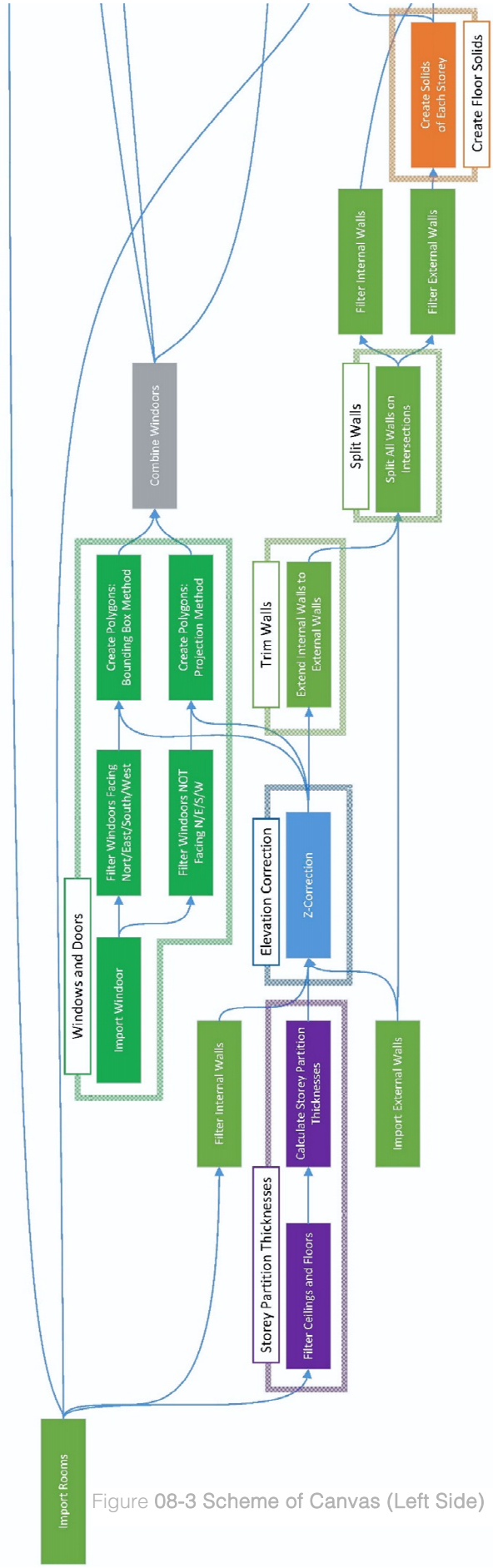


Figure 08-3 Scheme of Canvas (Left Side)

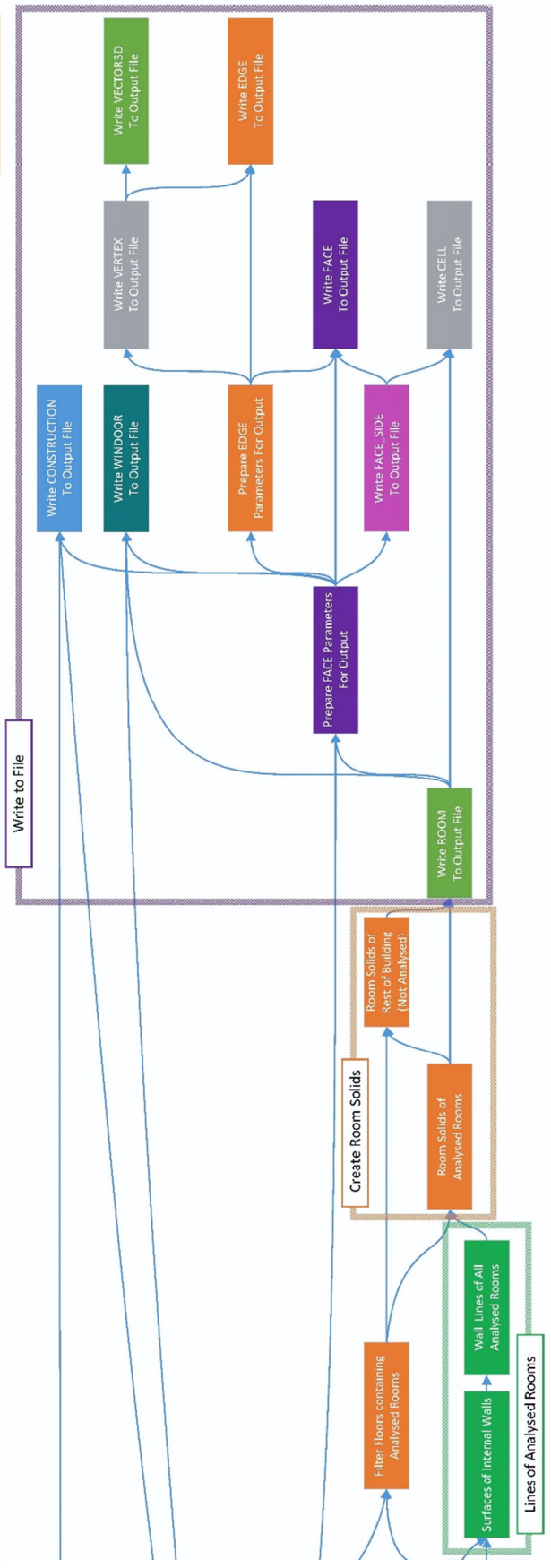


Figure 08-4 Scheme of Canvas (Right Side)

The next two blocks find the wall-host of each element and identify the vector of direction of each host – which subsequently indicates the facing direction of the windoor element. Afterwards split the windoor elements in two groups: one group that faces one of the project cardinal directions (N/E/S/W), and another, which face different directions. In the script, there is a set of nodes, which filter elements not facing horizontal direction – such as skylights, or windows on slanted walls or roofs. This is meant to be used in future development, if support for such windoor elements is implemented.

Each of the two groups of nodes are being processed using two methods – bounding box (being faster and lighter), and projection method (being heavier but supporting elements not facing the cardinal directions). Each of them is described below:

Method 1: Bounding Box

It creates a cuboid shape, aligned to the x and y axes, around each element. Then intersects that solid with the surface representations of the external walls, and the result is the desired surface of the windoor element. Since the bounding box is always aligned to the project x and y axes, it is not a suitable method for elements facing different direction, and that is the reason for using two separate methods.

Method 2: Projection

The method involves projecting all vector points of the elements, towards the external walls, in the direction, in which the elements face. This is done via the node Surface.ProjectSurfaceOnto. The node has three inputs: Surface (representing the walls), geometryToProject (vertex points of the elements), and projectionDirection (the vector of facing direction of each element).

The input geometryToProject if calculated by exploding the geometry of all of the elements in the group and extracting all vertex points. The input of Surface.ProjectSurfaceOnto is calculated by finding the closest wall-surface to the centre of each windoor element by using Geometry.DistanceTo node. That can cause incorrect results in rare occasions of very thick and short walls with windows, but is used as a compromise between accuracy and resource usage – a topic, discussed in sub-chapter 06.7. Resource Usage vs. Accuracy. Finally the input projectionDirection is calculated by finding the facing direction of the wall-host of each element (vector).

Normally, having three lists of equal lengths of inputs can be simply connected normally to the node Surface.ProjectSurfaceOnto. The three lists are surfaces of walls, geometry of elements, and vector directions. However, since each element has many vertex points, each of which has to be projected to the same surface in the same projection, that makes the second list much longer than the other two. While normally this issue could be solved with looping, for reasons discussed in sub-chapter 05.3. Software Limitations, this is not an option. Had there been only two lists, Dynamo does offer the node List.Map, which would work, but since the required inputs are three, that would not work. The chosen solution is to implement a workaround by duplicating the items in the two shorter lists as many times, as there are vertices in each element. That causes the three lists to be of equal length

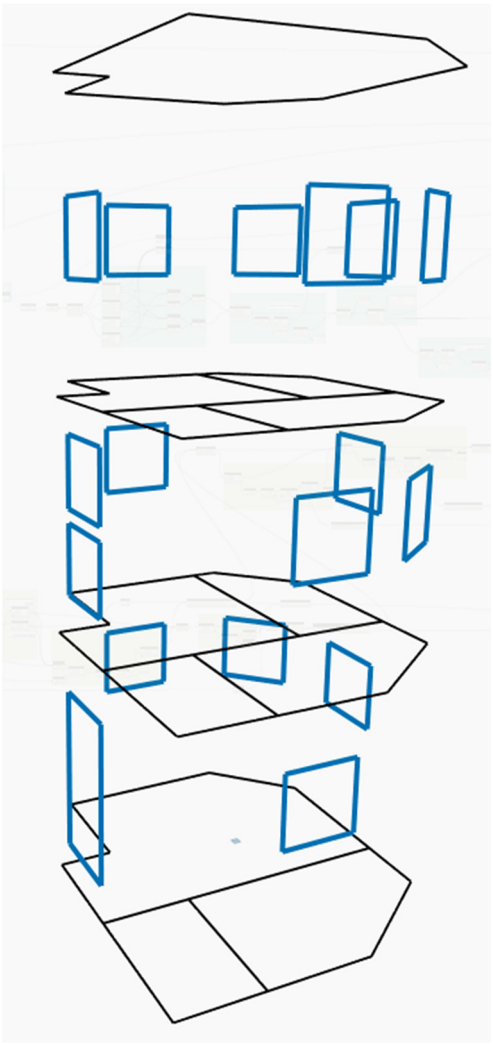


Figure 08-5 Windoor Geometry

and all items to be ordered correctly. Then the three lists can be processed using the node List.Combine with a combinatory of Surface.ProjectSurfaceOnto. This described issue in this situation is found in several places throughout the script and the chosen workaround of duplicating the items in the shorter lists, to match the length of the longest list, is found to be the most efficient solution.

Finally, having projected all vertices of each windoor element, the outermost points for each element need to be extracted. This is done by means of converting the 3D points to 2D points on the surface they lie on, and finding the outermost points using their 2D coordinates. Those four points are in the end connected to form the surface, representing the element. The result of this group of building blocks can be seen as blue lines on Figure 08-5 Windoor Geometry to the left. The process has been performed on the test file, shown in Figure 06-1 Revit Test Model – 3D View and Room Plans on page 26.

Storey Partition Thicknesses

This group's purpose is calculating the thickness of each storey partition. That thickness is used later in the script to correct the elevations of the surfaces of walls, so the values passed on from this group of blocks actually represents the corrections of the elevations for each floor. As described in chapter 07. Output Requirements, the geometry of the lowest storey partition needs to lie at its bottom, the top storey partition – at its top, and all storey partitions in-between – in the middle. That requires the values passed on by the group "Storey Partition Thicknesses" to be: full thickness for the lowest storey, and half thickness for all storeys above. These values are referred to as Z-correction throughout the script, as they correct the Z-coordinate (elevation) of the geometry. Refer to Figure 08-6 Storey Partition Geometry.

The group "Storey Partition Thicknesses" comprises of two building blocks: Filter Ceilings and Floors, and Calculate Storey Partition Thicknesses. While the first is fairly simple, the second block represents more than 10 groups of nodes in the script. All of these groups of nodes are coloured in purple or pink in the script, for visual clarification. The script can be found in Appendix A.

Filter Ceilings and Floors

As can be seen on Figure 08-3 Scheme of Canvas (Left Side) on page 33, the input (blue line) is the building block Import Rooms. It is intentional decision not to import all ceiling and floor elements, but only the ones around analysed rooms. It could be the case that there may be several storeys of a building without analysed rooms, and thus no storey partitions will be generated by the script. That is also intentional, and the resulting space is combined into one CELL in the output file. This saves calculation time and lowers the complexity of the output file. The group of blocks "Filter Ceilings and Floors" takes all elements binding the analysed rooms and extracts only the ceiling and floor elements. From the floor elements, the floor with lowest elevation is taken out, and the Z-correction for the lowest storey is created from the floor's thickness.

