Comparative evaluation of parametric design systems for teaching design computation

Robert Aish and Sean Hanna

The Bartlett School of Architecture, University College London, Gower Street, London WC1E 6BT, UK

Note: this paper has subsequently being published in Design Studies as
Aish, R., & Hanna, S., Comparative evaluation of parametric design systems for teaching design computation, Design Studies (2017), http://dx.doi.org/10.1016/j.destud.2017.05.002

Abstract: Three parametric design systems were tested by the authors to assess their suitability for undergraduate teaching. We used criteria taken from the 'cognitive dimensions' literature and an exercise of typical geometric operations in ascending order of complexity. For each system the cognitive barriers associated with the sequence of operations were plotted to create a 'learning curve'. Different parametric systems presented distinctly different learning curves. The test exercise had to be completed in its entirety to assess the potential challenges which students with different educational levels, skills and abilities might encounter, so a single expert user conducted the tests. This research is intended to develop methods, both design exercises and evaluative criteria that could be used in future empirical studies.

Keywords: architectural design, design education, human-computer interaction, parametric design, evaluation

Digital media and working methods are considered to have a pronounced influence on design thinking (Oxman, 2008), therefore understanding the way parametric systems support parametric design thinking is of critical importance for both students and educators.

Students will develop their parametric design ability through the use of these applications. Indeed, the way the selected system presents its functionality may well be taken by students as the definition of parametric design. Therefore the influence of parametric design systems on the students and responsibility which goes with this means that it is essential that the available applications are systematically evaluated. Often the choice of parametric system is influenced by other extraneous factors such as the 'platform' or the application software associated with the parametric system, where the platform or application might have already been selected by the user's institution. Similarly students with partial knowledge of parametric design may be influenced to select tools with which they are already familiar even if these systems may not be best suited to later learning stages. In this study we have deliberately excluded these extraneous factors and focussed exclusively on a systematic evaluation of the different parametric design systems.

Learning rates may differ between students. Different parametric software may be more or less suited to different parametric modelling tasks and to different students. All these differences interact. While it may be possible to observe this type of parametric design learning informally in a classroom setting, it will be challenging to design controlled empirical tests for students learning to use parametric design software for the first time, sufficient to provide a statistically valid description of the learning progress. Additionally, there are the considerable challenges inherent in coordinating sufficient resources and appropriate student volunteers to make investigation of nontrivial skill learning practical. Further, such observations may not directly explain the underlying reasons for ease or difficulty in learning, which are of interest both to software designers and educators. For these reasons we propose an alternative approach, which establishes a set of relevant criteria by which the software can be evaluated, aiming both to limit the subjectivity of different users and to correspond to particular cognitive factors which explain potential learning challenges. This is intended to equip a single user, often an expert, most probably with some existing bias, to make this evaluation with sufficient objectivity. These two approaches, expert evaluation and empirical studies, can ideally inform one another, but at least the first should be explored before the second and it is the first which is the topic of this paper.

The purpose of this evaluation is to explore the cognitive issues involved with parametric design software which would be experienced by a novice user rather than the subjective experience of students with particular backgrounds or levels of skill. This evaluation involved constructing the same abstract parametric geometry model with the different systems and evaluating the model building process with nine criteria developed from the 'cognitive dimensions' literature. The design of the test exercise, the development of the evaluative criteria, the model building activity with the different systems and the review of the different model building processes with the evaluative criteria has been done by the authors. The authors have a background in developing, using and teaching parametric design and design computation. They also have similar levels of unfamiliarity with the current interfaces and functionality of the three systems tested. Experts associated with the three software developers were consulted by the authors to ensure equal knowledge for each system.

In many applications of parametric design for example to architecture, parametric modelling involve operations which create and use collections. Therefore how collections are presented by a parametric design application to the designer is crucially important. The model building exercise involves:

1. Creating of a 2D array of points
2. Creating a surface using the 2D array of points
3. Creating a set of curves through the points
4. Creating a set of curves through the transpose of the array of points
5. Creating a single curve by making an arbitrary selection of points from the 2D array

While the test model is quite abstract and only exercises a small subset of the functionality of the parametric design systems, it is difficult to see how the more advanced functional of a parametric design system can be harnessed without the user first becoming proficient with this functionality. Therefore how this functionality is supported is a convenient indicator of the overall suitability of the different parametric systems.

1 **Evaluative criteria**

The commonly used term 'learning curves' describes cognitive challenges over the duration of a learning process. This concept of the 'gentle slope' was first introduce by MacLean, Carter, Lovstrand, and Moran (1990) and then further developed by Myer (2002). Both suggested that in teaching computer science to novices, programming languages and tools should be selected which presented a 'gentle slope' of concepts of gradually increasing complexity. It is suggested that this approach is also valid for teaching parametric design. [Figure 1].

While all these learning curves may be idealisations they serve as a way to think about the overall educational challenge, that is: how can parametric and computational concepts be simplified and

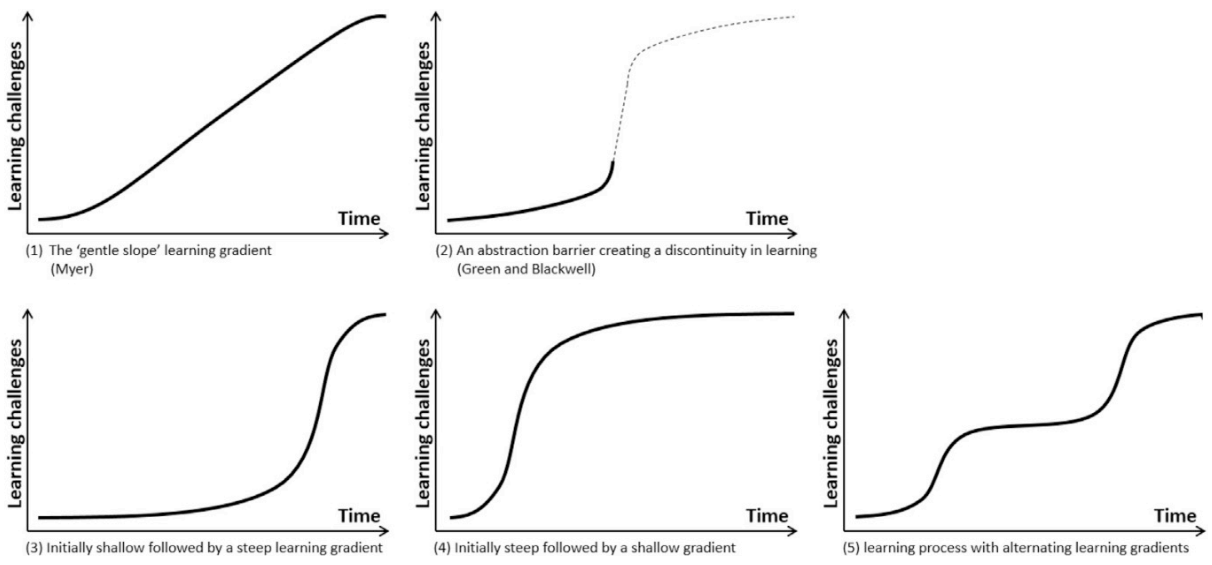


Figure 1: Five possible learning curves

made intuitive for the novice user, while still providing a conceptually valid educational foundation for the acquisition of parametric and computational design fundamentals should the novice user wish to proceed to more advanced computational design. Myer suggested the gentle slope approach may have considerable advantages over other learning curves.

The following evaluative criteria are features of the system whose presence or absence may create cognitive or practical barriers for novice users, changing the slope of the curve. They are based on Green and Blackwell's research into the 'Cognitive Dimensions of Information Artefacts' (1998) and on the capabilities of modelling and programming languages which are generally considered to be fundamental for their effective use.

1.1 Cognitive dimensions

'Cognitive dimensions' describe different aspects of a computer system which allows or obliges a user to think and act during the use of a system. There are complex interconnections between cognitive dimensions and the capabilities of the system. We propose a number of additional dimensions which combine or extend the original Cognitive Dimensions research.

1.1.1 Abstraction Barrier

Green and Blackwell (1998) offer the following definition: 'The abstraction barrier is determined by the minimum number of new abstractions that must be mastered before using the system.' Here the emphasis is on the additional ideas or ways of working whose relevance is not currently appreciated by the user, but which have to be mastered before the functionality of interested can be accesses. The abstraction becomes a barrier if the user is oblige to understand it before the abstraction's value to the user can be appreciated. There may be valid arguments that some abstraction barriers are confronted early in the learning process in order to minimising disruption over the complete process.

Ideally new abstractions should be discovered and applied in the order in which they appear to be relevant to the user's interest and where the delta in understanding between a known abstraction and an unknown abstraction is within the user's ability to comprehend. In an educational context the aim is not to avoid abstractions but to avoid abstractions becoming a barrier.

1.1.2 Semantic interference

This is an additional cognitive dimension which builds on the ‘clear names’ design pattern proposed by Woodbury (2010) as a critical ‘element of parametric design’, as follows: ‘Good names are clear; they convey what you intend. They are meaningful; usually this means they relate to either the form or function of a design. They are as short as they need to be (and no shorter)’. If, as Woodbury suggests, that users of parametric design applications should be extoled to use ‘clear names’ then the design of these parametric design applications should also use ‘clear names’.

Semantic interference occurs when there is a mismatch between a term and the meaning to be conveyed. A parametric design application may use a term with a particular domain-specific or vernacular meaning which the novice user may be familiar with and therefore the novice user might accept the term and its meaning as definitive. In an educational context this semantics may interfere with the objectives of the instructor who may want to use the established conceptually defined terminology with precise meaning.

Semantic interference may also occur when the same terminology has multiple meanings in the same or different parts of the system or when multiple terms are used for the same concept or feature of the system. The origins of some specialised terminologies are essentially metaphors which have become established. End user computing systems often introduce un-established metaphors (Barr, 2003). We can describe a ‘metaphor trap’ as a special form of semantic interference where an inappropriate metaphor is used by the application to describe an underlying computing concept and the natural language meaning associated with the term does not describe the precise meaning or generality of the concept, so that the novice user is unaware of the full scope of the functionality being referred to by the metaphor.

Consider the following progression:

General computing:

I want to transpose the array of numbers
[precise terminology describing an abstract operation being applied to abstract data]

Domain specific computing [retaining computing abstractions]:

I want to transpose the array of beams
[precise terminology describing an abstract operation applied to domain specific term: beams]

An example of metaphor based domain specific computing [where arrays are implemented as a tree data structure and operation on trees use vernacular metaphors, such as ‘flip’]:

I want to flip the tree of beams
[the vernacular metaphor ‘flip tree’ combined with domain specific term ‘beams’ results in a hybrid language which is neither a valid computational expression nor understood as a valid in the application domain. Not only is this confusing but it may also mask the functionality of the abstract operation]

1.2 Properties of parametric design systems

1.2.1 Consistency between representations

Most parametric design system offer multiple representations including a visual graph based data flow representation, geometric representation and sometimes a text based program representation. However, to help the user build a unified internal mental model it is important that there is consistency between these different representations. Specifically for the exercise considered where, is the apparent geometric organisation [a 2D array of points] reflected in the logical structuring of the data in the graph based visual programming environment? The importance of the consistency of mapping between representations specifically with parametric design applications has previously been discussed by Aish and Woodbury (2005) and Harding, Joyce, Shepherd, and Williams (2012).

1.2.2 Discoverability

This describes how the functionality of the system is presented and documented so that it can be discovered by the user unaided. If the educational intent is to teach student users about the underlying concepts in computation and geometry, then it might be appropriate to use a classification system based on some clear conceptual basis. For example, the geometry functionality may be classified as an object-oriented class hierarchy using the 'dimensionality' of different geometry types [0D for points, 1D for curves, 2D for surfaces and 3D for solids]. The concept of 'type' and class hierarchy (from general to specific) allows the novice user to understand what functionality is common and what functionality is unique to different types of geometry or other domain specific aspects of the application. However, without any overall logic, the novice user is forced to consume additional cognitive resources to directly learn the idiosyncrasies of the menu structure. This represents an investment on the part of the user not in generally transferrable knowledge (of the logical classification of geometry and other parametric and computational concepts) but in the specifics of a particular parametric design system. This then creates two disincentives for the user to move to a different parametric design system: abandoning the investment in one system and investing in learning another system.