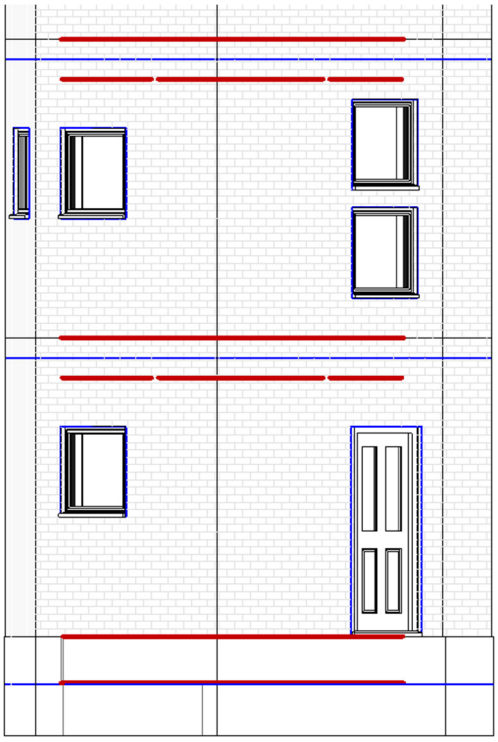


Figure 08-6 Storey Partition Geometry
Red lines represent the storey partitions
Blue lines represent the created geometry

Calculate Storey Partition Thicknesses

The rest of the floors are combined with the ceilings and grouped by level – as defined in the Revit model. This is done by reading the Revit parameter "Level" of each element. A peculiarity in the ceiling elements in Revit is that their "Level" parameter points to the level below them, and not to the actual storey partition they are belong to. This is accounted for in the script by offsetting them by one level when grouping by level.

The algorithm then calculates the distance from the bottom of each element to the level. That is done differently for ceiling and for floor elements.

Floors: for each element, the thickness is added to the offset from the corresponding level.

Ceilings: Since the ceiling elements are hosted by the level below, the same method does not work. The absolute elevation of each element (base) is calculated, and then their corresponding level's elevations are deducted.

For both Floors and Ceilings, there are groups of nodes in the script named Preserving Indices. Their purpose is to prevent elements being assigned incorrect level, in case there are not such elements on every level. Preserving Indices maintains a list of elements of the same length, as the amount of Levels in the Project. If there are not elements on each level, placeholders are used to populate the respective Level.

At the end, all calculated storey partitions for each level are combined. Since, the script can only create a single surface for each storey partition of the building, a single value needs to be chosen for its thickness, and respectively its elevation. As discussed in sub-chapter 06.5. Exported Storey Partition Elevation?, after a discussion, it has been decided that the maximum thickness of the storey partition is chosen and thus, that is the value that is passed onward to the rest of the script.

Elevation Correction

The purpose of this group of building blocks is to correct the elevations of the wall-lines using the Z-corrections calculated in “Storey Partition Thicknesses”, and assure the lines of the internal and external walls respectively lie on the locations that are required. The group of building blocks represents all nodes from the script, on the same relative location of the canvas, grouped under blue colour. The input of the group of blocks are the internal and external wall lines, on their original locations, while the output is the same lines with corrected elevations and external lines offset outwards to the outer wall face. Visualisation can be observed on Figure 08-7 Wall Location Lines. The blue lines represent the original locations, and the black lines beneath them – the corrected locations. Not that the lines at the lowest level are moved down by a larger distance, since that is the lowest floor of the building. It should be noted that the object Curve in Dynamo can often represent straight lines, and as such, some of the line-objects are sometimes referred to as “curves”. For the scope of this project, both terms are used interchangeably.

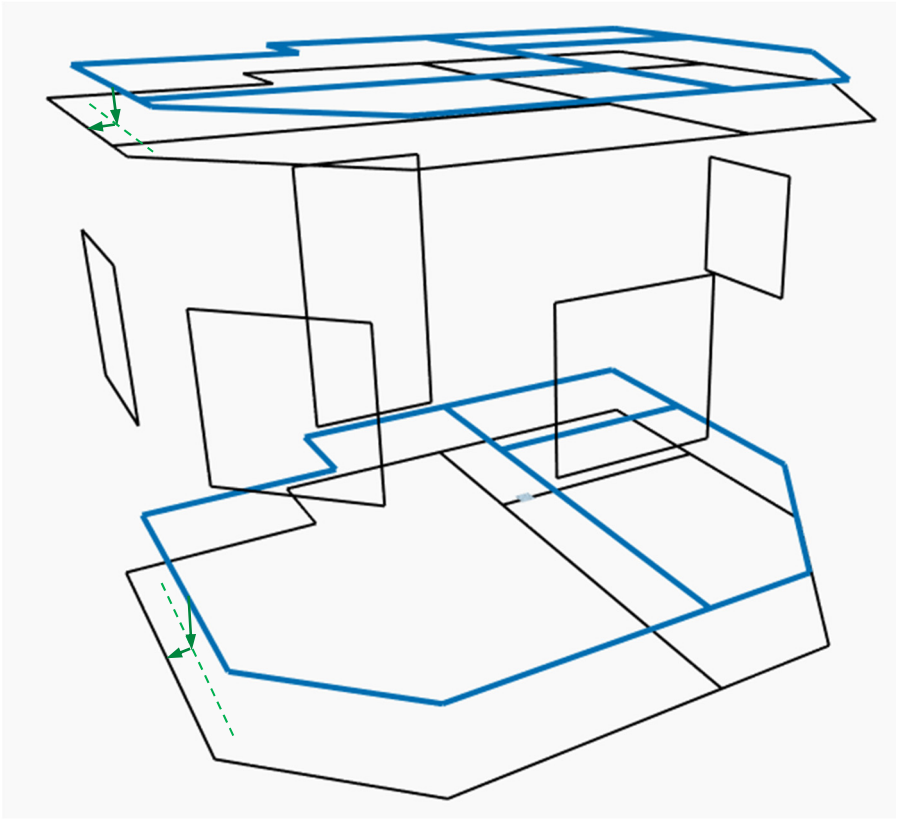


Figure 08-7 Wall Location Lines of Level 0 and Level 1

Wall Curves Z-Correction

Using the location lines of the imported internal walls from the building block “Filter Internal Walls”, seen in blue on Figure 08-3 Scheme of Canvas (Left Side), a correction in elevation is performed with the respective Z-correction, from the group “Storey Partition Thicknesses”. The same is done for the centre-lines of the external walls, which are imported from the building block “Import External Walls”. On the same figure, the downwards pointing green arrows show the Z-correction, and the green dashed line – the corrected location of the wall centre-lines. Note that the external walls are all walls from the Revit model, while the internal walls are only the ones in contact with analysed rooms.

Offset External Wall Curves Outside

The imported external wall lines are located in the centre of the walls, while the geometry needs to be generated on the outer face of the walls. Thus they need to be offset by half of the thickness of the walls towards the outer face. The horizontal green arrows on Figure 08-7 Wall Location Lines represent the direction of offset. There are two ways to offset a curve in Dynamo – offset in a direction, determined by the way the curve is created, or explicitly specify direction to offset. Since dealing with multiple curves, rotated in different directions on the XY plane, determining offset direction for each curve individually is resource demanding. However relying on consistent method of creation of the curves is not reliable. Therefore, the solution used in the algorithm is to combine the curves of external walls of each floor to a polygon and offset it twice: once with the distance equal to half of the thickness of the wall, and once with the same distance, but with negative value. All offsets are being done in the same plane of the polygon, therefore, there are only two options of offset. After that, the two polygons are compared and the larger of the two is being selected.

The reason the Z-correction is done before the horizontal offset instead of after, is that the centre-lines are required to be on the correct elevation because they are used in later in the script in the building block “Extend Internal Walls to External Walls”.

Top Wall Curves

Since all curves are created at the bases of the walls, until this point there is nothing defining the geometry of the curves of the roof. These are created by reading the height of the external walls of the top floor, and copying the external wall face-lines of the top floor by the same distance.

Trim Walls

Seeing as initially, all wall lines are created at the centre-line of the walls, when offsetting outwards the lines of the external walls, the ends of the internal wall lines are being disconnected from the external wall-lines, as seen on Figure 08-8 Internal Walls, Which Need Extending. The solution is to extend them, so they intersect the external wall-lines. Typically drawing softwares have trim/extend feature but Dynamo does not have one natively, so the algorithm is created manually. Dynamo distinguishes each line with Start and End, each of which needs to be processed separately.

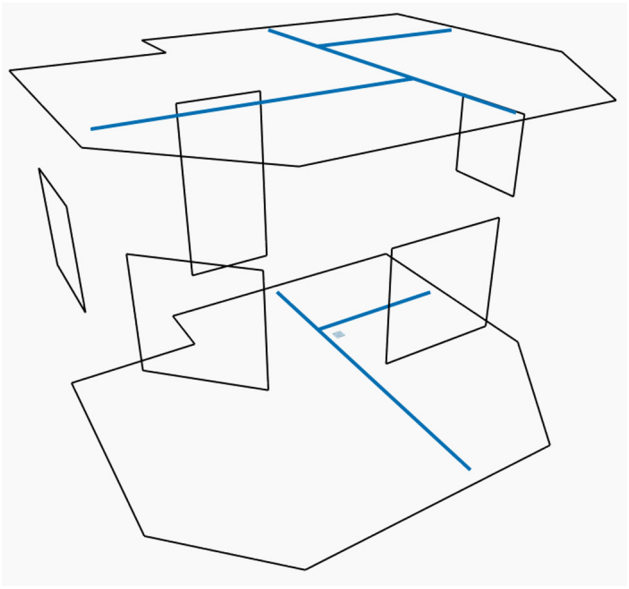


Figure 08-8 Internal Walls, Which Need Extending

Does the Start/End Point Intersect with External Wall Centre-Line?

Determining which lines need to be extended, means determining which lines intersect with external walls. That is achieved by testing if their start points lie on any centre-line of external wall. That is later done for their end points. It is important to do the tests one after another and not simultaneously, because each end needs separate processing.

Extending Lines

After determining whether a line start or end should be extended, the corresponding distance is calculated to the nearest wall face line, and the internal wall-line is then extended by that number. There is one scenario of internal wall meeting external wall in a T-section, which causes issues. Refer to Figure 08-9 Wall Lines – Problematic Area below.

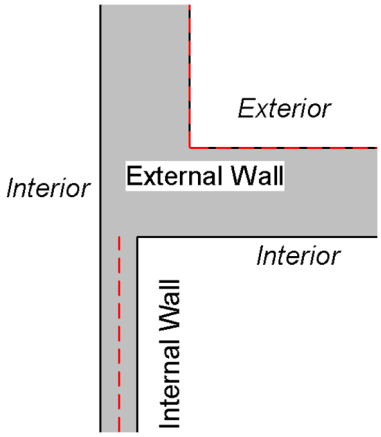


Figure 08-9 Wall Lines – Problematic Area

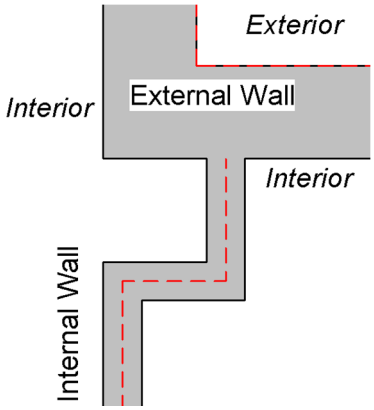


Figure 08-10 Wall Lines – Suggested Solution

Extending the internal wall line by any amount does not result in a connection of the wall and the external wall-line. The suggested solution is to modify the wall in the Revit model, as shown in Figure 08-10 Wall Lines – Suggested Solution. This way its centre-line correctly extends to the external face-

line. The same situation causes another issue, described in group of blocks “Lines of Analysed Roms”, and requires more specific needs for the workaround, defined in the description of the group later in the sub-chapter.

Split Walls

All wall lines until this point of the script span the entire length of the wall element in the model. However, for the creation of the geometry later in the script, it is required that all wall-lines are split at every intersection with another line. Or in other words, all wall lines should only intersect another wall-line at either its start- or end-point, not anywhere in-between. Since Dynamo does not have such intersection tool that can be used for this purpose, it is created manually.

Extract Intersection Points

Firstly all lines are intersected with all lines using Geometry.Intersect node. That causes every line to inevitably be intersecting with itself, which results in a line segment with the same length and location as the original. From the aggregated list of geometry, only the points are extracted and any duplicated points are deleted.

Filter Out Walls Not Containing Intersection Points

The next step involves taking all internal and external wall-lines from the previous building blocks and filtering only the ones, which intersect with any of the intersection points.

Split Walls on Intersections

This is the main part of the algorithm that finds which walls need to be split and splits them at the correct location. The group of nodes in the script is shown in Figure 08-11 "Split Walls on Intersections" Nodes below.

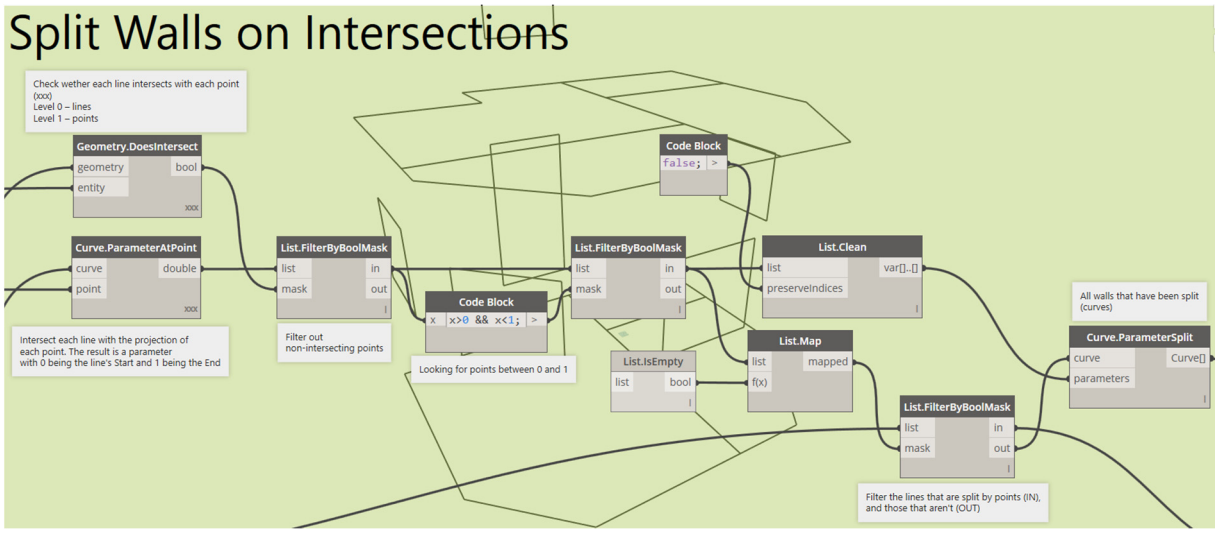


Figure 08-11 "Split Walls on Intersections" Nodes

Using the node Curve.ParameterAtPoint, allows to see at what distance from the start of a line, a given point lies. The result is a parameter with value of 0 for the start of the line and 1 for its end. So a value of 0.5 would mean the point lies exactly in the middle of the line. This is used to find which wall lines contain intersection points anywhere between their start and end. The result of the node for each of these lines is 0<X<1. Then on these lines the node Curve.ParameterSplit is used to split them at the same parameter value. At the end all wall-lines are combined in a list and duplicate lines are removed. Since this list is not sorted in any particular order, the following building blocks in the script filter the lines in two groups: external and internal walls, and after that group each of them by level

Create Floor Solids

This group contains one building block, whose purpose is to generate solid objects for each floor of the building. The script later splits these solids into rooms, and further – each room (solid), to faces (surface), then edges, and then vertices.

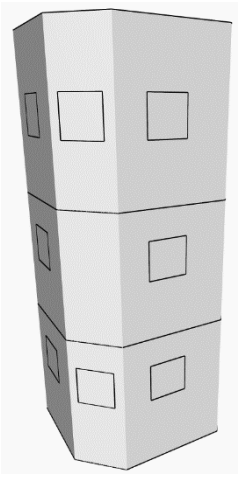


Figure 08-12 Building Solid

Whole Building Solid

The first step is creating a solid for the whole building. That is done by creating a “loft” from the bottom and top level external wall-lines, using the node Solid.ByLoft. The result can be seen on Figure 08-12 Building Solid to the right.

Splitting Building Solid to Floors

Next, the building solid needs to be split for each floor. Dynamo does not offer a node that can do this operation easily, so a more creative approach is taken. The node used is Geometry.Trim. It splits a geometry in two, using a “tool” (a surface), and deletes one of the two sides, determined by a “pick” (a point). It has three inputs: geometry (to be trimmed), tool (to slice the geometry), and pick (point, determining which side of the geometry delete).

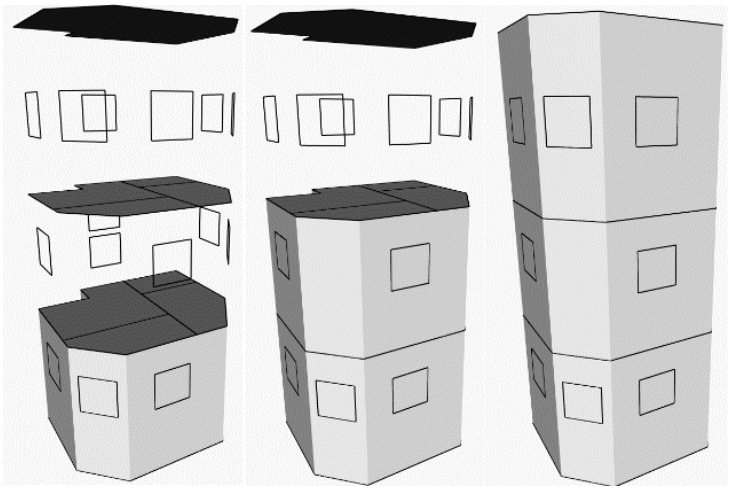


Figure 08-13 Initial Results of Geometry.Trim

Since the building needs to be split several times, the “tool”-s need to be the storey partition surfaces. However, if the node is given severalo “tol”-s at once, it slices the original geometry once for each of the tools, instead of all at once. If the top geometry is deleted, the result can be seen on Figure 08-13 Initial Results of Geometry.Trim. The algorithm then uses the node again on each of the

results, this time deleting the lower geometries, and the results can be seen in Figure 08-14 Building Floors – Final Result below.

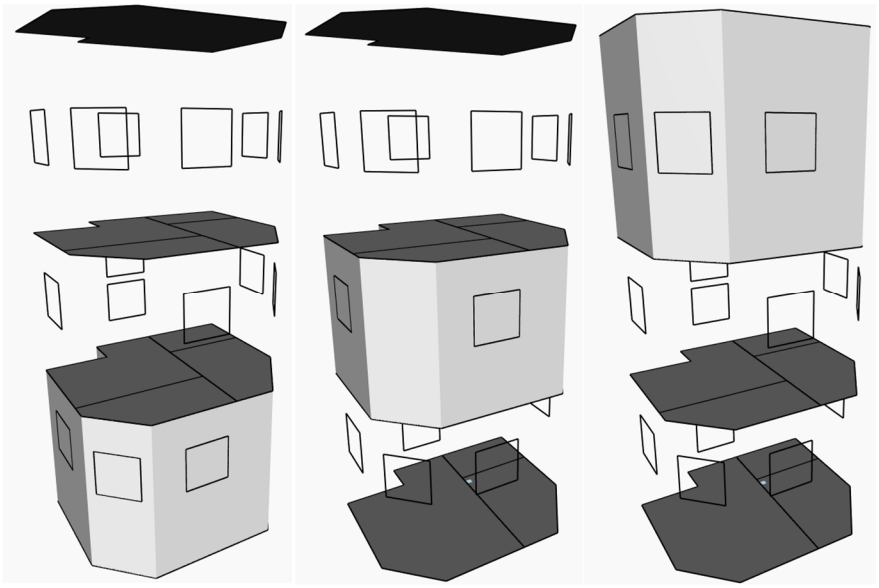


Figure 08-14 Building Floors – Final Result

The method of using Geometry.Trim two consecutive times is creating two pick points – one above and one below the building. Then use the node for the whole building with the first pick point (above the building). That gives the results in Figure 08-13. Then for each of the results, excluding the first one, perform the Geometry.Trim, using the second pick point (below the building).

Lines of Analysed Rooms

The purpose of this group of building blocks is to generate the geometry of wall-lines around each analysed room, which will serve as a base for creation of room solid later in the script. It consists of two building blocks – Surface of Internal Walls and Wall Lines of All Analysed Rooms. The first one generates surfaces representing the internal walls, to be used in geometry intersection later. The second one searches through all of the created wall-lines in the previous blocks, and finds the correct match for each of the walls for each of the analysed rooms. There are methods used in the algorithm to minimise the amount of lines that are searched. Both building blocks are described below.

Surfaces of Internal Walls

All lines of internal walls are extruded by 10 metres to form a surface. All surfaces are “intersected” with the solid of the floor they belong to, to cut them down to the correct height. Afterwards the surfaces of each floor are combined in one “polysurface”, which is an element in Dynamo that represents a collection of surfaces and acts as one solid object. That is required for later processing.

Wall Lines of all Analysed Rooms

The method used in the first part of the building block is using the custom node Room.Boundaries from Clockwork package, which creates lines at the edges of the floor of each of the analysed rooms, referred to as “room lines” from here on, for clarity. These lines are matched with their corresponding lines generated thus far in the script, referred to as “final lines” from here on. The final lines represent the wall-lines with correction for the elevation and horizontal offset. On Figure 08-15 Room Lines and Final Lines below, the room lines are shown in blue, the final lines are shown in red, and green arrows represent the correction in elevation and horizontal offset. There are four analysed rooms on the shown example, therefore the blue lines form four polygons.

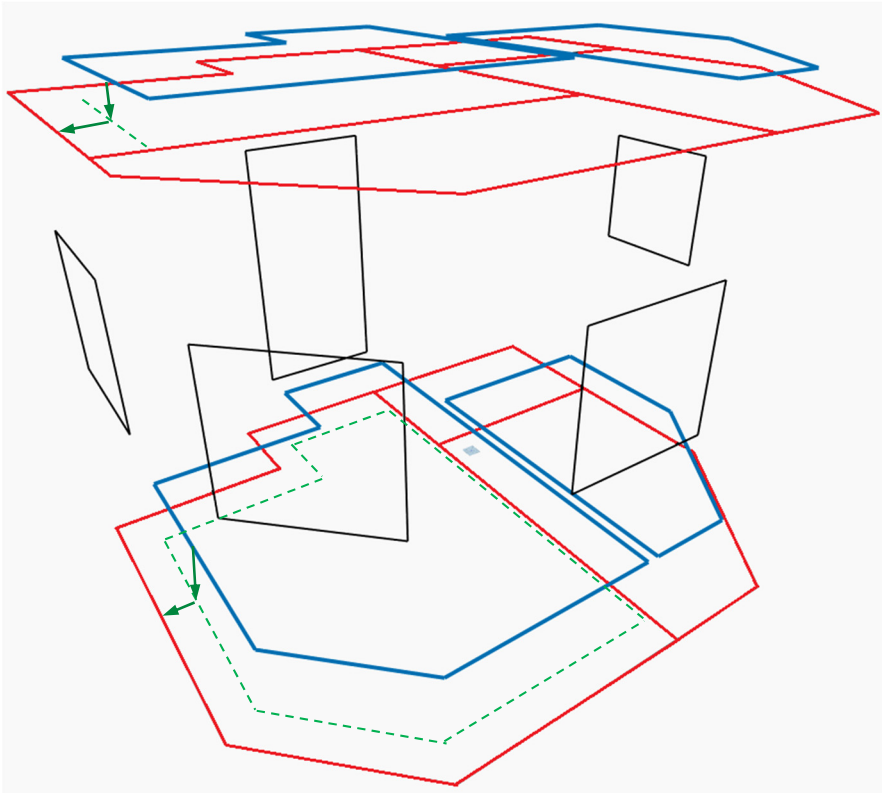


Figure 08-15 Room Lines and Final Lines

A horizontal plane is created for each of the rooms, with the elevation of the room lines. That surface is used to intersect the internal wall surfaces to determine which walls correspond to which room. On Figure 08-16 Intersecting Room Floors with Internal Walls to the right, the floor surfaces are shown in blue, and the wall surfaces – in grey. The result of this operation is that all wall lines, which are on a floor with at least one analysed rooms, are extracted to a “pool” of wall lines. That pool is used for searching in the following matching algorithm.

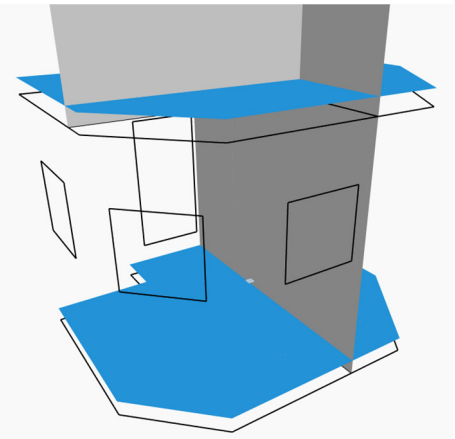


Figure 08-16 Intersecting Room Floors with Internal Walls

The following part of the script is used for the actual matching of room lines and correct lines. Figure 08-17 Wall Intersection Algorithm below serves as a visual representation of the logic. It represents the workaround described in the “Trim Walls” building block, but with additional details. The room lines are shown in blue and the correct lines are shown in red.