1.2.3 Flexibility

One of the key issues in the design of a computer system is the flexibility it offers the user. In the original Cognitive Dimensions research, Green and Blackwell (1998) define three different dimensions which to describe the consequence for the user to flexibility:

First, does the system requiring the user to perform actions (and therefore to think about those actions) in an inappropriate order? [Premature Commitment].

Second, does the system allow the user to make tentative decisions which can be subsequently changed? [Provisionality].

Third, how difficult is it to make these subsequently changes? [Viscosity].

Overall parametric design applications based on visual data flow programming data flow are extremely flexible compared to modelling application based on direct manipulation. While direct manipulation systems offer high levels of flexibility during initial sketch, changing these models is often extremely arduous and often require all or substantial parts of the model to be deleted and for the user ‘to start over’. The principle advantage of parametric design systems most frequently referred to by users is the capability to revisit and change previous modelling operations and the consequences of these changes are automatically propagated through the model, without the user having to delete and to manually remodelling.

There are also criticisms that once built, complex parametric models are difficult to change and this inhibits design exploration (Davis, Burry, & Burry, 2011). These comments reinforce earlier conclusions from Burnett et al. (1995) that there are scaling and usability issues with visual programming. For example a node in a data flow graph combines: the name, the ‘type’, and the calculation method used to create its value. While type and method may be interdependent, in a regular text based programming language the user is free to change any one of these aspects independently. In a node based system these options are often not available forcing the user to create a new node, then to move the connections from the old node to the new node and finally to delete the old node. A clear example of ‘viscosity’.

1.2.4 Side effects

This is an aspect of the functionality of the system were some minimal change by the user (for example, to the input data) has a wide ranging and unexpected effect on the behaviour of the model or program (for example, on the output). This is a slightly different and extended interpretation to that used in computer science.

1.2.5 Work arounds

This is modification or additional operations which the user is obliged to add to the model or program, which from the user’s perspective is neither part of the design intent and nor appears to be logically required. Work arounds may also be required to circumvent a previous side effect. Work arounds may require the user to understand new abstractions which are unrelated to the user’s current interest and therefore work arounds are likely to introduce abstraction barriers. Work arounds are considered fragile, because they are developed in response to some limitations in the original system. To function correctly the work around is now dependent on that limitation. If the system is corrected then the work around may no longer be required and its continued presence may give the wrong result, which the user may be potentially unaware of.

1.2.6 Convoluted workflow

Consider the situation where the required functionality is supported by the application. The functionality is documented and no new abstractions are required to be learnt. Nevertheless to complete the task the user has to adopt such a complex workflow that the whole process appears to be counterproductive and discouraging to the point where the task [and hence the application] might be abandoned.

1.2.7 Liveness

Liveness is a concept borrowed from the performing arts to describe the spontaneity and responsiveness of a performance. Liveness is also used as a term to compare live and recorded performance and the role of life performance (Auslander, 2008).

In user oriented computing applications ‘Liveness’ is used to describe the system’s performance and support for interactivity. In discussing the concept of ‘Liveness’, perhaps the closest comparison to parametric design systems [used for architectural design] are Digital Music systems [used for composition and performance]. In this context, Liveness has been described as ‘a quality of the design experience that indicates how easy it is for user to get an impression of the end product during intermediate stages of design’ (Nash & Blackwell, 2014).

In the case of ‘user oriented’ computing applications such as parametric design systems (as examples of interactive computing) ‘Liveness’ is associated with aspects of program performance and user interaction, such as:

- Modeless interaction - Is the user aware (or not aware) of distinct modes of operation? The user may not be aware of such modes if, for example, the system’s default mode is ‘continuous execution’.
- Latency - How quickly does a program respond to user events?
- Dynamics - Can the user control the ‘quality’ of program dynamics, for example by determining the trade-offs between the complexity and completeness of the model when this is being recomputed in real-time.
- Directness of interactions - How does the user interact with the system, for example, indirectly by using a keyboard to change the numeric value of input parameters, via ancillary analogue interactions devices such as sliders or by direct manipulation of the geometry within the user’s model?

2 **Test results**

The exercises require the application of different geometric and logical concepts. The reader is encouraged to imagine how the instructor can maintain a coherent narrative explaining the various parametric and computational concepts while at the same time explaining the functionality and terminology used by the different systems to implement these concepts.

2.1 Generative Components [version 08.11.09.288]

With GenerativeComponents, the complete exercise was easily and directly achieved in 10 nodes [Figure 2] without any unusual functionality, terminology or workarounds. The only problem was a ‘discoverability’ issue, associated with transposing the point array. This is because there is no

‘Transpose’ node available in the visual programming environment. Instead the Transpose of the point array [Point01] is encapsulated in an expression within the input port of the curve01 [Figure 3].

- Abstraction Barrier: Generally no abstraction barriers, but challenging in one case where a script expression had to be used (for the Transpose function).While this ability to use script expressions may be extremely useful for an experienced user, it may not be suitable for a novice user. This is because it requires the novice user to learn about script notation very early in the use of the system. Ideally, the whole set of exercises should be completed using Visual programming nodes and then only subsequently should scripting be offered as a more advanced option.
- Semantic Interference [including metaphor trap]: Good, there were no problems with terminology.

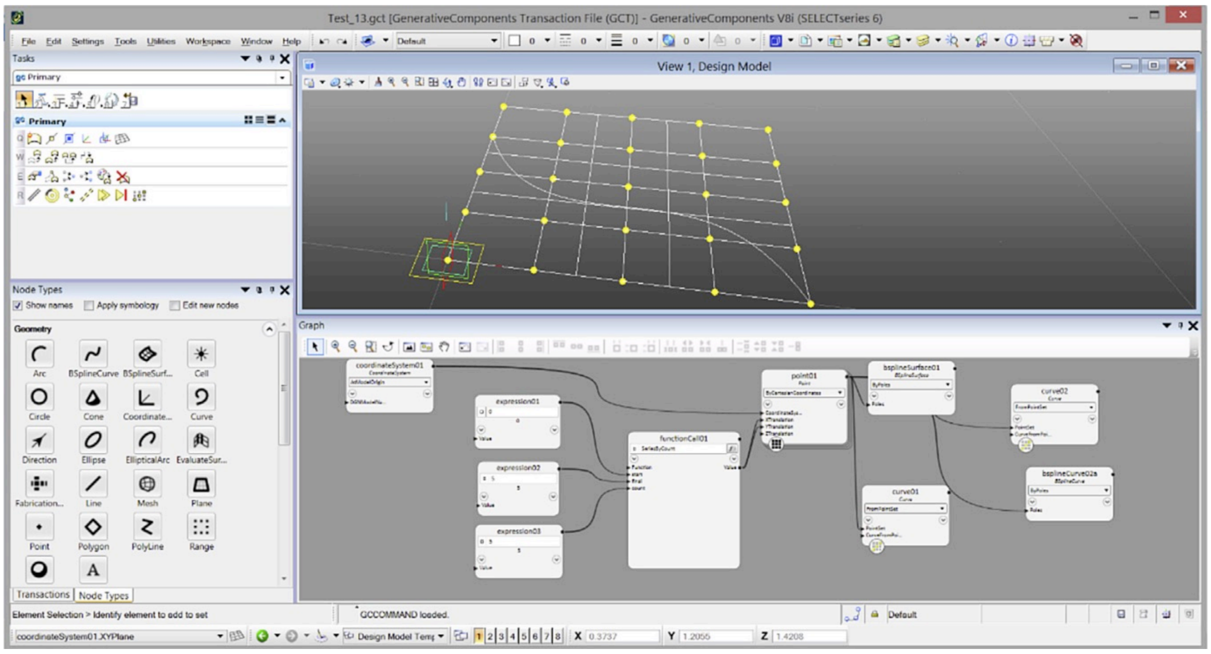
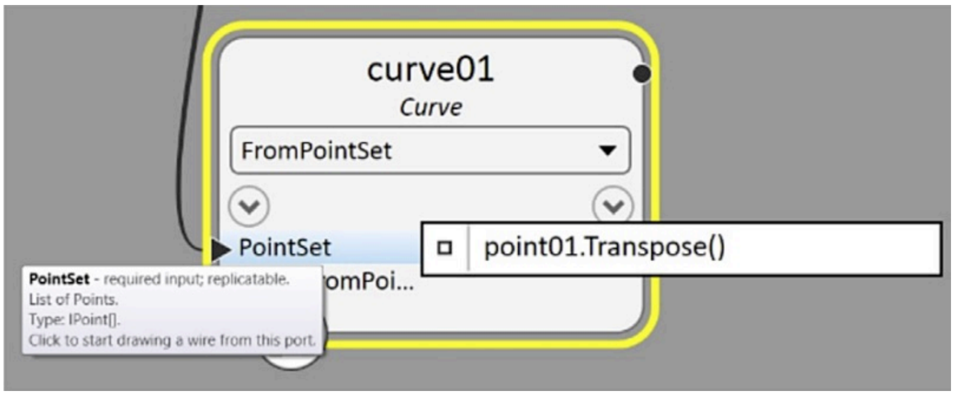


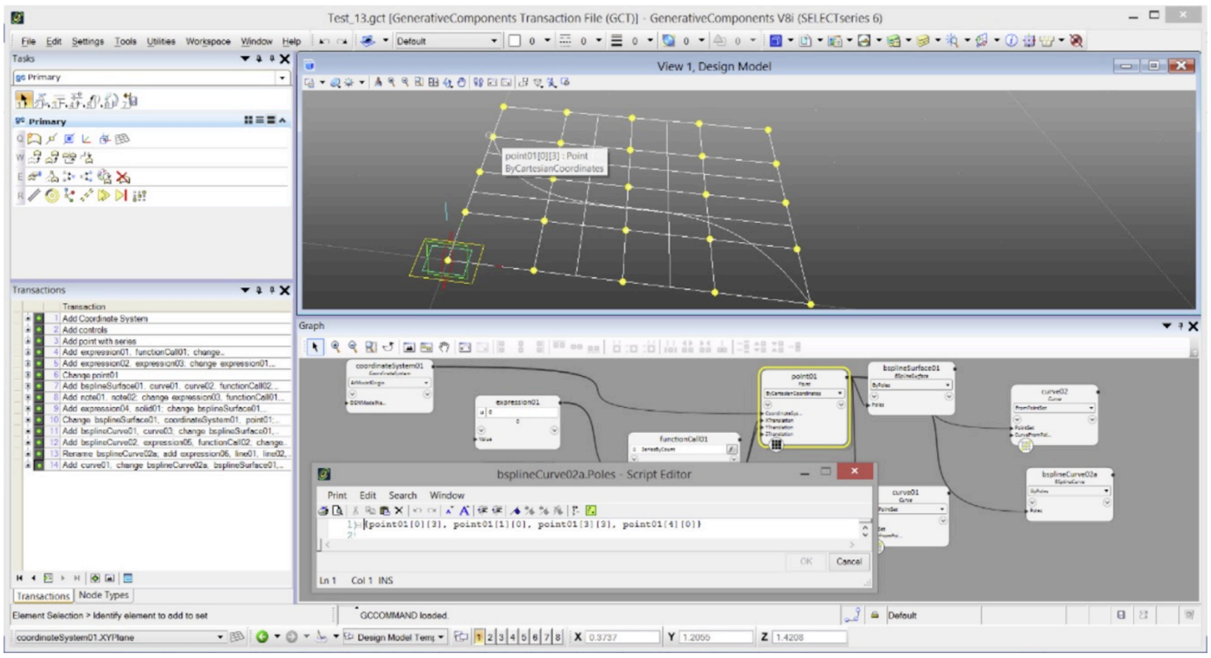
Figure 2: The completed exercise in GenerativeComponents



PointSet - required input; replicatable.
List of Points.
Type: IPoint[].
Click to start drawing a wire from this port.