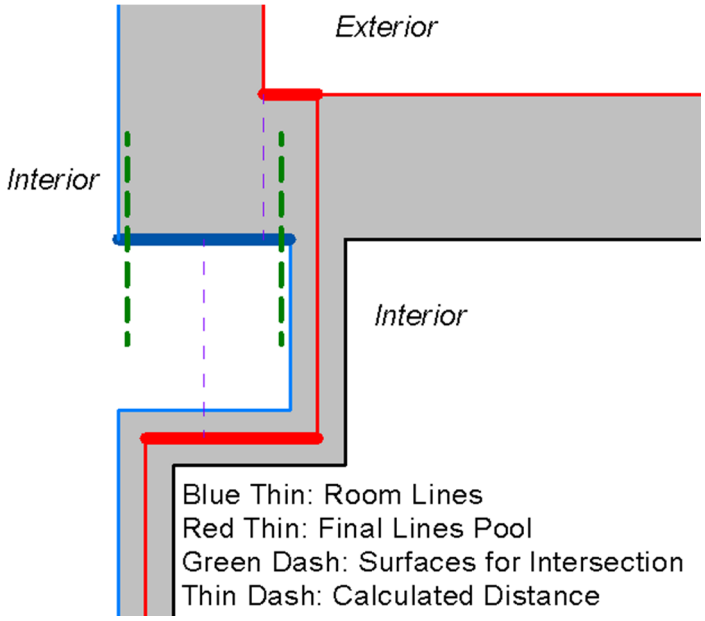


Figure 08-17 Wall Intersection Algorithm

Two planes generated near the start and end of each room, which are perpendicular to the line. The example shown in Figure 08-17 above describes the calculations for the room line shown in thick blue colour. The two planes for this line are shown as green dashed lines. These planes are used to intersect each of the lines in the pool generated in the previous step (shown as red lines). On the figure, the lines that are being intersected by at least one of the planes are shown wider than the rest. After the intersection the algorithm calculates the smallest distance from the room line to each of the intersected lines (shown as wide and red colour) and chooses the closest one as the best match. Performing these steps for each of the room lines, results in a closed loop of correct lines, which are used later in the script for generating room solids.

As seen in Figure 08-17 of this example, if interior wall, shown as thick red line, is too close to the calculated room line, the algorithm would inaccurately chose it as the best match, and result in overlapping lines which do not form a closed loop. That will subsequently result in not being able to create a solid for the room and inevitably the script not being able to output correct geometry. Therefore, as described in sub-chapter 08.3. Script Constraints a safe clear distance between the external wall and the parallel internal wall segment is at least as much, as the thickness of the external wall. The same is true for the segment of the internal wall, intersecting with the external wall: the distance between its face (facing west on Figure 08-17) and the face of the parallel external wall

(facing the same direction) needs to be at least as wide as the thickness of the external wall. This requirement is also explained, in a simplified way, in the User Guide in chapter 09. User Guide.

Had the surface used for intersection been only one instead of two, as shown on Figure 08-18 Wall Intersection Algorithm – Alternative, then it would have only intersected the wrong line of the pool (shown as wide and red colour). That is the reason why the algorithm uses two surfaces at the start and end of each room line. That way the movement of the walls in the model for the workaround does not need to be as drastic.

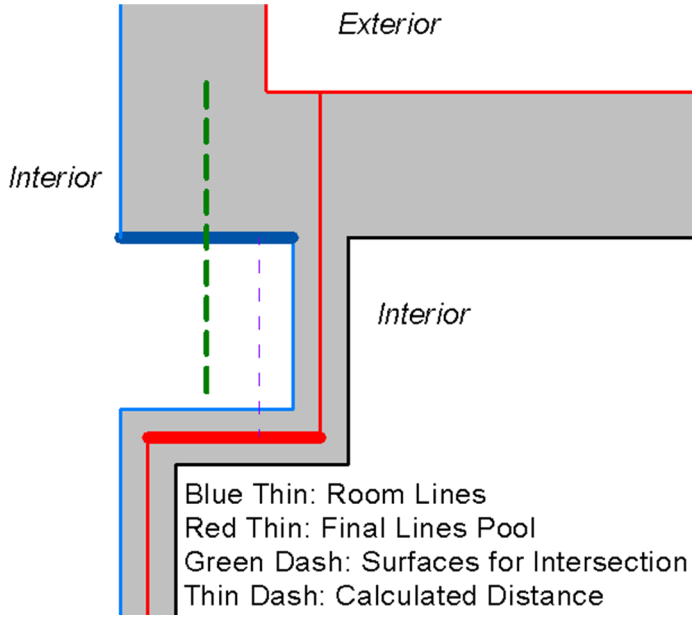


Figure 08-18 Wall Intersection Algorithm – Alternative

Create Room Solids

This group of building blocks creates solids of each analysed room and combines the rest of the space on each floor in as few rooms as possible. That is done to improve the efficiency of the script and decrease the complexity of the output file.

Using the closed wall lines for each room as a base, a solid is made by extruding the form vertically by 20m using the node Solid.BySweep. The solids of each room are intersected with its corresponding floor solid, generated in building block “Create Solids of Each Storey”, as seen on Figure 08-19 Intersecting Room and Floor Solids.

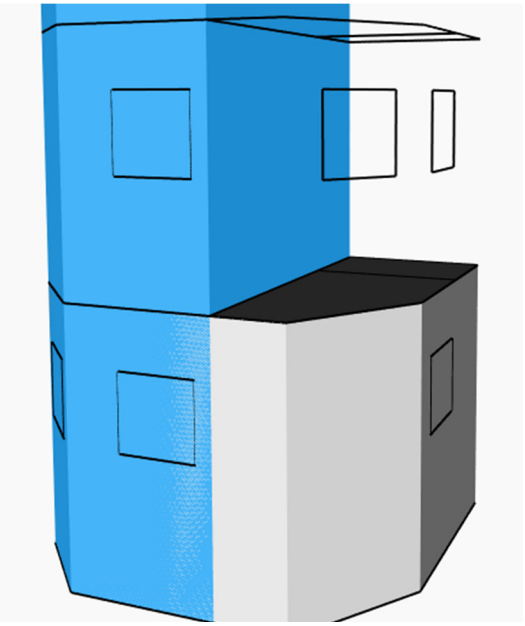


Figure 08-19 Intersecting Room and Floor Solids

The resulting geometries have the exact dimensions of the rooms, needed for output. Refer to Figure 08-20 Solids of Analysed Rooms below.

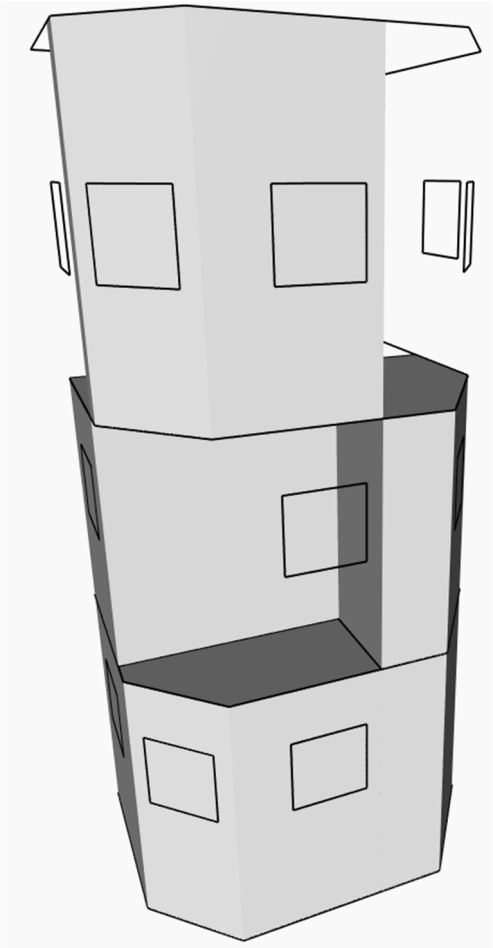


Figure 08-20 Solids of Analysed Rooms

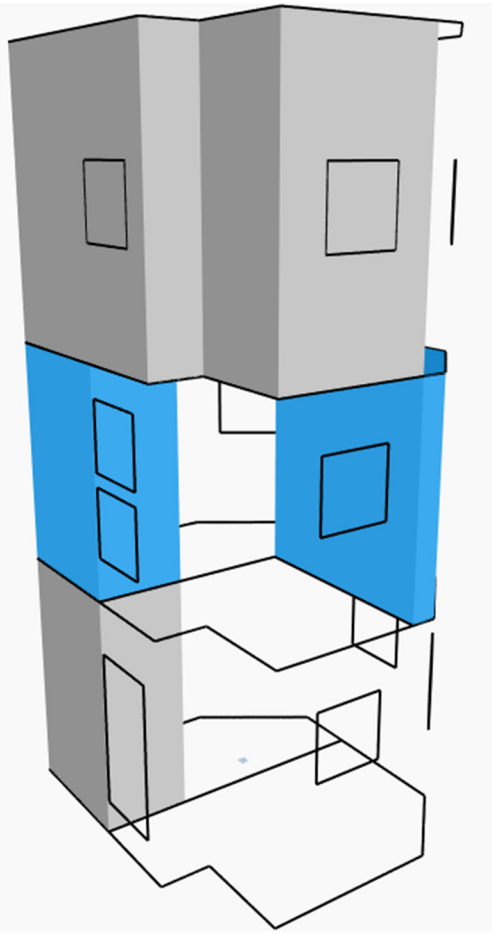


Figure 08-21 Solids of Rest of the Building

The node Solid.DifferenceAll is then used to remove the volume of these solids from the floor solids. That results in the rest of the space of each floor to be combined in one solid. However, if there are more than one formed solids on each floor, even if they are physically separated, in Dynamo they behave as one unit. As seen on Figure 08-21 Solids of Rest of the Building above, the blue coloured spaces behave as one object. That would cause them to be exported as one room, even though they should be two.

The solution is to convert them to polysurfaces and then back to solids. That creates a separate solid for each of the enclosed spaces. Then both lists of rooms are combined and passed to the rest of the writing part of the script

Write to File

This group of building blocks comprises of significant amount of the nodes in the script, however this description does not go in details of every building block. Instead it presents the general structure and describes the logic used in defining the unique ID's of the objects (RID).

This group of blocks can be divided in two algorithms running side by side – RID's and geometry. The geometry part extracts smaller objects from bigger objects and preserves the hierarchy and list-nesting necessary for proper exporting.

The RID's of the objects are required to be unique. They need to start with # string, followed by series of numbers. The script uses certain positions of the RID for specific object identification. The first characters are reserved for the ROOM object. The amount of characters saved depends on how many rooms there are in the file. The leading character is always 1, followed by the consecutive number of room with zeroes used as fillers. So, as example, if there are 12 rooms in the project, rooms will have consecutive numbers from 01 to 12. Including the # and the leading character of 1 creates a series of numbers like #101, #102 … #112. The reason for including 1 in front of the number is to keep the same amount of digits in the RID's of all objects of the same type.

The following character of the RID describes either a CELL or a WINDOOR. A cell always contains one ROOM, so the RID of a CELL is always the RID of its ROOM with 0 added in the end. If the number is 1, it describes a WINDOOR with consecutive numbers following.

The following characters describe a FACE. Each CELL has several faces and each face's RID starts with the RID of its CELL and its consecutive number. This consecutive number is always the same amount of digits with leading zeroes as fillers.

The following character is FACE_SIDE. Each FACE always has two FACE_SIDE's, therefore this number is always 1 or 2, if the RID is of a FACE_SIDE, or 0, if the RID is of a EDGE, VERTEX, or VECTOR3D.

The following numbers represent EDGE, VERTEX, and VECTOR3D respectively, and each of the series maintains the same amount of digits with leading zeroes. The RID scheme looks as presented in Figure 08-22 RID Scheme.

	A
1	rid
316	#1500500310

Figure 08-22 RID Scheme

Green: ROOM (1+consecutive number)
Red: CELL (0) or WINDOOR (1)
Blue: FACE (consecutive number)
Yellow: FACE_SIDE (1 or 2) or EDGE, VERTEX, or VECTOR3D (0)
Purple: EDGE (consecutive number)
Grey: VERTEX (0 or 1)
Black: VECTOR3D (0)

08.3. Script Constraints

There are things that the script cannot do, either because of a compromise, due to the resource requirements, or due to limitations set out by Dynamo itself. This sub-chapter describes these constrains. Most of them can be fixed by incorporating more complex algorithms, with the drawback of making the script more recourse demanding, and all of them could be fixed by using code in Python, overcoming the main constrain of the visual programming environment of Dynamo – lack of well implemented Looping.

Some of the information in this sub-chapter may overlap with some already presented information but it is placed here because it is relevant and in the case of some reader being interested in this particular topic and not much in the others.

External walls need to be connected in one chain

As described in the previous sub-chapter (08.2. Building Blocks), this is a constraint that is intentionally set, with the purpose of simplifying the script and improving the calculation time and resource requirements. Having the walls in such order allows for the extracted centrelines to form one connected polygon, which is then offset outwards to meet the wall faces.

It is considered that most buildings, which are going to be analysed, do have all external walls in one connected chain. In case of the presence of additional buildings in the same Revit file, they can be deleted before execution of the script.

External walls need to have the same width

The purpose of this constraint is to allow the formed polygon of wall centrelines, described above, to be offset by the same amount and meet the external wall faces. Had some walls been of different width, then the polygon would have had to be offset by different amounts in its different segments. Alternatively, before creating the polygon, each individual wall line could be offset by the corresponding value and then the lines trimmed to meet each other in a polygon. Dynamo does not have native “trim” node, therefore that would have to be created manually, adding additional computational time and resource demands.

Uniform storey partitions throughout the storey

This constraint is that for each storey partition of the building, all areas, parts of analysed rooms, need to have the same thickness. Having different thicknesses throughout one storey partition would require change in elevation of the created floor-surface in the script, due to the requirement to have the surface lying in the middle of the thickness of the storey partition. This difference in elevation of the surfaces, need to be connected together. The surface that is required to connect them is not representative of a floor in the model, therefore would need to be created in the script, which is significant addition to the complexity.

Sloped ceilings are not supported

The script is designed to be able to create only vertical and horizontal surfaces. Creating surfaces in other planes requires different algorithms that are not implemented in the current version to keep the script less complex.

Curtain walls are not supported

After testing it was found that curtain walls are not fully supported. When curtain walls are used as normal walls, the script does not consider them as “windoor” elements, which, for the purpose of correct output, it should, but as normal walls, since that is their type in Revit. This results in having them represented as normal walls in the output file, which is a problem, if the curtain wall represents a glazing façade, or any kind of transparent surface, allowing light to come into the building. Solution to this issue is somewhat more complex than adding a simple filter to the script, because these elements are not “hosted” by another wall. And a “windoor” element needs to be hosted by a wall element in the output file. Thus, it requires generating a wall that is not currently in the Revit model, and the script does not contain such algorithm, hence such has to be created.

Another use of curtain walls is when they are embedded in another wall. That case does not require generating a new geometry, not existing in the input model, but it does involve the problem that Dynamo reads these walls as one inside another. And since the curtain wall element is neither a window, nor a door, it attempts to create a wall inside a wall, which is not a valid construction and thus fails to finish execution properly. A solution is to treat each instance of embedded curtain wall as a window or a door. That would require a filter to all external walls to find any walls that are of such type and process them as “windoor”.

Additionally there needs to be a filter to distinguish embedded curtain walls from ones that are not. Because each needs different kind of processing.

Floor covering – no support for stacked separate floor elements

A practice used in some architectural models is to create a common floor element for the load bearing construction of the floor (with negative offset from the floor level) and creating another floor element with the floor covering, for example parquet, tiles, or carpet. The way the script works is, it starts from the room and finds all elements that are in contact to the room. That means that the floor it finds in this case is the covering element, but not the load bearing element, which is of higher importance. There are ways to improve on the algorithm and make it more sophisticated, described in chapter 10. Future Development, but the solution for the current version of the script is to edit the Revit model by doing one of two things:

1. Split the load bearing floor to the boundaries of each covering, include the covering as top layer in the element and delete the covering element itself. After that correct the offset of the

floor to match the level of the room. That keeps the same construction as designed but has only one floor element in the room and is correctly processed by the script;

2. Delete the floor covering element and correct the offset of the load bearing element to match the designed floor. That solution is simpler but changes the construction of the floor element. However, if considered that this has negligible effect on the simulations, it may be a faster way to prepare the Revit model

Internal and external wall connection problem in specific situation

There are situations when internal wall meets an external wall in a way, shown in Figure 08-23 Wall Connection Problem. This happens when an internal wall meets parallel external wall, and is located on the inside of the external wall's external face, and only if the walls belong to a room that is marked as "analysed". Note that wall elements in Revit may span multiple rooms.

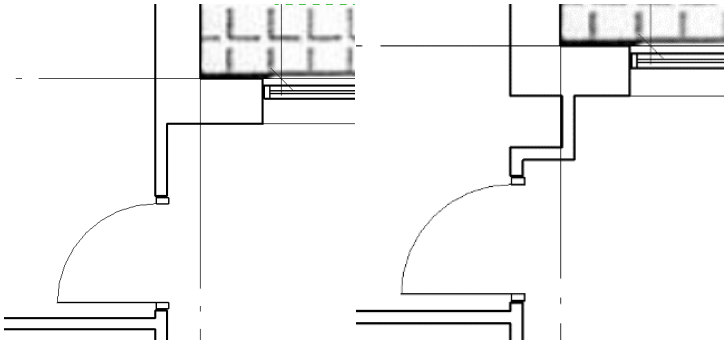


Figure 08-23 Wall Connection Problem (left) and Solution (right)

The details on the solution are explained in the User Guide. The reason for the problem occurring in this situation is that there are two places in the script, where such geometry causes problems – when the internal walls centrelines are extended to meet the external wall's external face, and in the creation of the room solids that are extracted from the overall building floor solid. The summary of the two algorithms is presented below:

Extending internal wall centre-lines to external walls external faces

As described in sub-chapter 08.1. General Structure, as step 7, at that point the script has all internal walls represented by their centre-lines, and all external walls – as lines, lying at their external faces. As shown on Figure 08-24 to the right, internal walls end at the point of meeting the inner face of external wall. Their centre-lines are not reaching the external faces of the external wall. The internal wall lines needs to be extended to meet the

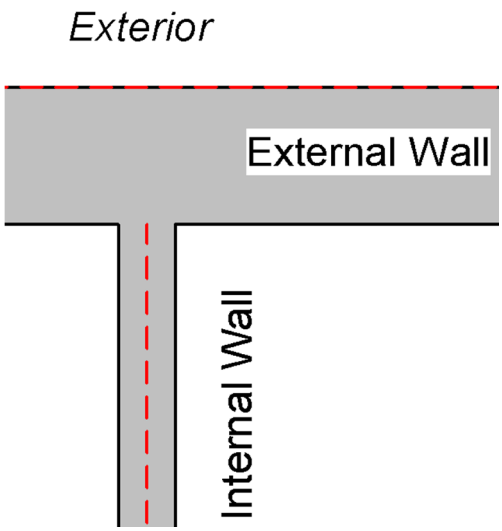


Figure 08-24 Wall Lines Diagram

external wall line. But a problem occurs if they actually cannot intersect the external wall's lines. Refer to Figure 08-26 Wall Lines – Problematic Area.

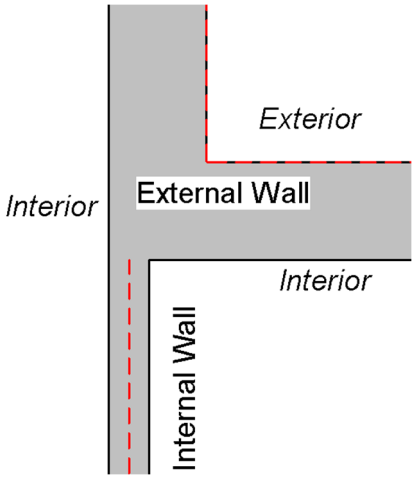


Figure 08-26 Wall Lines – Problematic Area

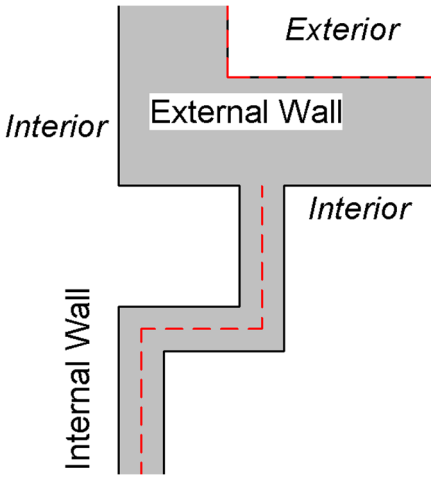


Figure 08-25 Wall Lines – Suggested Solution

As can be observed, regardless of how much the internal wall line is extended, it will not intersect the correct external wall line. This requires that a simple workaround takes place. Refer to Figure 08-25 Wall Lines – Suggested Solution. By changing the internal wall path in the shown manner, the centre line can be extended to the line of the external wall with no problem. The added volume to the room on the left will alter the results of the simulation but not by a large margin.

This suggested solution does solve the problem, however, as mentioned earlier, there is another algorithm to consider.

Generating room solids

As described in sub-chapter 08.1. General Structure, steps 11 and 12 involve creation of solids of the rooms that are analysed in order to subtract them from the volume (solid) of the whole building floor. One difference between the volume of the rooms that need to be generated and the rooms in the Revit model, is that the floor of the generated room (bottom face of solid) lies in the middle of the storey partition, while the bottom of the room in the model lies at the top of the storey partition. This means that when creating the solid of the room, each of the lines forming the bottom face, needs to be "moved" *down* to match the correct elevation. Moreover, since the bottom face of the volume of the Revit room is bounded by the internal surfaces of all walls, they also need to be moved *out*, to match the correct location of either centre-lines (for internal walls) or face-lines (for external walls). Figure 08-27 Correction of Wall Lines below gives a visual representation.

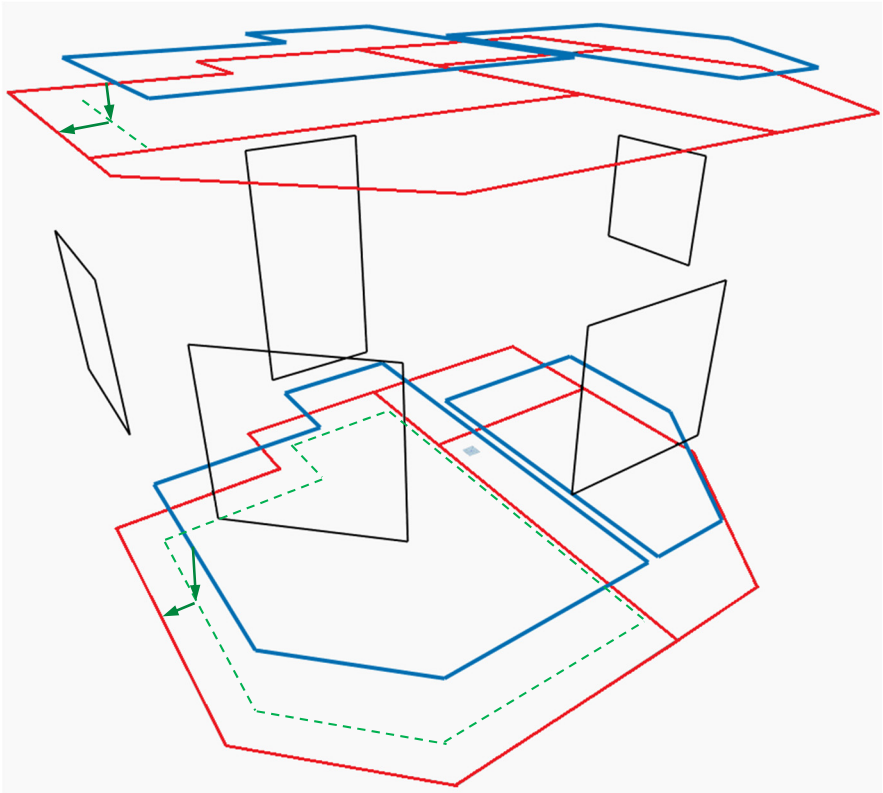


Figure 08-27 Correction of Wall Lines

In aim of doing this complicated calculation as simple as possible, the matching is done by an algorithm that finds the closest already generated line (the red lines in Figure 08-28 Wall Intersection Algorithm to the right) to each of the lines in the base of the Revit model room (blue lines in the figure). The algorithm creates two surfaces on each line near the start and end (shown as green dashed lines) that are perpendicular to the line. Then only searches through lines that are intersected by one of the two surfaces.