Figure 3: Transposing the point array using a scripted expression in the 'PointSet' input port of the Curve node

- Consistency between representations: Good, including cross highlighting between generated geometry and graph node
- Discoverability [of functionality, including logical ordering]: Good, the menu is clear and easily navigable [Figure 2]. There is an attempt to provide the user with an 'object-oriented' description of the functionality of the geometry library. When the cursor hovers over each of the 'top level' icons representing different geometry types, a 'fly-over' label appears which documents the different interfaces implemented. However, there is no self-discoverable documentation to describe the methods which each of the interfaces implements, therefore the potential pedagogic advantage of explaining the functionality of the geometry library in 'object-oriented' terms is not completely realised. In addition there were challenging aspects, for example the 'Transpose' function [the use of which was an essential aspect of the tasks] is not available as a graph node. It is available within the script editor. However this is not documented and there is no 'auto- completion' in the script editor to offer this and other methods to the user. So there is no way for the novice user to discover this important functionality.


Test_13.gct [GenerativeComponents Transaction File (GCT)] - GenerativeComponents V8i (SELECTseries 6)

View 1, Design Model

point01[0][3] : Point
ByCartesianCoordinates

Transactions

	Transaction
1	Add Coordinate System
2	Add controls
3	Add point with series
4	Add expression01, functionCall01; change..
5	Add expression02, expression03; change expression01...
6	Change point01
7	Add bsplineSurface01, curve01, curve02, functionCall02...
8	Add note01, note02; change expression03, functionCall01...
9	Add expression04, solid01; change bsplineSurface01...
10	Change bsplineSurface01, coordinateSystem01, point01;...
11	Add bsplineCurve01, curve03; change bsplineSurface01,...
12	Add bsplineCurve02, expression05, functionCall02; change..
13	Rename bsplineCurve02a; add expression06, line01, line02,...
14	Add curve01; change bsplineCurve02a, bsplineSurface01,...

bsplineCurve02a.Poles - Script Editor

```
{point01[0][3], point01[1][0], point01[3][3], point01[4][0]}
```

Ln 1 Col 1 INS

GCCOMMAND loaded.

Element Selection > Identify element to add to set

coordinateSystem01.XYPlane

X 0.3737 Y 1.2055 Z 1.4208

Figure 4: The 'S' curve was create by selecting points from the point array directly in the geometry window. Note the identity of the point selected is displayed as part of the flyover label with the indexing into the 2D point array. The corresponding graph node is highlighted. The collection expression code fragment [in the script editor] is being built for the user from the interaction with the geometry model. As the user selects a further point so the collection expression is automatically extended with a new member

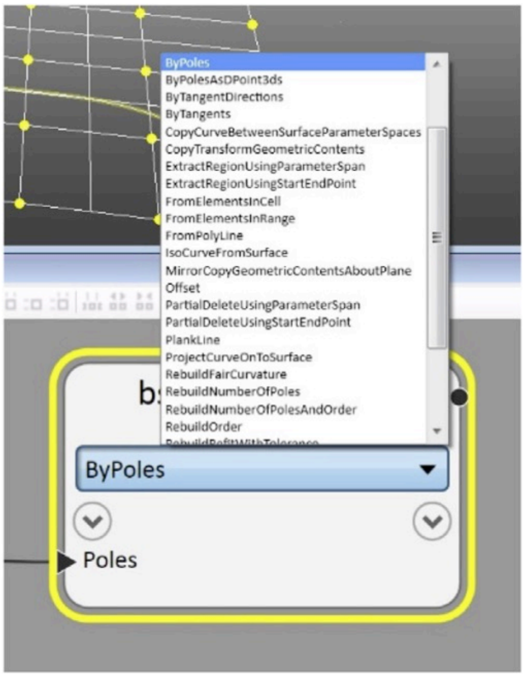

ByPoles
ByPolesAsDPoint3ds
ByTangentDirections
ByTangents
CopyCurveBetweenSurfaceParameterSpaces
CopyTransformGeometricContents
ExtractRegionUsingParameterSpan
ExtractRegionUsingStartEndPoint
FromElementsInCell
FromElementsInRange
FromPolyLine
IsoCurveFromSurface
MirrorCopyGeometricContentsAboutPlane
Offset
PartialDeleteUsingParameterSpan
PartialDeleteUsingStartEndPoint
PlankLine
ProjectCurveOnToSurface
RebuildFairCurvature
RebuildNumberOfPoles
RebuildNumberOfPolesAndOrder
RebuildOrder
ByPoles
Poles

Figure 5: Once a specific type of graph node has been created, the method [or 'technique'] used to construct the geometry can be changed by selecting from a list of available methods. This allows the user to experiment with different methods, without having to deleted and recreate the node and its connections. This adds considerably to the ease of changing the model

- Flexibility [of editing, minimising reworking]: Good, the node method can be changed without having to delete and recreate the node [Figure 5]. Also the design of the graph node allows the user to reference the XYZ properties of the point node by adding optional output ports rather than by creating additional nodes [Figure 6]. No new node had to be added to the graph to expose these properties.
- Side effects: none detected
- Work arounds: none detected
- Convoluted workflow: none detected. The task could be completed with 10 nodes.
- Liveness [including direct interaction with geometry]: Good: There is the ability to directly select geometry (for example, to build a new collection by selecting geometry from an existing collection) [Figure 4]. There is also the ability to directly interact with geometry in the model and automatically update the graph. (For example: moving points along the axes of coordinate systems, along curves, or on planes or on surfaces).

2.2 Grasshopper [version 0.9.0076]

While it was possible to complete the tasks in Grasshopper, the exercise was made more difficult by the use of 'data trees' as the principle implementation of collections. This required understanding the rather unusual functionality and terminology involved. It became apparent that whichever way the points were created, the novice user would have to learn additional 'workarounds' to complete the tasks.

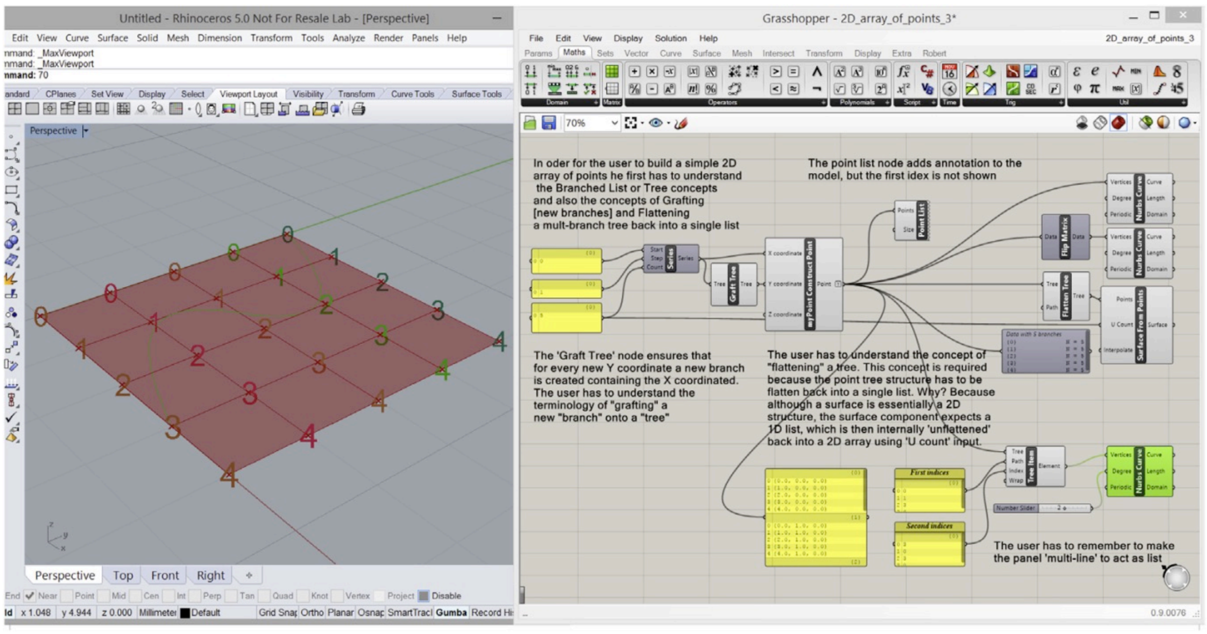
One approach is to create a tree [or 2D collection] of points [Figure 7]. This is most probably the logical approach which many users would take since the 2D structure of the data corresponds to the 2D spatial layout of the point geometry.

However to create the surface, the user has to add an additional operation to 'flatten' the 2D collection into a 1D list because the surface creation node requires a 1D list as input. Effectively the user is having to provide this additional 'flatten' operation (or workaround) to undo the hard coded 'unflatten' operation which is built into the surface creation node. Working out what is happening inside the surface node and then how to circumvent its built-in 'unflatten' functionality was challenging. An alternative approach is to create a list [or 1D collection] of points [Figure 8]. This can be directly input into the surface creation node, but the novice user now has to use the 'Partition List' workaround to restructure the points back into a 2D collection in order to create the curves.

- Abstraction Barrier: Data trees are an important 'abstract data type' which supports a clearly defined set operations. It can also be used to implement other collection types such as lists and arrays. The issue of data tree is discussed at length by Rutten including a discussion about other data structures found in high level programming languages and there uses.

  Effectively the use of data tree in Grasshopper is an implementation convenience which is exposed as an end-user metaphor. However, from an end user's perspective [both instructor and student] the terminology of 'trees' may not be a useful metaphor to explain or harness the concept of arrays.
  The question remains whether the intention behind the use of data trees in Grasshopper is driven by implementation considerations or whether the intentions are pedagogic (Rutten, 2015).

  We came to the conclusion that too many cognitive cycles would have to be spent explaining the relationship between the metaphor [tree, branches, etc.] and the underlying abstractions [collections, arrays] and in particular having to explain that the underlying abstractions have characteristics that go beyond the metaphor. If it is anticipated that the students will progress beyond visual programming to scripting and programming then they will have to learn about arrays and indexing anyway, therefore it might be argued that these concepts should be presented 'up front' to the students. This is not to dispute that data trees is a powerful abstraction which is incredibly valuable when correctly applied to data which is genuinely 'tree-like'. But in this context data trees becomes an unnecessary abstraction barrier and a metaphor trap and it presents the user with a representation which is inconsistent with the user's conceptualisation of the data.


In oder for the user to build a simple 2D array of points he first has to understand the Branched List or Tree concepts and also the concepts of Grafting [new branches] and Flattening a mult-branch tree back into a single list

The point list node adds annotation to the model, but the first idex is not shown

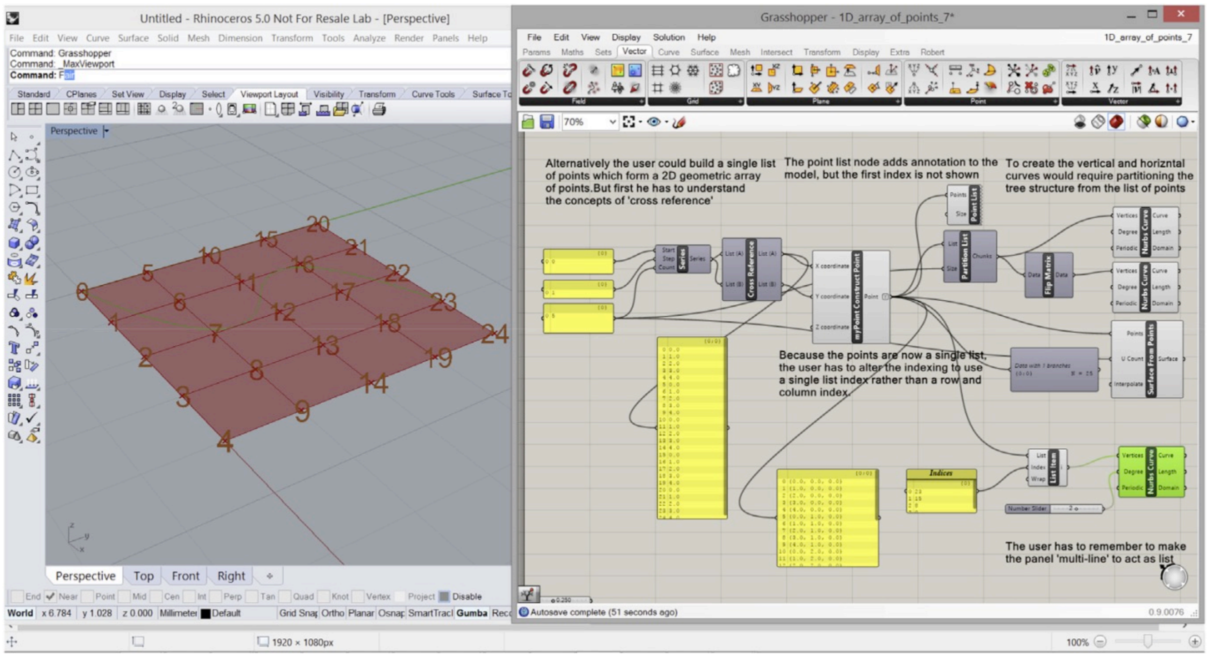
The 'Graft Tree' node ensures that for every new Y coordinate a new branch is created containing the X coordinated. The user has to understand the terminology of "grafting" a new "branch" onto a "tree"

The user has to understand the concept of "flattening" a tree. This concept is required because the point tree structure has to be flatten back into a single list. Why? Because although a surface is essentially a 2D structure, the surface component expects a 1D list, which is then internally 'unflattened' back into a 2D array using 'U count' input.