That creates the requirements for the size of the area added to the room. Each side of the formed "square" needs to be at least as long as the width of the external wall.

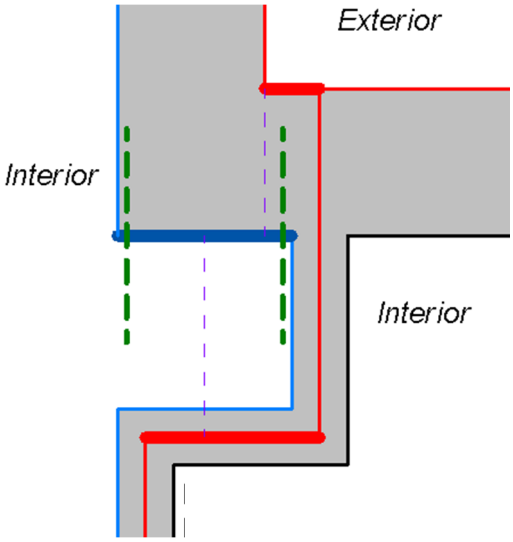


Figure 08-28 Wall Intersection Algorithm

09. User Guide

This chapter is aimed at the user of the script and summarises a lot of the information described in chapter 08. Dynamo Script Design in the form of simplified and easy to understand requirements and recommendations. It comprises of two sections: Requirements and Recommendations

09.1. Requirements

This sub-chapter describes what the user is required to do in preparation of the model, in order to have successful execution of the script and result, matching the expectations. These requirements are caused by limitations of the script or the software (Dynamo), which are described in more details in sub-chapters 08.3. Script Constraints and 05.03. Software Limitations.

I. Software and Packages

It is required to have minimum versions of Dynamo and the used additional packages equal to the ones used for creation of the script. If later versions are used and errors occur, downgrade to may be required. The version used are:

Dynamo 0.9.1.3703
archi-lab.net: 2015.11.12
Clockwork for Dynamo 0.9.x: 0.90.3
If Equal Return Index: 0.1.0
LunchBox for Dynamo: 2015.11.25
Rhythm: 2015.11.25

II. Walls

All walls in the building are required to be connected in *one* chain and have the same thickness. If there are separate structures in the same model file, the script will not be able to manage it and will likely stop execution before finishing, or export incomplete or wrong data in the output file.

The external walls do not need to be of the same type but need to have the same construction. A workaround, however not tested, could be to add an internal layer of “material” air to the thinner external walls to match the thickness of the rest. That air material should be defined in the wall family in Revit as having the lowest priority function – Finish 2 [5]. This preserves as much as possible the designed connections between the wall and other elements. Moreover, the material should have the parameter “Wraps” checked and “Structural Material” as *not* checked.

Furthermore, each wall type need to have the type parameter “Function” set appropriately. The script distinguishes external from internal walls using this parameter. If this parameter is set to anything other than “External” or “Internal”, the walls are not processed by the script.

III. Wall Connections

Revit allows to have two walls not forming a connection, if explicitly set to do so. Internal walls, reaching external walls need to be connected for the script to process walls correctly. If the wall joint is “disallowed” in the model, the script will not manage to distinguish the adjacent rooms as separate. Example of incorrect and correct connections can be observed in Figure 09-1 and Figure 09-2 below. A correct connection, shows the end point marker of the internal wall (when the wall is selected) in the centre of the external wall.

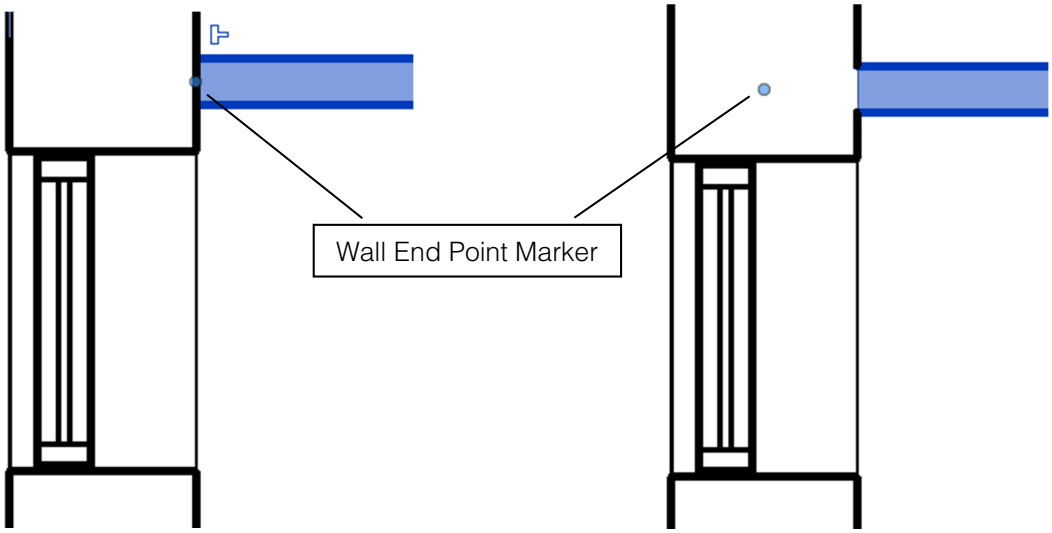


Figure 09-1 Incorrect Wall Connection (Plan View)

Figure 09-2 Correct Wall Connection (Plan View)

Another wall connection that needs attention is a place where internal wall connects to a 90° external wall, as shown in Figure 09-3 Wall Connection Problem below. Due to the algorithms used in the script, such connection cannot be processed correctly and a workaround is required. The internal wall needs to be split as shown on Figure 09-4 Wall Connection Solution below.

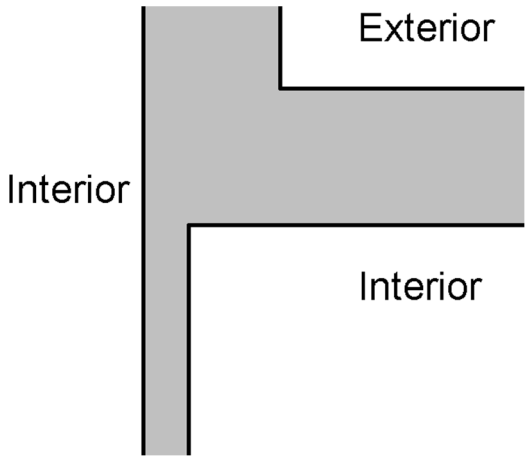


Figure 09-3 Wall Connection Problem

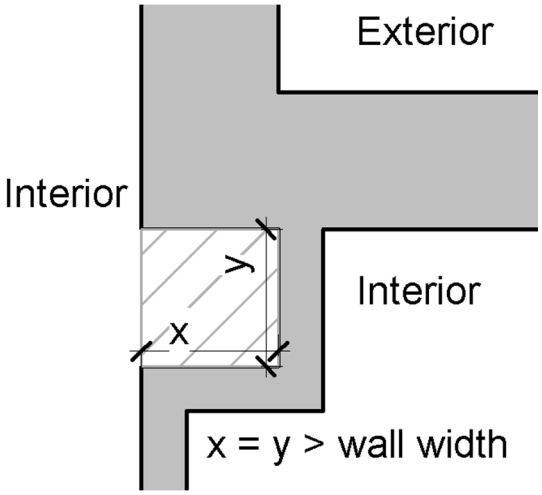


Figure 09-4 Wall Connection Solution

There is a minimum requirement for the two dimensions of the newly added area to the room (shown as x and y), such that both of them need to be *bigger* than the width of the external wall. Note that equal length is not enough. The reasons for this limitation are explained in details in sub-chapter 08.3. Script Constraints.

IV. Curtain Walls

The script does not support curtain walls. A suggested workaround is to delete the curtain wall and create a similarly sized window (or door) with similar parameters to the curtain wall. Alternatively, to simply exclude the curtain walls from the analysis, instead of deleting them, the wall type function parameter can be changed to anything different than “External” and “Internal”. That would cause the script to disregard this type of wall.

V. Windows and Doors

The script projects the geometry of the windows and doors to their host wall, after which finds the outermost points and generates the opening geometry. This process will produce inaccurate results if there is any geometry of the window, outside of the *projected* opening on the wall. This means that any geometry that is outside of the opening, but still within the boundaries of the projection, will not cause problems – for example a window sill, protruding out of the wall. A properly created family usually does not contain such geometry. If the connection of the element to the wall is meant to be shown, it is typically done via 2D representation, visible in plan or section view. These 2D elements are not geometry that is used for the projection in the script.

Some families, however, are created without much concern for this, since typically that would not cause a problem in a Revit model. Example of “geometry” that is occasionally placed outside of the opening boundaries is a flip-switch or a model line (3D object) that is supposed to be detail line (2D object). Since control over the creation of the families is usually not possible, and inspecting each element separately is very tedious task, a suggested approach is to run the script and visually see the results in the Revit model. The script creates 3D lines of the edges of the surfaces of all elements (windows, doors, walls, and storey partitions) in the Revit model. Example can be observed in Figure 09-5 Dynamo Geometry Visualisation below.

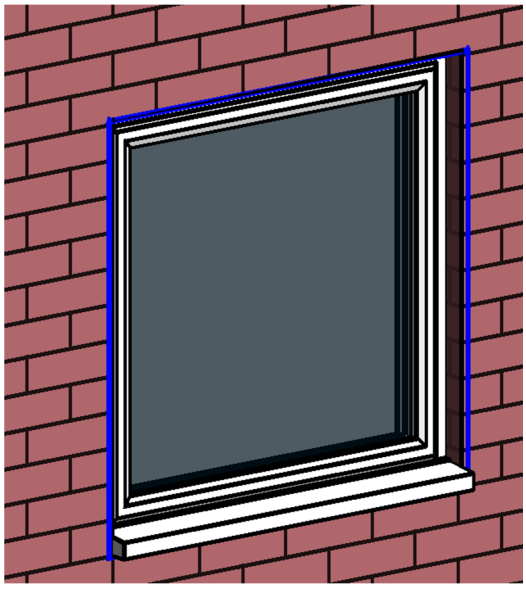


Figure 09-5 Dynamo Geometry Visualisation

On the figure, the blue lines represent the geometry, generated by the script. Typically, on a 3D view of the model in Revit, it is easy to identify lines that do not match the actual openings of windows and doors. These are the families that need to be revised for geometry outside of the boundaries of the projected wall opening. Revision of the elements could be a difficult task, considering that different families are created using different techniques and Revit many different ways to accomplish one goal. A suggested, more systematised method is, in Revit, to edit the family, go through each of the views, select everything, use "Filter" to filter out only annotations and hide them in view. After hiding the annotations (2D geometry), what is going to be left is 3D geometry, that can be checked one by one in search for any, which is outside of the borders of the wall opening.

In addition to the factors described above, the function family type parameters of all doors need to be set correctly. Using external door type families inside the building can create errors, and using internal type doors on external walls, will cause the doors to be ignored and not generated.

VI. Balconies and Terraces

The script currently does not support Balconies and Terraces. The script will succeed in execution, as long as the balconies or terraces are not adjacent to an analysed rooms. However, the geometry of the balconies and terraces will not be exported, and on their place, there will be only external walls, as if they had been closed off.

VII. Building Storeys Amount

The script supports only buildings that have equal amounts of storeys throughout. Some bigger residential buildings for example may have sections with different amount of storeys. Such buildings are not supported by the script, and could be analysed only if the different sections are analysed as separate buildings, or the difference in storeys is removed.

VIII. Storey Partitions

There are two storey partition requirements: all parts of each storey partition can be of only one thickness, and all storey partitions that need to be analysed, need to be bounding an analysed rooms.

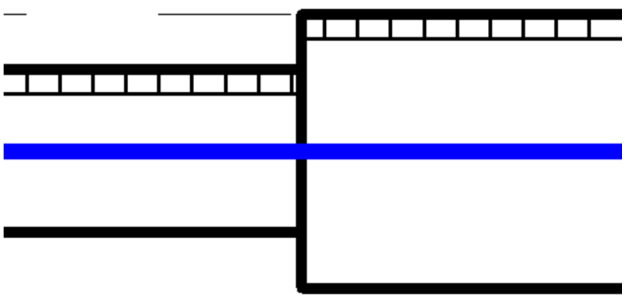


Figure 09-6 Floors with Different Thicknesses

If there are two rooms that are going to be analysed on one floor, and their respective floor slabs are of different thicknesses, the whole storey partition is being generated with a thickness equal to the thicker slab. This input model would not cause issues in generating the output, however, it may cause the output data to be inaccurate. In the described example, the thinner floor will be offset lower than the thicker floor, due to the reason that all storey partitions are aligned by their centre of thickness. Refer to Figure 09-6 Floors with Different Thicknesses below. The blue line represents the created geometry from the script.

The same logic applies to the ceilings of each room. Note that there may be storey partition with entirely different thicknesses in the model, but as long as they are not adjacent to any analysed rooms, their thickness is disregarded.

Additionally to the above described requirement, all floors need to be “bounding” a room. For Dynamo the term “bounding” refers to any element that is at least partly intersecting the room object in Revit. A room in Revit, although not visible unless selected, is considered a family and has some of the typical “Constraints” parameters associated with any family. In particular it has base and top Level and base and top offset. Although they are named slightly differently, it helps to consider them the same. Therefore, if the floor of a room has a negative offset from the level (similar to the one on the left on Figure 09-6 Floors with Different Thicknesses), while the room has 0 offset, that would make the floor *not* bounding for that room. A fix for such situation is to correct either the base offset of the room, or the top offset of the floor.

Another factor to consider is floor covering. It is not uncommon to have floor coverings, such as carpets, parquets, tiles, etc., created as a separate elements on top of a common structural floor. This scenario causes the script to identify only the floor covering as a room bounding element, disregarding the structural floor beneath, since it is a separate element, not in contact with the room. One solution to this is to delete the floor covering element and lower the base offset of the room, to match the offset of the floor beneath. That increases the total volume of the room but is a simple fix. If considered that the increase in volume is significant for the results and thus ideally avoided, alternative fix is to delete the covering, then split the structural floor in a way that creates a separate element just for the analysed room. Then duplicate the family type of the floor and add a top layer(s) representing the covering. In the end, correct the offset of the floor to match the original offset of the covering element. The net result from this operation is the same construction as designed, but having both floor elements combined in one.

IX. Sloped Ceilings and Sloped Windows

The script can only create horizontal and vertical surfaces. Therefore it cannot support sloped ceilings or floors, and any windows hosted in them. Having any of them present in the model can result in the script not completing, incorrect data output, or crash.

X. Room Requirements

Any room separator lines have to be flush with the wall face of the room, to prevent problems arising in the algorithm analysing room bounding elements. Figure 09-7 and Figure 09-8 represent incorrect and correct placement of a room separator.

Figure 09-7 Room Separator – Incorrect Placement

Figure 09-8 Room Separator – Correct Placement

There are two room parameters that need to be created in the Revit model, for correct execution of the script. One of them defines each room as being analysed or not and the other – which thermal zone it belongs to, for the purposes of following the energy analysis. The parameters are inserted by from *Manage* → ***Project Parameters*** → ***Add***. Refer to Figure 09-9 Revit Room Parameter Setup below.

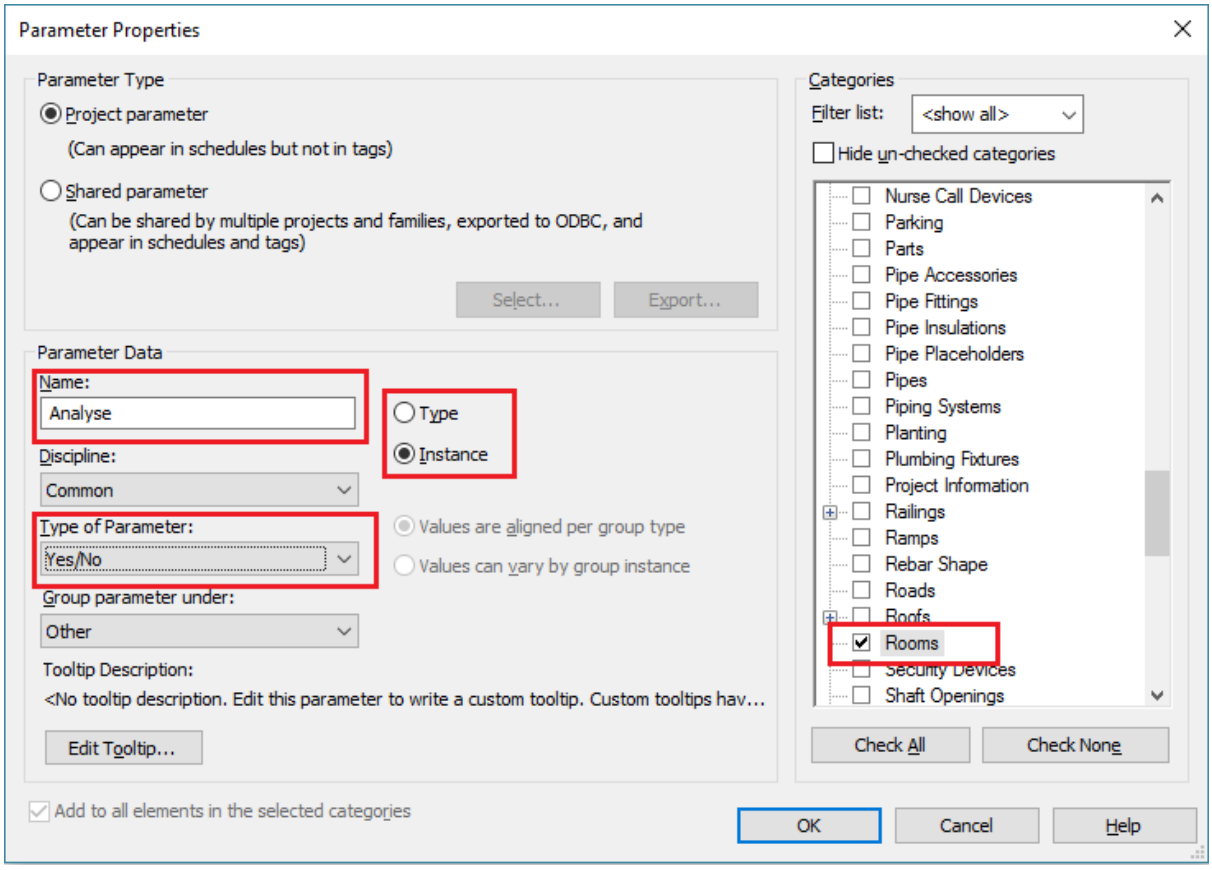


Figure 09-9 Revit Room Parameter Setup

The red markers on the dialog box for adding a new project parameter, shown on the figure, mark the data that needs to be modified. There are two parameters that need to be setup for each room. Both of them need to have Room selected in the *Categories* section on the right, and set as Instance parameters, so each room can have a separate value. The rest of the parameter settings are of no importance for the script and can be set as desired. Individual settings are:

- Name: *Analyse*, Type of Parameter: *Yes/No*
- Name: *Thermal Zone*, Type of Parameter: *Text*

The parameters for each room can be set by either selecting the rooms and changing the parameters, or by creating a room schedule and including the custom parameters. The parameter values can also be visualised in a room plan by having colour-coding, based on the room parameter values.

Only rooms with parameter “Analyse” set to *Yes* are analysed by the script, and related data exported with high level of accuracy. The rest of the rooms are combined in one, for the purpose of increasing the execution time and hardware demands of the script and simplicity of the output file.

XI. Workflow Requirements

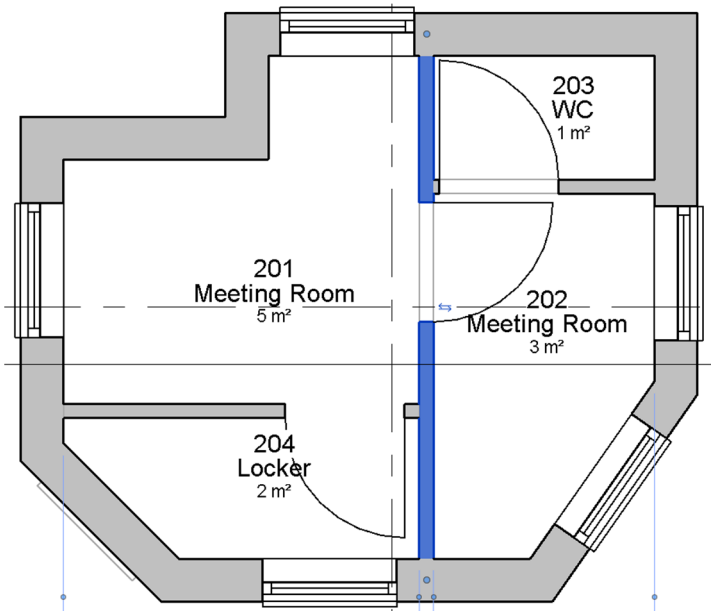
Dynamo is created in such way, that, once a script is opened, after the first execution, any following execution recalculates only the nodes that are affected by any changes made to the script. Typically it accounts for changes in the Revit model too, however, due to the complexity of the script, the changes made in the Revit model after the script has been executed once do not reliably affect the result of the script. That sets the required that, if any changes are made to the Revit model, the script needs to be reopened. Otherwise there is a high risk of inaccurate representation of the changes made in the model.

09.2. Recommendations

This sub-chapter comprises of recommendations to the user for improving his experience with the script. Unlike the requirements set out in the previous sub-chapter, the recommendations are not necessary for correct execution of the script. However they can improve the execution time and respectively the hardware demands of the script, or serve as potential solution to a crash occurring during the script execution.

I. Deleting Unnecessary Internal Walls

The script imports only internal walls that are bounding to at least one room, with the purpose of importing the least amount of needed information from the model. However, walls in Revit typically span multiple rooms, and the script imports the whole wall, not only parts of it that are relevant for the analysis. Refer to Figure 09-10 Wall Spanning Multiple Rooms below.



The selected wall in the figure is imported as a whole, regardless of which of the rooms on the plan are being analysed.

A way to decrease the computation time of the script is to delete the segments of the wall that do not bind any analysed room. That way less information is being imported to the script and processed.

Figure 09-10 Wall Spanning Multiple Rooms

II. Unexpected Performance Loss

Official release of Dynamo is still in the future, and thus there is some unexpected performance loss is occurring – unexpected, but predictable. That needs to be considered, in order to avoid wasting time in using the script. In particular, considering the following situation: after the script has been executed, Dynamo monitors the model for changes. Since the script is very big, any small changes in the model would require very long time to take effect. Thus, to save time in editing the model, it is best practice to close the Dynamo window. However, closing the Dynamo window does not release the used memory from the script (judging from Windows Task Manager), and further changes to the model still take significant amount of time to complete. For this reason, it is recommended that if any changes to the model are required to be done, the whole Revit programme is being closed and opened again. That significantly improves the waiting time.

10. Future Development

While limited by some constraints, the current script is fully functional in respect to the set out requirements, described in detail in Chapter 07. Output Requirements. However, there are many aspects that can be optimised for higher efficiency and thus less computational time and demands on the hardware, as well as additions of new abilities of the script. Both of these topics are described in this chapter. Some of the suggested optimisations and additions have not been implemented due to the lack of skill of the author at the time in which the corresponding part of the script was being created. However, some of the suggestions in this chapter have intentionally not been implemented, due to the balance between script complexity and accuracy.

10.1. Possible Optimisations

Windows and Doors – Size

As described in chapter 06. Development Process and Decisions, currently the windows and doors sizes are extracted from their geometry. That has several drawbacks, such as the computational time required to extract the whole geometry of every element, calculating its boundaries, and in the case of the elements not facing one of the project four cardinal directions (North, East, South, West), projecting the geometry to the hosted wall surface. Alternatively to these steps, the width, height, and location data could be read from the Revit element, however that requires specific names of the Revit family parameters to be defined in the script. While the decision to use the element geometry in the script is taken, whenever possible to work with Revit families, created in a systematic manner, their parameters could be predicted and defined in the script. In such cases, it is suggested that the script is updated to use that logic instead, for the purpose of being “lighter”.

Windows and Doors – Alternative to Projection

As described in chapter 08. Dynamo Script Design, there are two methods for handling window and door elements. First is used whenever the elements face one of the project four cardinal directions (North, East, South, West). It creates a bounding box of the geometry, converts that bounding box to cuboid and intersects that with the wall-host. The second is for elements not facing the project cardinal directions. The node that creates a bounding box of the Revit element, creates the bounding box aligned to the project coordinate system. Therefore the first method cannot be used for elements not aligned to the coordinate system of the project (facing the cardinal directions).