The user has to remember to make the panel 'multi-line' to act as list

Untitled - Rhinoceros 5.0 Not For Resale Lab - [Perspective]
Grasshopper - 2D_array_of_points_3*
Perspective
Top
Front
Right
First indices
Second indices

Figure 7: Creating the points as a tree, effectively a 2D array


Alternatively the user could build a single list of points which form a 2D geometric array of points.But first he has to understand the concepts of 'cross reference'

The point list node adds annotation to the model, but the first index is not shown

To create the vertical and horizntal curves would require partitioning the tree structure from the list of points

Because the points are now a single list, the user has to alter the indexing to use a single list index rather than a row and column index.

The user has to remember to make the panel 'multi-line' to act as list

Untitled - Rhinoceros 5.0 Not For Resale Lab - [Perspective]
Grasshopper - 1D_array_of_points_7*
Perspective
Top
Front
Right
Indices
Autosave complete (51 seconds ago)

Figure 8: Creating the points as a list, effectively a 1D array

- Semantic Interference [including metaphor trap]: Challenging: The most important issue when valuating Grasshopper is the use of data trees together with the related terminology used to describe operations on data trees, such as Branching, Grafting, Flattening, Path, Flip, etc. These terms are used to construct an uncomfortable vernacular metaphor which masks the underlying array concepts. Thinking about the needs of the novice user, it might have been preferable to directly expose the concept and terminology of the 'array' or even a consistent 'list of lists' concept.

  Other terminology in Grasshopper appeared misaligned with the underlying concepts. For example, the term 'cross referencing' in ordinary usage applies to an instance within a document which refers to related information elsewhere in the same document. However, in Grasshopper it is used to imply a form of combinatorial expansion, where a new output list A is created by copying the original input list A once for each members in the original input list B AND a new output list B is created by copying the original input list B once for the every members in the original input list A. The term 'Cross reference' does not seem to be an appropriate description of the underlying process.

  Overall, the terminology is highly vernacular and metaphoric. The icons take the metaphor of the 'tree' to near visual excess. The problem is that the apparent simplicity of metaphor masks some complex functionality and therefore the value of the metaphor as an 'intuitive lead-in' to this functionality for the novice user may be lost.

- Consistency between representations: Challenging: An additional problem with the list approach is that the data structure (a 1D list) does not match the geometric structure (2D configuration of points) [Figure 8]. The user has the additional cognitive load of understanding the different logical and spatial representations and translating between the two to understand the correspondence between the data and geometry. Because the points are now a single list, the user has to alter the indexing to use a single index rather than the row and column indices used with the 2D collection, when selecting points with which to create the 'S' curve.

- Discoverability [of functionality, including logical ordering]: The menus are reasonably well-structured however there are some awkward classifications for example Field, Grid, Plane, Point, and Vector can only be found under Vector tab.

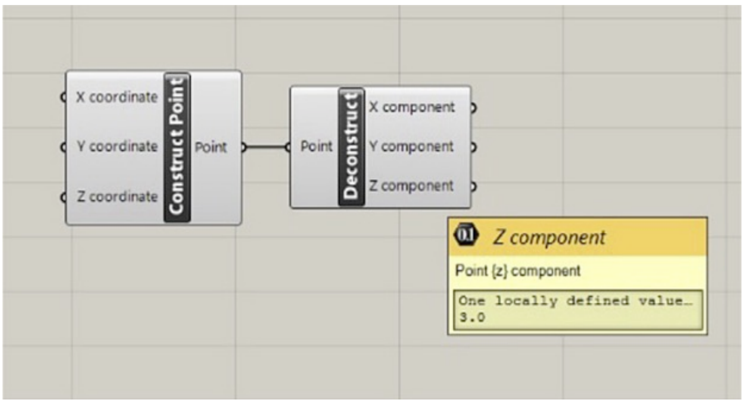
- Flexibility [of editing, minimising reworking]: Historically, Grasshopper was the first visual programming environment where the user directly created and interacted with the graph, as opposed to the graph being generated as a by-product of other interactions. However there are also challenges. For example it is not possible to change the method used by a node. To change the methods, the node has to be deleted. All the connections to that node are lost. The node has to be re-created and the connections re-established. Also to inspect the XYZ coordinate properties of the point node, an additional node had to be added [Figure 9].

- Side effects: none

- Work arounds: required to mitigate convoluted workflow, see below.

- Convoluted workflow: Challenging: As a consequence of the use of data trees, there is no single approach to the creation of the points which can be used both to create the surface and the curves. Two methods were explored: the method in Figure 7 required 16 nodes and the method in Figure 8 required 13 nodes.

- Liveness [including direct interaction with geometry]: Good/Challenging: While the overall performance of the system in dynamics is good, it would be preferable if there was better integration with the host application, specifically generated geometry is not locatable, except if 'baked' and then if it is 'baked', it cannot be re-generated. Ideally, all geometry should be locatable and re-generatable and the user should not be aware of the distinction between 'baked' and 'unbaked' geometry.



```
One locally defined value…
3.0
```

Figure 9: To inspect the XYZ properties of a point node, a 'deconstruct' node must be added to the graph

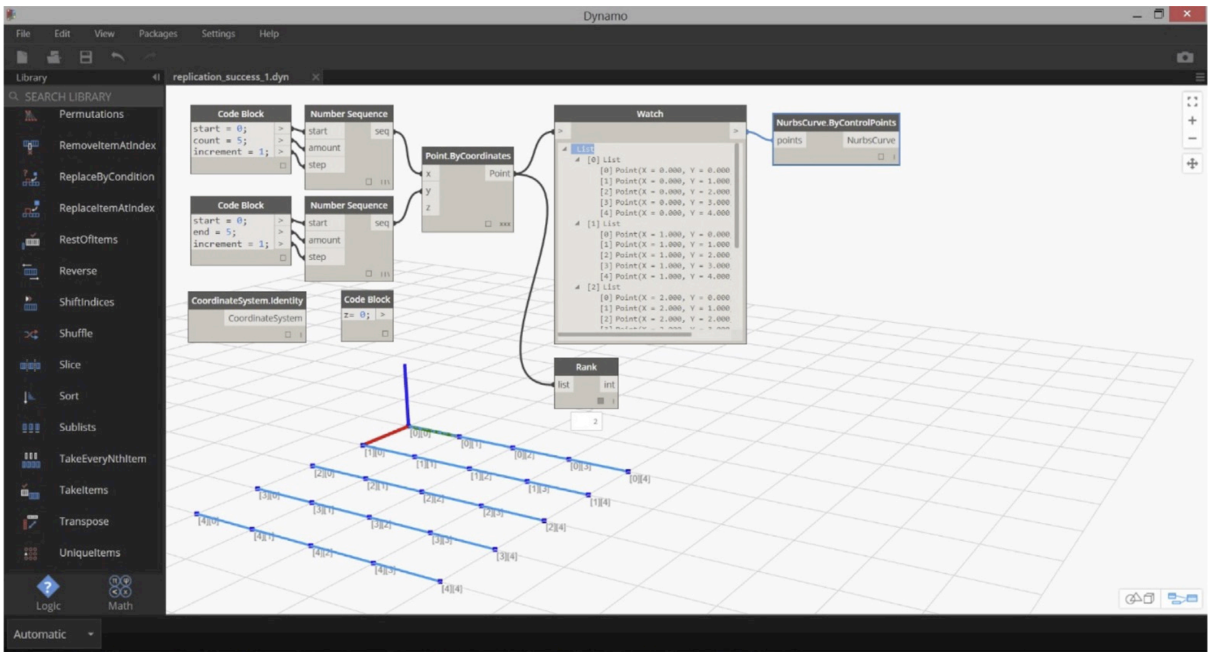
2.3 Dynamo [Version 0.8.0.950]

Note: this analysis has been retested on Version 1.1.0.2094. It was possible to complete the modelling task, but there were significant challenges with the functionality and terminology of Dynamo [Figure 10], as follows:

- Abstraction Barrier: Challenging: Because of the issues with the dimensionality of collections, (see Side effects and Workaround, below) new abstractions have to be introduced such as 'normalised depth' which could otherwise be avoided.

- Semantic Interference [including metaphor trap]: Challenging: There are occasions when it might be preferable to use terminology which is more precise and more consistent. For example in the 'Sequence' node [Figure 11], the second argument is 'Amount' and the third argument is 'Step', but the explanation (in the 'flyover' label) uses the word 'Space'. Dynamo uses the term 'lacing', while the underlying DesignScript language uses the term 'replication' [Figure 12]. It might be preferable to use a single consistent term. While the underlying replication functionality can created 'lacing patterns', it can also create more complex pattern than the term lacing might suggest. Lacing is essentially a metaphor that may be masking this extended functionality.

In addition Dynamo UI uses the term 'Cross Product' to describe the generation of the product set between different inputs sets [Figure 12]. While the use of the term 'cross product' is not incorrect, the more widely accepted term is 'Cartesian Product'.



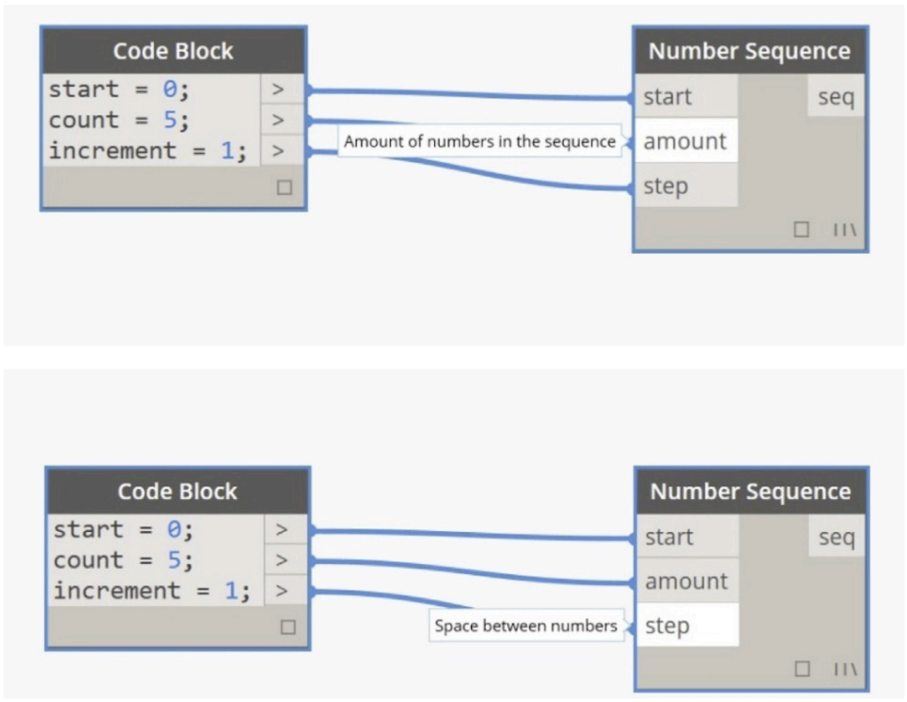
```
start = 0;
count = 5;
increment = 1;
```

```
start = 0;
end = 5;
increment = 1;
```

```
z= 0;
```

```
List
  [0] List
    [0] Point(X = 0.000, Y = 0.000
    [1] Point(X = 0.000, Y = 1.000
    [2] Point(X = 0.000, Y = 2.000
    [3] Point(X = 0.000, Y = 3.000
    [4] Point(X = 0.000, Y = 4.000
  [1] List
    [0] Point(X = 1.000, Y = 0.000
    [1] Point(X = 1.000, Y = 1.000
    [2] Point(X = 1.000, Y = 2.000
    [3] Point(X = 1.000, Y = 3.000
    [4] Point(X = 1.000, Y = 4.000
  [2] List
    [0] Point(X = 2.000, Y = 0.000
    [1] Point(X = 2.000, Y = 1.000
    [2] Point(X = 2.000, Y = 2.000
```