The method used for elements not facing the cardinal directions is as follows: extracts the geometry; from that extracts only the vertices; creates 3D points from them; projects all 3D points to the surface of the wall-host; transforms the 3D point coordinates to relative 2D coordinates on the surface; finds the four outermost points and connects them with lines to form a polygon.

Alternative, however not tested, method to this after extracting the element geometry, using the node BoundingBox.ByGeometryCoordinateSystem. That allows creating a bounding box from a geometry,

aligned to a custom coordinate system. That custom coordinate system can be created to be aligned to the facing direction of the element. After creating that bounding box, the same steps can be used from the first method – creating cuboid from the bounding box and intersecting it with the wall. After successful application of the algorithm it can be compared to the existing algorithm, which projects the geometry on the wall surface, and it can be concluded whether it is more efficient.

Alternatively, another approach could be the same as described above, but without extracting the geometry of each element. Instead extracting three parameters: width, height, and relative location on the wall-host. This allows for manual creation of a polygon on the surface of the wall using the extracted parameters. Which of the two alternatives to be used can be decided by considering the arguments presented in "Windows and Doors – Size" optimisation suggestion above.

Code in Python

As described in sub-chapter 05.3. Software Limitations, Dynamo cannot use looping. Inserting shorts blocks of code with looping and conditional statements, written in Python, can have a significant positive impact on the required resources of the script. Logic that can be improved with Looping is:

- Filtering of windows and doors in two groups: facing cardinal directions, and facing other directions;
- Using looping for the cases where List.Cycle node is used to repeat list values, to match the length (amount of items) of another list. There are many such cases in the script;
- Finding corresponding wall faces (wall-hosts of window and door elements);
- Alleviating the restrictions of having all external walls being connected in one chain and having the same thickness. Potentially Python code can extract the external faces of external walls much more simply, than the logic used in the script;
- Splitting, trimming, and extending internal wall centre-lines;
- Filtering ceiling and floors, and combining them in lists, grouped by level.

Calculation Time Analysis and Optimisation

An efficient way of finding the parts of the script that are main contributors to the computation time, is analysing the amount of time different parts of the script take to execute. Finding out these parts provides high priority targets for optimisation. While not having performed such analysis in controlled conditions, based on the experience developing the script, the author has hypothesised that two factors have significant contribution to the processing demands of the script – communicating with Revit and processing long list of arrays. Thus, it has been a goal throughout development to minimise data requests to the input model and to split very large arrays of data to smaller ones and processing them separately.

Keeping Link Between Construction and Geometry

The present version of the script extracts data from the Revit model elements and creates geometry as per the set requirements, then matches the generated geometry back to the model elements, to establish the link between both. Alternative approach would be to keep the information for the element from the initial import of data, and keep it running parallel to the generated geometry by manipulating the structure of both sets of lists. While theoretically possible, it is unclear whether that approach would yield benefits in terms of computational time. It will not require the matching of geometry and model data, thus omitting the heavy algorithms of intersecting and finding closest geometry (which require communication with the model – assumed to consume substantial computational time), however, managing a second set of lists from the beginning may counterbalance the benefits and prove to be less efficient. That said, managing the second set of lists would theoretically only require finding the changes happening to the first set of list, thus requiring far less resources than them. For this reason it could be hypothesised that it may be more efficient and worth analysing. The reason the first approach was chosen for this thesis is that more importance was assigned to generating the geometry per BSim requirements and less – to the linking of the construction data from the elements, since that could also be done manually, while the geometry generation would save the majority of the time previously spent on manual recreation of the architectural model.

10.2. Possible Additions

Curtain Wall support

As stated in sub-chapter 08.3. Script Constraints, the current version of the script does not support Curtain Walls. Since Curtain Walls are not uncommon in architectural models, this is a reasonable addition to the initial version of the script. As described in the same sub-chapter, there are two types of curtain walls that need to be managed in two different ways – hosted and not. While the hosted curtain walls can be managed the same way window and door elements are managed, not hosted elements need to be managed like normal walls. Thus a filter, distinguishing one from the other is needed in the script. Since that cannot be read from a parameter of the particular curtain wall instance, this filter needs to analyse if the curtain wall is lying inside another wall.

Sloped Ceilings Support

Currently the script can only generate horizontal and vertical surfaces. Sloped ceilings lie on planes, which are neither horizontal, nor vertical, thus a new algorithm needs to be developed for their support. Furthermore, the current algorithm would yield incorrect results in presence of sloped ceilings. The script finds the height of each of the building storeys and creates one surface for each storey, after which the building's solid geometry is split using these surfaces. In case some of the ceilings are sloped, one surface for the whole floor cannot be used, and a combination of horizontal

and sloped surfaces need to be used. Thus this new algorithm will need to manage all types of storey partitions, not only sloped, and has to be approached as such from the beginning of its development.

Sloped Windows Support

As discussed in chapter 08. Dynamo Script Design, currently two methods are used for managing window (and door) elements: one, for elements facing project north, east, south, or west, and another, for elements facing directions in-between the cardinal. Both of these methods assume the windows are vertical and their direction vector lies in the XY plane of the project coordinate system. Sloped windows, and thus windows on any sloped surface, have a direction vector, not meeting these criteria, hence a new method needs to be developed.

Similarly to the method used for windows not facing one of the cardinal directions, a projection method of geometry can be used for these elements. However, sloped windows are usually hosted by roof elements, which are entirely different than wall elements. The current version of the script projects the geometry only to the external walls, and sloped windows would require projection onto roof geometry. And more specifically, representation of roof geometry that is compatible with BSim. Thus an algorithm for that also needs development, as described below.

Roof Geometry Support

Roofs can be simple but also very complex. Dynamo does offer a simple method of importing any roof geometry as a solid object in the script. From this solid, only external surfaces need to be extracted, by means of using Python code or manually in the visual environment. If implementation of support for sloped windows is intended, these surfaces could serve as a projection surfaces for the window elements geometry. Several other factors need consideration to successfully implementing roof geometry in the project:

1. Internal and external walls need to connect to the roof surface, which usually would require their geometries to be extended by certain amount.
2. In case of sloped ceilings, attached to the roof, they need to be included in the construction of the element, and not having a separate surface representation. If there is air between the ceiling and roof construction, that air could be described as a layer in the construction in BSim's database of construction types. Alternatively a pocket of air could be defined as a separate room entirely, and then the bounding elements (ceiling and roof) would need to be defined separately, by their respective surfaces.

Results Feedback

As an addition to the present script, an algorithm can be developed to import the results of the BSim simulations to the Revit Model. Dynamo does offer ways to write and modify Revit views. A good approach is creating parameters to the Rooms that would be filled with the simulation data and presented in a schedule and as colour-coded plan views.

The feedback script can be part of the original script, however it would be more efficient to have it as a separate script. Being part of the original script would require checking for presence of simulation results first, and, if present, execute feedback to the model; if not, perform the original script. Computational time however can be expected to be longer using only one script, due to the limitation of Dynamo to calculate both True and False results of an IF condition (described in sub-chapter 05.3. Software Limitations).

Split Script in Separate Files

As suggested in the above paragraph, the script could be separated in different files. Doing so would require less hardware requirements for the execution and less computational time. Also, it would be much less likely to result in a crash. Downside to splitting the algorithm would be that the whole process would not require one action from the user and result in a ready output file, but instead, the user has to monitor the process and interact.

The script can be split in different parts, each of which writes to the Output file. That way the next part of the script could read the processed data, if needed, and would start fresh. Splitting the script in separate files could, however make the development process more difficult, since different files might need to be worked on simultaneously. Though, smaller files make navigation and interaction with the interface much smoother. Therefore whether it will make development more difficult or less difficult, depends on what part of the whole script needs changing – if it is only specific part of the script that needs work, then working with separate files would be much easier.

Multiple thickness storey partitions

Currently the script can handle only one storey partition thickness for one storey partition in the building. The reason is because the requirements for the exported geometry state that the surface, representing the storey partition, needs to be in the middle of the thickness of the storey partition. This means that different thicknesses of storey partition require separate surfaces at different elevations. The current algorithm, however, uses the storey partition of each floor to split the building solid to separate objects for each storey. In order for this operation to succeed, each storey partition needs to be one combined object. So the surfaces with different elevations would need to be connected with a vertical surface. The creation of this surface needs to be managed in the new algorithm.

Balconies/Terraces

The script requires all external walls to be connected in one chain. That excludes the possibility of analysing a building with balconies or terraces. Thus, implementing support for such, requires management of external walls to be different. As suggested in sub-chapter 10.1. Possible Optimisations, Python code can be used to extract the outer faces of external walls. Textual coding allows the flexibility needed to manage structures of balconies and terraces.

Furthermore, changes in the method used to create the volumes of rooms, and consecutively – faces, edges, and vertices, needs to be modified or changed. Currently it creates a solid of the whole building by taking the building's base and top edges and connecting them to form a solid. That would cause a wall to be running through any balconies of the building's facades. Additionally, a top-floor terrace, would cause an error in the algorithm, since the used function (commonly known as "Loft" in modelling software) requires two planar profiles to be connected. These profiles are represented by the bottom and top edges of the building. The presence of a top-floor terrace would cause the top edges not to lie on the same plane, and the Loft function would fail.

11. Conclusion

In today's world of constant increasing demands for energy performance, a great focus is placed on quality energy analysis, early in the process. That can only be achieved with good communication between modelling software and analysis tools. This thesis aims at bridging the gap between Revit and BSim – two very widely used software tools for modelling and energy analysis in Denmark. Together, the author of this thesis and a group of students in Indoor Environmental and Energy Engineering created a communication flow for information transfer from Revit model to BSim.

This objective of this thesis is to create and describe the first part of the information flow, namely: *How to export information from a Revit building model to Excel file, formatted in a BSim compatible schema, to allow automation of energy simulations.* The present paper describes the process and details involved in the creation of the script that performs this operation.

The sub-questions, set out have been investigated, and have guided the process of creation of the script and overall flow. At several points in time some parts of the information processing have been moved from one part of the flow to another. The decisions for these arrangements have been made on the basis of efficiency of the overall process.

In conclusion to the presented work, it can be stated that the final product of this thesis fully satisfies the initial objective to export the information form the model to Excel file, formatted in the required structure. The script is created in a way, allowing possibilities for future development, such as feeding back the results from the simulation, support for more complex geometries, such as balconies and attics, and even optimisation suggestions, such as suggesting new shape or number of windows.

With the future of Dynamo, many new possibilities could be opened for this script. Additional features could make it more useful and even incorporate the full information process in the script, exporting the data directly to XML, ready for simulations. Until then, the current version can save significant amount of manual work. Energy engineers can spend their time on what is important and make higher quality analyses for less time.

12. Bibliography

Autodesk, 2016. *BIM Overview.* [Online]
Available at: http://www.autodesk.com/solutions/bim/overview
[Accessed Jan 2016].

Dynamo Primer, 2015. *Dynamo Primer.* [Online]
Available at: http://dynamoprimer.com/
[Accessed 2015].

Dynamo Primer, 2015. *What is Visual Programming.* [Online]
Available at: http://dynamoprimer.com/01_Introduction/1-1_what_is_visual_programming.html
[Accessed 2016].

Epstein, E., 2012. *Implementing Successful Building Information Modeling.* s.l.:Artech House.

Gillen, D., 2015. *Are Computers Bad for Architecture?.* [Online]
Available at: http://www.archdaily.com/618422/are-computers-bad-for-architecture
[Accessed 2016].

Graphisoft, 2016. *BIM as a Platform for Communication.* [Online]
Available at: http://helpcenter.graphisoft.com/guides/archicad-17-guides/graphisoft-collaboration-guide/interoperability/bim-as-a-platform-for-communication/
[Accessed January 2016].

Hudson, M., 2014. *Dynamo Hero: Using Revit Scripting Tools to Optimize Real-World Projects.* [Online]
Available at: http://au.autodesk.com/au-online/classes-on-demand/class-catalog/2014/revit-for-architects/ab6495#chapter=-1
[Accessed 2015].

Interaction Design Foundation, 2015. *Contextual Design.* [Online]
Available at: https://www.interaction-design.org/literature/book/the-encyclopedia-of-human-computer-interaction-2nd-ed/contextual-design
[Accessed 2016].

Keough, I., 2014. *Explore the Possibilities with Computational BIM.* [Online]
Available at: http://au.autodesk.com/au-online/classes-on-demand/class-catalog/2014/revit-for-architects/ab6542#chapter=6
[Accessed Nov 2015].

13. Table of Figures