Figure 10: Creating a 2Dpoint collection. This uses a sequence of X coordinates and a sequence of Y coordinates and the default value [0.0] for Z. This 2Dpoint collection can directly be used to create the collection of curves, or a surface [not shown]. Note: the Z coordinate is not defined and the default value of zero is used



```
start = 0;
count = 5;
increment = 1;
```

Figure 11 to create the array of points, the user must first create a number sequence. It might be preferable to use the more precise term 'length' rather than the more ambiguous term 'amount' and to use the more precise term 'increment' rather than the terms 'step' and 'space'

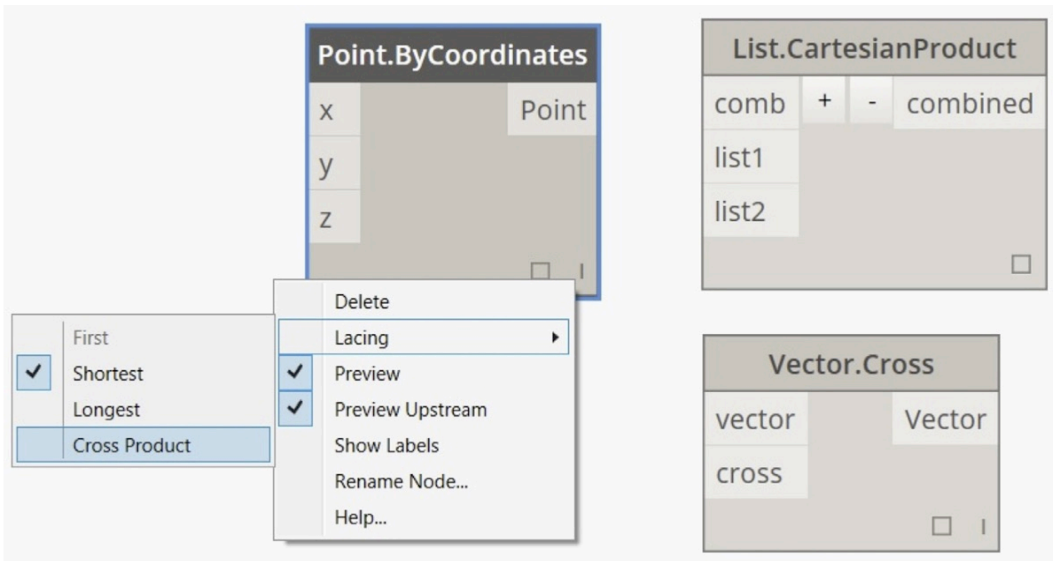


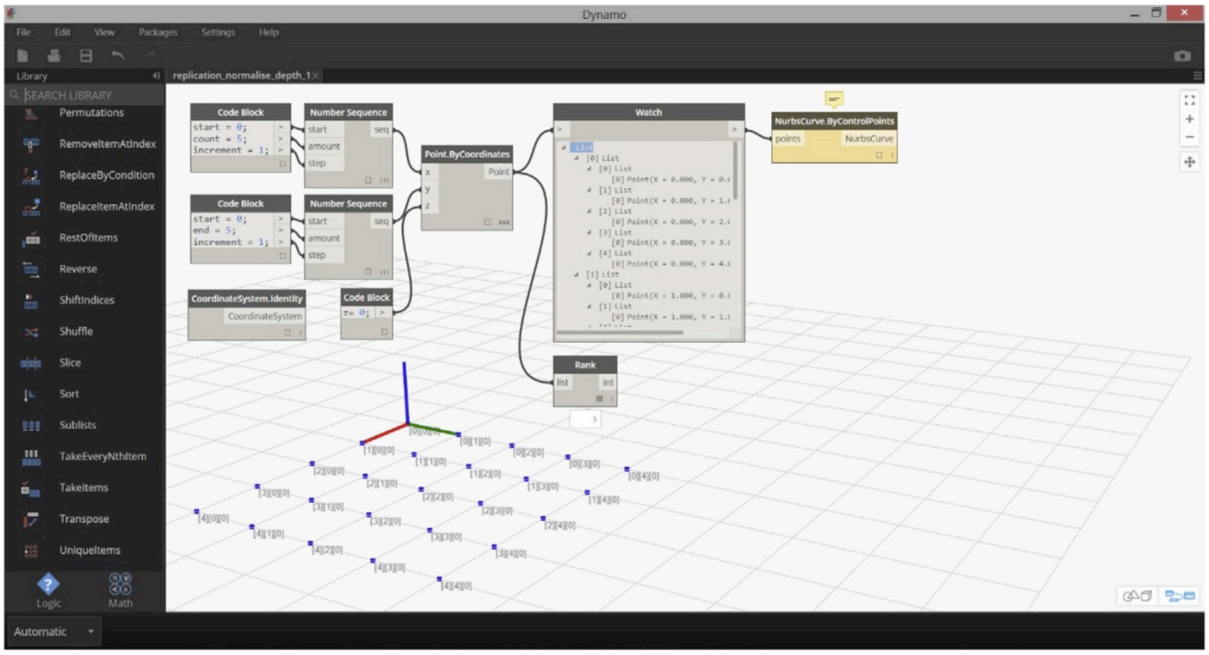
Figure 12: In Dynamo to create the 2D array of points, the user needs to combine the sequence of X coordinates with the sequence of Y coordinates using the 'Cross Product' option. [See 'Semantic Interference section]

Not only is the term 'Cartesian Product' used in the underlying DesignScript language, but there is a Dynamo Node called List.CartesianProduct. Dynamo also implements the 'Cross Product' vector operation. So it would appear that the creation of the product set is separately referred to as a CrossProduct and as a Cartesian Product and the term Cross Product is used to describe both a set operation and a vector operation. It might be preferable in an education context to have a one-to-one mapping between terminology and functionality.

- Consistency between representations: Challenging: In Figure 13 there is a mismatch been the dimensionality of the collection of points [3D] and the visual representation [2D array].

- Discoverability [of functionality, including logical ordering]: Challenging: Both the List functionality and the Geometry functionality are presented with inconsistent menu structures. For example, the 'GetItemAtIndex' [the 'get' method] is found under menu/core/List (together with 54 other methods), while the matching 'insert' method [which inserts an item into a list at an index] is found under menu/BuiltIn (together with 33 other list methods) [Figure 21]. Also, the same term (Geometry) is both the main and sub-menu name, but this menu structure does not communicate the essential functional classification: that Curve, Surface and Solid are all sub classes of Geometry, and that Line, Arc and Circle are all sub classes of Curve, sharing common properties and methods.

- Flexibility: It is not possible to change the method used by a node. To change the methods, the node has to be deleted. All the connections to that node are lost. The node has to be re-created and the connections re-established. While a single 'watch' node can be used to inspect the XYZ coordinate properties of the point node, three separate nodes for the X, Y and Z properties had to be added [Figure 20] in order to access these values.



```
start = 0;
count = 5;
increment = 1;
```

```
start = 0;
end = 5;
increment = 1;
```

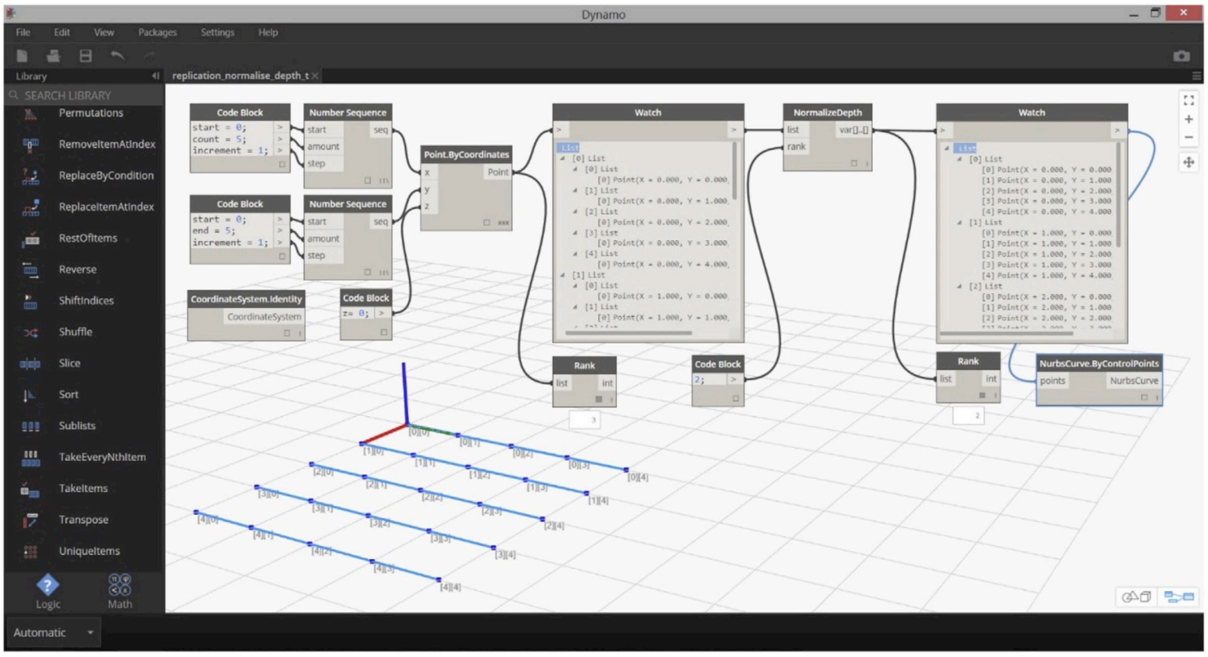
```
z= 0;
```

```
List
  [0] List
    [0] List
      [0] Point(X = 0.000, Y = 0.(
    [1] List
      [0] Point(X = 0.000, Y = 1.(
    [2] List
      [0] Point(X = 0.000, Y = 2.(
    [3] List
      [0] Point(X = 0.000, Y = 3.(
    [4] List
      [0] Point(X = 0.000, Y = 4.(
  [1] List
    [0] List
      [0] Point(X = 1.000, Y = 0.(
    [1] List
      [0] Point(X = 1.000, Y = 1.(
```

Figure 13: Creating an array of points using a collection of X coordinates and a collection of Y coordinates. In Figure 10, no value was defined for the Z coordinate, the default value of zero is used and a 2D array of points was created. In this case a single Z coordinate is defined [with the value zero], but the behaviour alters and a 3D collection of points is created. There is no change in the value of the Z coordinate [which is zero] just a change from using the default value or explicitly defined value. This change has created an unexpected side effect

- Side effects: Challenging: During the creation of the array of points an important unanticipated side effect occurred. In Figure 10, when the 2D point array is initially created there is no explicit value defined for the Z coordinate. Instead the built-in default value [0.0] is used. However, in Figure 13, if a value is explicitly provided for the Z coordinate [even the value 0.0 which is the same as the default value] then the replication strategy 'recognises' three variables as inputs to the Point node and therefore creates a 3D array of points. The spatial configuration is a 2D, but the data structure is a 3D array. This not only gives an 'inconsistency between representations' (noted above) but results in the Curve creation node having the wrong dimension of input which then fails. So comparing Figures 10 and 13, we can see that the user's action of simply connecting the Z coordinate node to the Point.ByCoordinates node, with effectively no change in value for the Z coordinate, changes the dimensionality of the resulting output. The established replication strategy in other applications [such as Generative-Components] or indeed in the underlying DesignScript language [within Dynamo, see Figure 15] does not add an extra dimension to the output collection if a single value input is detected, but only when a collection is detected. So the regular nodes in Dynamo appear to be presenting a different replication behaviour to that available in the underlying DesignScript language. This issue is now the subject of a discussion on the Dynamo discussion groups (Dynamo issue #6528, 2016)
- Workarounds: Challenging: To correct for this side effect, the user has to introduce a workaround to reduce the dimensionality of the point array back to 2D [Figure 14] using the 'normalised depth' node. While the concept that an array can be 'flattened' inherent in the 'normalised depth' node is important, requiring its use in this context presents an unfortunate and unnecessary abstraction barrier to the novice user. Figure 16 shows an alternative way to

create the array of points using a code block node and a DesignScript expression with the underlying 'Point.ByCoordinates method and replication guides (see Figure 17).



```
start = 0;
count = 5;
increment = 1;
```

```
start = 0;
end = 5;
increment = 1;
```

```
z= 0;
```

```
2;
```

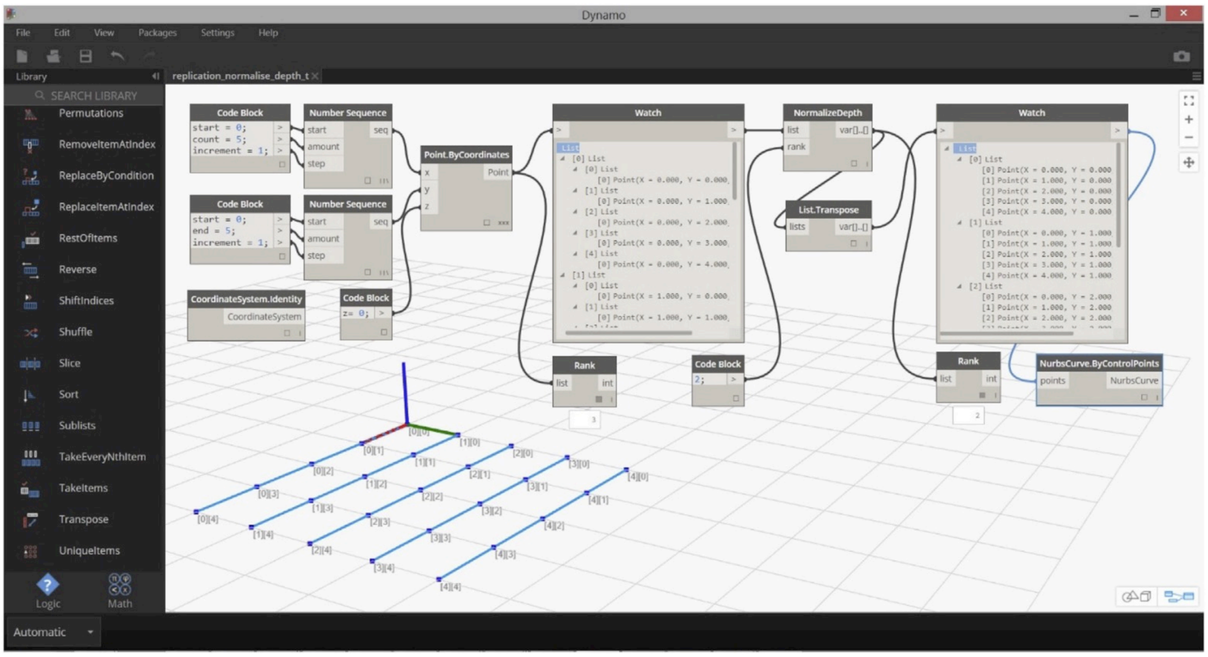
```
List
[0] List
  [0] List
    [0] Point(X = 0.000, Y = 0.000,
  [1] List
    [0] Point(X = 0.000, Y = 1.000,
  [2] List
    [0] Point(X = 0.000, Y = 2.000,
  [3] List
    [0] Point(X = 0.000, Y = 3.000,
  [4] List
    [0] Point(X = 0.000, Y = 4.000,
[1] List
  [0] List
    [0] Point(X = 1.000, Y = 0.000,
  [1] List
    [0] Point(X = 1.000, Y = 1.000,
```

```
List
[0] List
  [0] Point(X = 0.000, Y = 0.000
  [1] Point(X = 0.000, Y = 1.000
  [2] Point(X = 0.000, Y = 2.000
  [3] Point(X = 0.000, Y = 3.000
  [4] Point(X = 0.000, Y = 4.000
[1] List
  [0] Point(X = 1.000, Y = 0.000
  [1] Point(X = 1.000, Y = 1.000
  [2] Point(X = 1.000, Y = 2.000
  [3] Point(X = 1.000, Y = 3.000
  [4] Point(X = 1.000, Y = 4.000
[2] List
  [0] Point(X = 2.000, Y = 0.000
  [1] Point(X = 2.000, Y = 1.000
  [2] Point(X = 2.000, Y = 2.000
```

```
3
```

```
2
```

Figure 14: In Dynamo, in order to correct for the side effect [in Figure 13] the user has to add a 'work around', which is to 'normalise the depth' of the point array. But this introduces an addition concept 'normalise depth' for the novice user. Having to understand this additional concept might become an 'abstraction barrier' for some novice users



```
start = 0;
count = 5;
increment = 1;
```

```
start = 0;
end = 5;
increment = 1;
```

```
z= 0;
```

```
2;
```

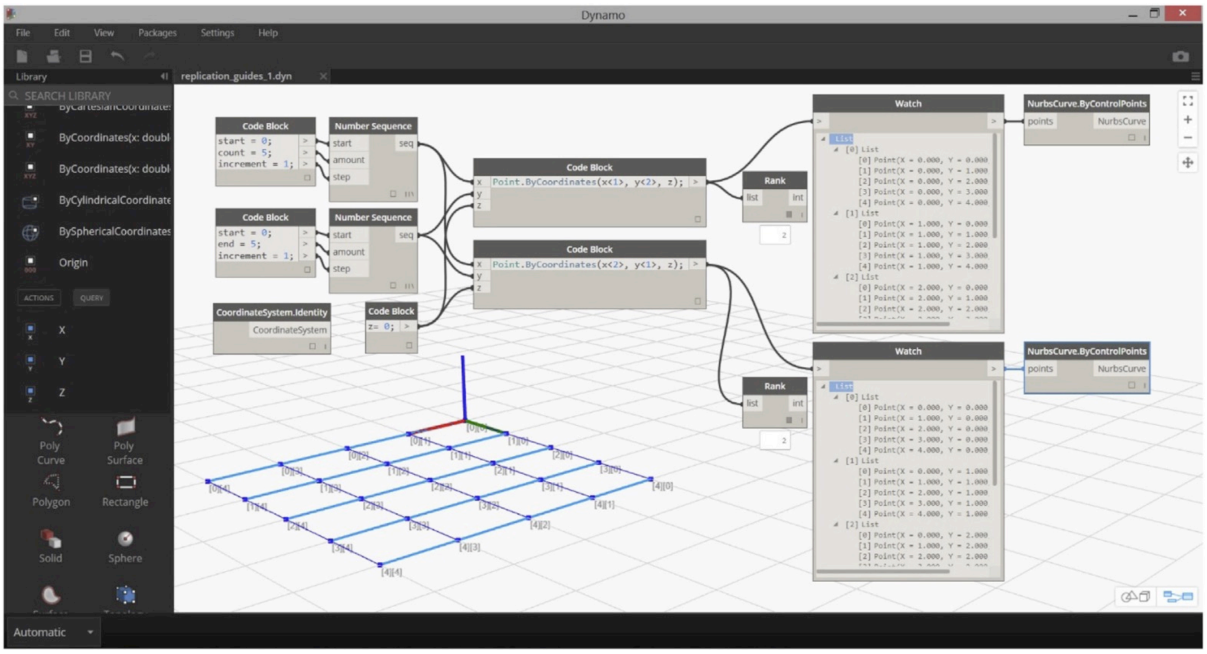
```
List
[0] List
  [0] List
    [0] Point(X = 0.000, Y = 0.000,
  [1] List
    [0] Point(X = 0.000, Y = 1.000,
  [2] List
    [0] Point(X = 0.000, Y = 2.000,
  [3] List
    [0] Point(X = 0.000, Y = 3.000,
  [4] List
    [0] Point(X = 0.000, Y = 4.000,
[1] List
  [0] List
    [0] Point(X = 1.000, Y = 0.000,
  [1] List
    [0] Point(X = 1.000, Y = 1.000,
```

```
List
[0] List
  [0] Point(X = 0.000, Y = 0.000
  [1] Point(X = 1.000, Y = 0.000
  [2] Point(X = 2.000, Y = 0.000
  [3] Point(X = 3.000, Y = 0.000
  [4] Point(X = 4.000, Y = 0.000
[1] List
  [0] Point(X = 0.000, Y = 1.000
  [1] Point(X = 1.000, Y = 1.000
  [2] Point(X = 2.000, Y = 1.000
  [3] Point(X = 3.000, Y = 1.000
  [4] Point(X = 4.000, Y = 1.000
[2] List
  [0] Point(X = 0.000, Y = 2.000
  [1] Point(X = 1.000, Y = 2.000
  [2] Point(X = 2.000, Y = 2.000
```

```
3
```

```
2
```

Figure 15 In Dynamo the user adds a transpose node in order to create the alternative set of curves



```
start = 0;
count = 5;
increment = 1;
```

```
start = 0;
end = 5;
increment = 1;
```

```
z= 0;
```

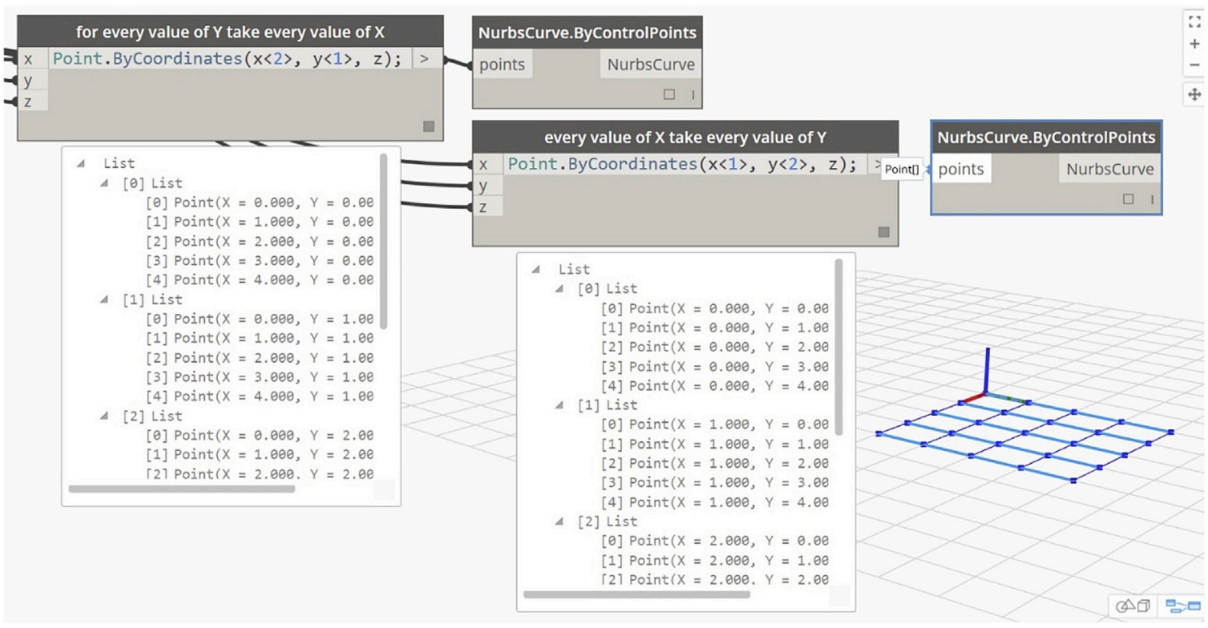
```
Point.ByCoordinates(x<1>, y<2>, z);
```

```
Point.ByCoordinates(x<2>, y<1>, z);
```

```
List
  [0] List
      [0] Point(X = 0.000, Y = 0.000
      [1] Point(X = 0.000, Y = 1.000
      [2] Point(X = 0.000, Y = 2.000
      [3] Point(X = 0.000, Y = 3.000
      [4] Point(X = 0.000, Y = 4.000
  [1] List
      [0] Point(X = 1.000, Y = 0.000
      [1] Point(X = 1.000, Y = 1.000
      [2] Point(X = 1.000, Y = 2.000
      [3] Point(X = 1.000, Y = 3.000
      [4] Point(X = 1.000, Y = 4.000
  [2] List
      [0] Point(X = 2.000, Y = 0.000
      [1] Point(X = 2.000, Y = 1.000
      [2] Point(X = 2.000, Y = 2.000
```

```
List
  [0] List
      [0] Point(X = 0.000, Y = 0.000
      [1] Point(X = 1.000, Y = 0.000
      [2] Point(X = 2.000, Y = 0.000
      [3] Point(X = 3.000, Y = 0.000
      [4] Point(X = 4.000, Y = 0.000
  [1] List
      [0] Point(X = 0.000, Y = 1.000
      [1] Point(X = 1.000, Y = 1.000
      [2] Point(X = 2.000, Y = 1.000
      [3] Point(X = 3.000, Y = 1.000
      [4] Point(X = 4.000, Y = 1.000
  [2] List
      [0] Point(X = 0.000, Y = 2.000
      [1] Point(X = 1.000, Y = 2.000
      [2] Point(X = 2.000, Y = 2.000
```

Figure 16: An alternative way to create the array of points is to use a code block node and a DesignScript expression with the underlying 'Point.ByCoordinates method using replication guides <1> and <2> to control the replication behaviour


for every value of Y take every value of X
x
y
z
NurbsCurve.ByControlPoints
points
NurbsCurve
every value of X take every value of Y
Point[]

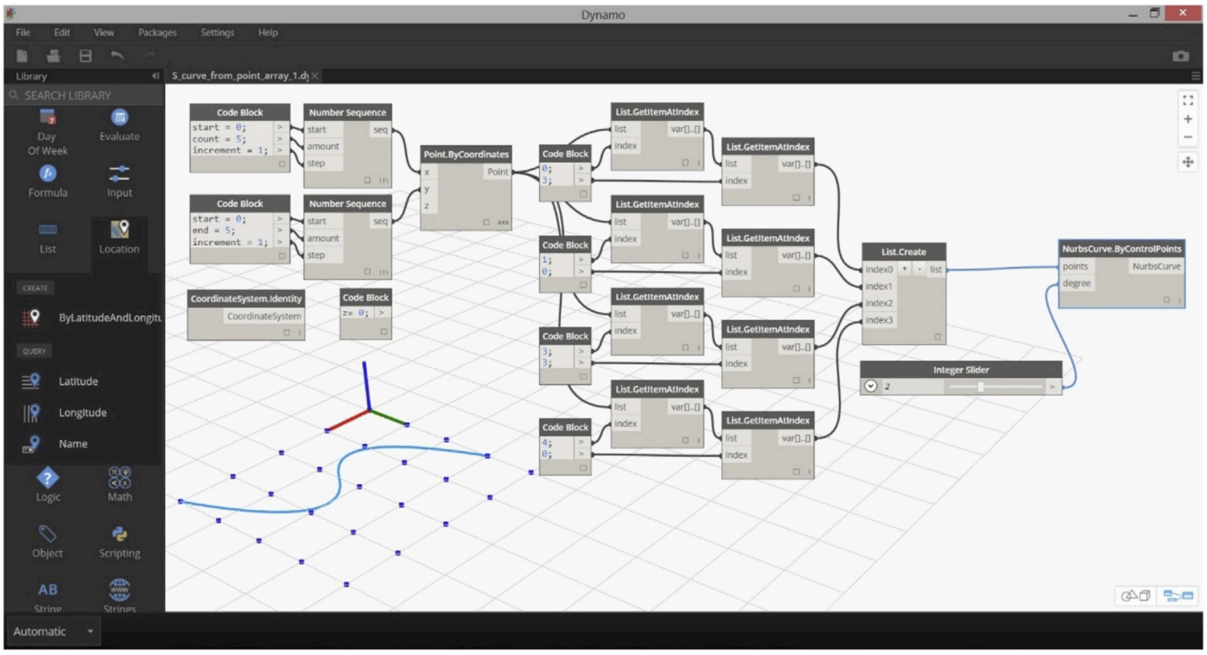
```
Point.ByCoordinates(x<2>, y<1>, z);
```

```
List
  [0] List
      [0] Point(X = 0.000, Y = 0.00
      [1] Point(X = 1.000, Y = 0.00
      [2] Point(X = 2.000, Y = 0.00
      [3] Point(X = 3.000, Y = 0.00
      [4] Point(X = 4.000, Y = 0.00
  [1] List
      [0] Point(X = 0.000, Y = 1.00
      [1] Point(X = 1.000, Y = 1.00
      [2] Point(X = 2.000, Y = 1.00
      [3] Point(X = 3.000, Y = 1.00
      [4] Point(X = 4.000, Y = 1.00
  [2] List
      [0] Point(X = 0.000, Y = 2.00
      [1] Point(X = 1.000, Y = 2.00
      [2] Point(X = 2.000, Y = 2.00
```

```
Point.ByCoordinates(x<1>, y<2>, z);
```

```
List
  [0] List
      [0] Point(X = 0.000, Y = 0.00
      [1] Point(X = 0.000, Y = 1.00
      [2] Point(X = 0.000, Y = 2.00
      [3] Point(X = 0.000, Y = 3.00
      [4] Point(X = 0.000, Y = 4.00
  [1] List
      [0] Point(X = 1.000, Y = 0.00
      [1] Point(X = 1.000, Y = 1.00
      [2] Point(X = 1.000, Y = 2.00
      [3] Point(X = 1.000, Y = 3.00
      [4] Point(X = 1.000, Y = 4.00
  [2] List
      [0] Point(X = 2.000, Y = 0.00
      [1] Point(X = 2.000, Y = 1.00
      [2] Point(X = 2.000, Y = 2.00
```

Figure 17: By selecting different replication guides, either x<1>, y<2> or x<2>, y<1>, the user can control the way the 2D array of points will be built and therefore the way curves will be built from the array of points



```
start = 0;
count = 5;
increment = 1;
```

```
start = 0;
end = 5;
increment = 1;
```

```
z= 0;
```

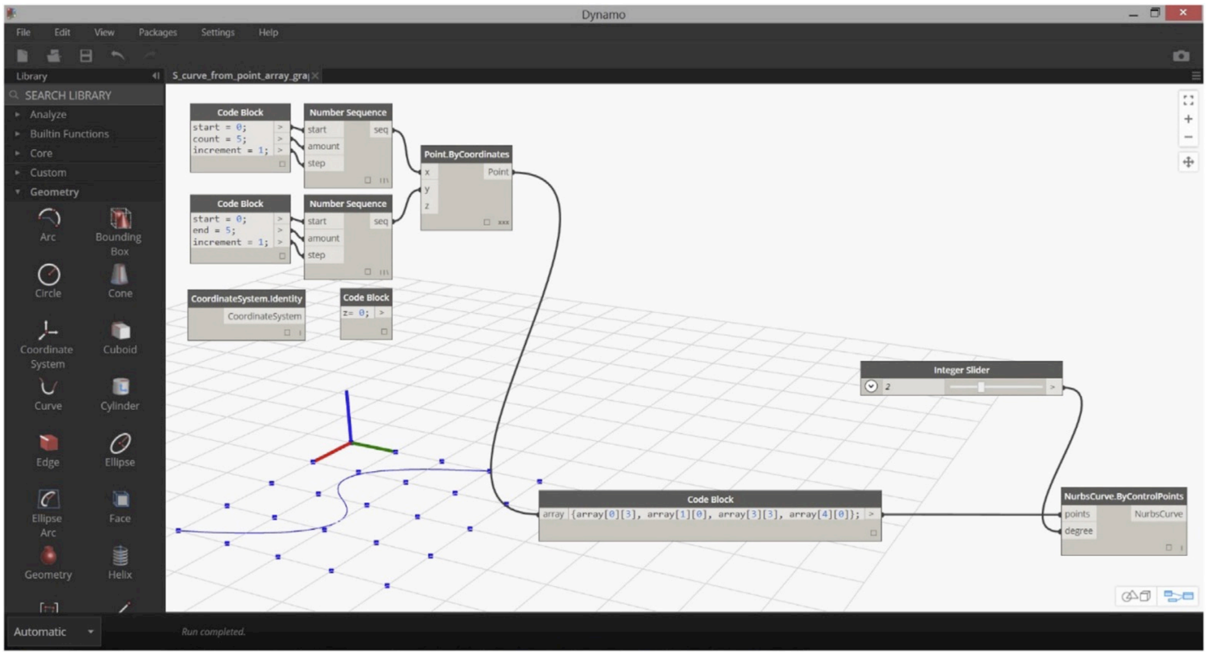
```
0;
3;
```

```
1;
0;
```

```
3;
3;
```

```
4;
0;
```

Figure 18: In Dynamo using the available nodes, to select points to create the 'S' curve, the user has to repeat a complex two stage selection process, first selecting the sub list from the 2D collection and then selecting the specific point from the sub list. There is no multi-index [or path] selection node to select an item from a multi-dimensional collection. 13 nodes are required to select 4 points.



```
start = 0;
count = 5;
increment = 1;
```

```
start = 0;
end = 5;
increment = 1;
```

```
z= 0;
```

```
{array[0][3], array[1][0], array[3][3], array[4][0]};
```

Figure 19: In Dynamo, and using DesignScript within a code block node, it is possible to select points from a collection using indices. In this case the user has to hand construct this code fragment

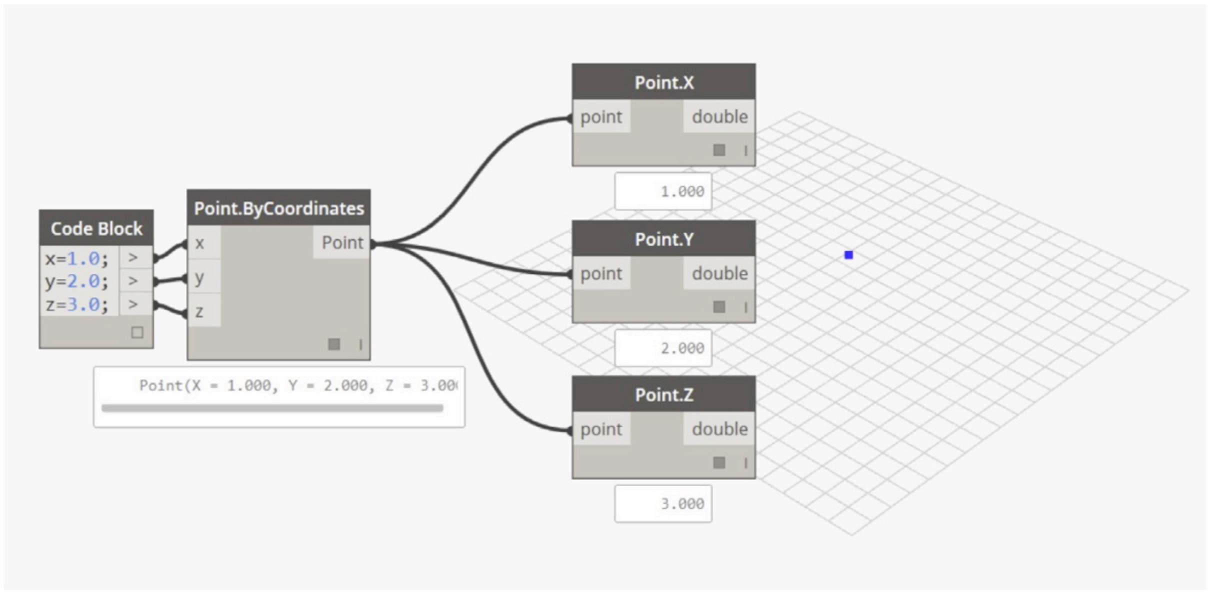


Figure 20: To inspect the XYZ properties of a point node, three separate nodes must be added to the graph

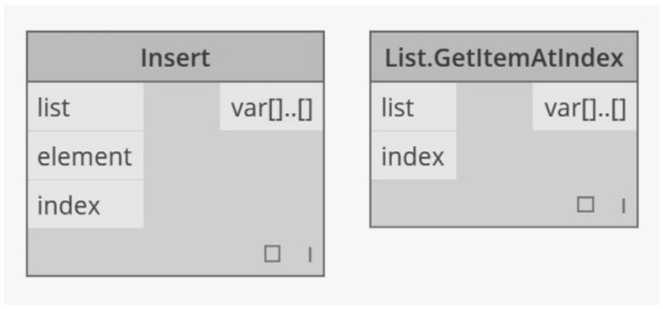


Figure 21: The 'Insert' function for 'List' is found under: 'Builtin Functions/Insert' submenu [and uses the term 'element'], while the corresponding 'Get' function for 'List' is found under 'Core/List/Action/GetItemAtIndex' sub menu and uses the term 'item'

- Convoluted workflow: see Liveness, below.

- Liveness [including direct interaction with geometry]: Challenging: In the absence of any interaction with the generated geometry, selection tasks may become extremely arduous. For example, to create the 'S' curve [Figure 18], the user has to repeat a complex two stage selection process, first selecting the sub list from the 2D collection and then selecting the specific point from the sub list. There is no multi-index [or path] selection node which can be used to select an item from a multi-dimensional collection. This effectively required the user to define 13 nodes just to select 4 input points for the 'S' curve. An alternative way to select the points for the 'S' curve, is to use a Design-Script collection expression with indices within a code block node [Figure 19]. However, the user has to hand construct this code fragment. This is essentially exactly the same code fragment which is used in Generative-Components [Figure 4], except that in the case of GenerativeComponents this code fragment is built automatically by the application in response to the user interactively selecting points in the geometry window. To inspect the XYZ properties of the point node, three additional node had to be added to the graph [Figure 20] compared to one node in Grasshopper [Figure 9] and no nodes in GenerativeComponents [Figure 6].

3 **Analysis**

The evaluation of the three parametric design systems is formally comprised of the stated cognitive criteria detailed above, which exclude influences such as institutional preferences and prior experience, variations in the ability and preferences of the students. The evaluation is summarised in Table 1 and visually presented in Figure 22. The learning curves in Figure 22 are purely indicative and use the same visual conventions introduced by Myer (2002). The normal learning activity is characterised by the inclined line and the cognitive challenges are represented by vertical lines. The height of the vertical line indicates the cognitive challenges at each stage in the test exercise. However this is not meant to be precisely quantified because different students with varying levels of interest, abilities and perseverance may react differently to these challenges. These cognitive challenges may be classified as:

'Absolute barriers'

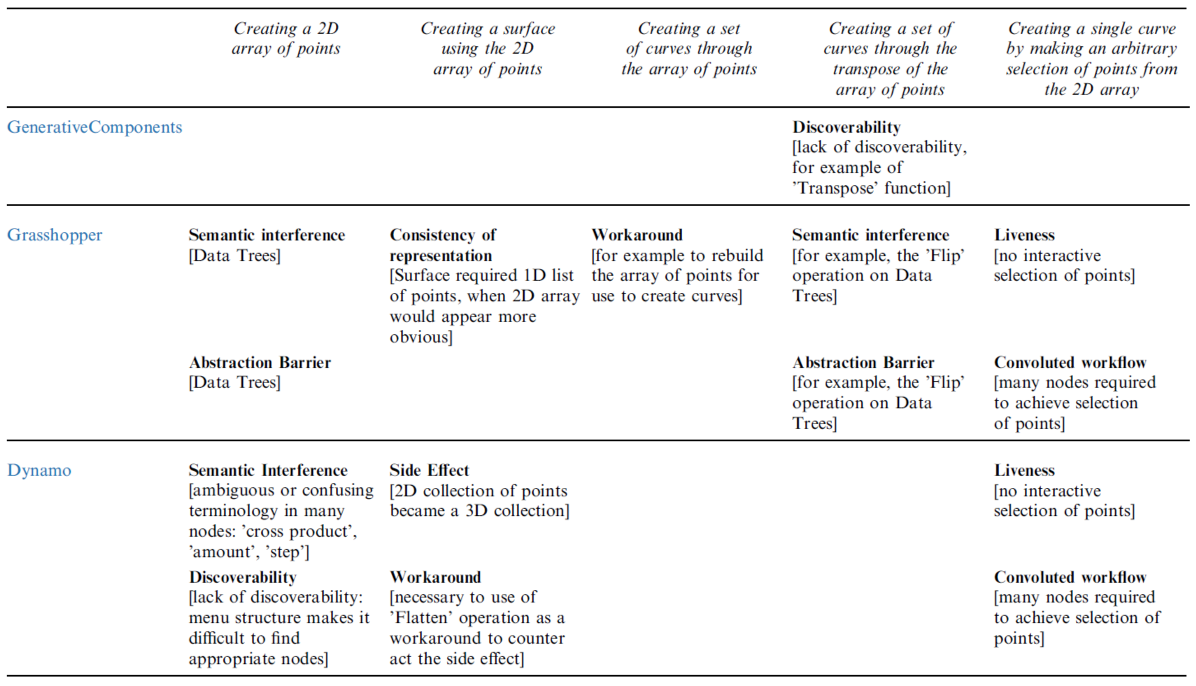
1. The required functionality does not exist [theoretically none of the applications failed to provide all the functionality because an experienced user was able to complete the tests. However, there were occasions with each of the three applications where even an experienced user would have failed to complete the tests without the direct guidance and intervention of experts from the respective software vendors].
2. [Discoverability] The required functionality exists but is undocumented, therefore the functionality would not be discovered by a novice user [for example: the Transpose function in GenerativeComponents]

'Effective barriers': theoretically the novice user should be able to complete the task. Whether the user overcomes or succumbs to these barriers may depend on other individual and contextual factors influencing the user's *commitment and perseverance. Both from an educational and practice perspective, it would be essential to eliminate or reduce these barriers.*

3. *[Discoverability] The required functionality exists and is documented, but in such an illogical way that the novice user is unlikely to find [for example: List methods in the menu structure in Dynamo]*
4. *[Abstraction barrier] The required functionality is not directly accessible but a more experience user might be able to infer or 'reverse engineer' the functionality out of other features of the system (which is unlikely for a novice user as this would present additional abstraction barrier). [for example: Grasshopper Data Trees could be used to emulate arrays]*
5. *[Side effects] The functionality exist, but creates unexpected side effects (which require additional concepts to be understood, potentially introducing additional abstraction barriers) [for example: under some unexpected conditions Dynamo replication (or lacing) changes the dimension of the generated collection]*
6. *[Workaround] A workaround to the side effects exists [but require additional concepts to be understood, again potentially introducing additional abstraction barriers] [for example: Dynamo 'flatten' functionality, used to address the 'dimensionality' side effect, mentioned above]*
7. *[Convoluted workflow] The required functionality exists and is documented but to complete the task requires a convoluted workflow [for example: Dynamo required 13 nodes to select 4 points]*

’Incorrect pedagogy’

1. *[Semantic interference] The required functionality exists but is describedby inappropriate terms therefore leading to an incorrect pedagogy. Thisis particularly important in an educational context. [for example: Dynamo’s use of the term ‘CrossProduct’, when the term ‘Cartesian Product’ might be more appropriate]*



	Creating a 2D array of points	*Creating a surface using the 2D array of points*	*Creating a set of curves through the array of points*	*Creating a set of curves through the transpose of the array of points*	*Creating a single curve by making an arbitrary selection of points from the 2D array*
GenerativeComponents				**Discoverability** [lack of discoverability, for example of ’Transpose’ function]	
Grasshopper	**Semantic interference** [Data Trees]	**Consistency of representation** [Surface required 1D list of points, when 2D array would appear more obvious]	**Workaround** [for example to rebuild the array of points for use to create curves]	**Semantic interference** [for example, the ’Flip’ operation on Data Trees]	**Liveness** [no interactive selection of points]
	Abstraction Barrier [Data Trees]			**Abstraction Barrier** [for example, the ’Flip’ operation on Data Trees]	**Convoluted workflow** [many nodes required to achieve selection of points]
Dynamo	**Semantic Interference** [ambiguous or confusing terminology in many nodes: ’cross product’, ’amount’, ’step’]	**Side Effect** [2D collection of points became a 3D collection]			**Liveness** [no interactive selection of points]
	Discoverability [lack of discoverability: menu structure makes it difficult to find appropriate nodes]	**Workaround** [necessary to use of ’Flatten’ operation as a workaround to counter act the side effect]			**Convoluted workflow** [many nodes required to achieve selection of points]

Table 1: Comparative cognitive barriers for the three parametric design systems, for exercises 1-5

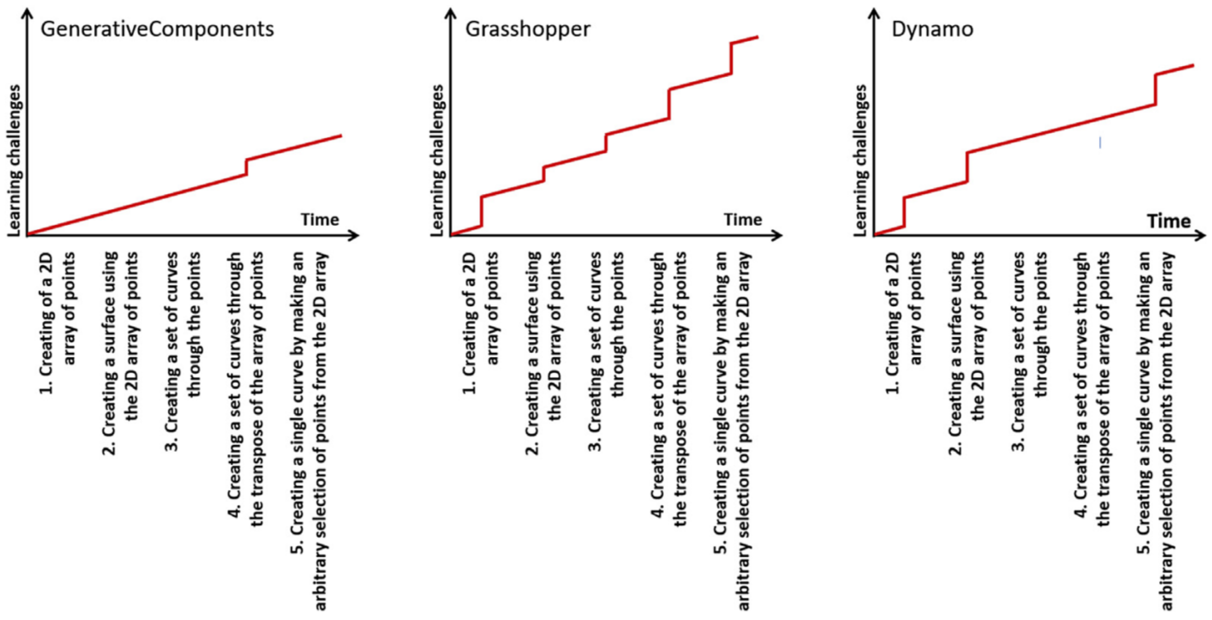


Figure 22: The learning curves for the three parametric design system

4 **Conclusions**

The important role of parametric design applications is to present parametric design concepts to designers and for the designers to be able to use these applications to express parametric design thinking. In this paper we have used a standard modelling exercise and a set of criteria inspired by the 'cognitive dimensions' research to evaluate important conceptual and usability aspects of three of the main parametric design systems. Our own immediate purpose in this evaluation was to help select systems which would be suitable for undergraduate education in parametric design thinking. The broader aim was to establish a means by which an academic or industry professional, typically an expert user, can evaluate and anticipate the ease with which novice user would learn to use a parametric design system. This evaluation necessarily pre-dated any particular parametric design course and was carried out by the authors. While the original intention of the study was to select a suitable system for teaching parametric design, this study has uncovered some major conceptual and usability concerns with all the available parametric design systems. Many of these concerns could easily be resolved by simply exposing or 'repackaging' the underlying functionality, by changes to the user interface design, or even by quite minor changes to terminology.

It may be noted that this study does not include empirical observation of actual novice use, and particularly more formal experiments with real student groups. There are several related reasons for this. In practice, novice student aptitude varies considerably, making reliable observation of the barriers to learning difficult without quite large statistical samples. Further, a description of such points of difficulty in a modelling exercise does not explain the underlying cognitive cause. It is just such an explanation that we have attempted by grounding this evaluation entirely in the cognitive dimensions given.

This lack of empirical experiment necessarily limits the conclusions of this study, in that real student experiences would be required to properly test the details of the evaluation method: whether the particular cognitive dimensions used best represent real student samples, whether differences in background or bias result in different curves, etc. As such, the results must remain generic at present, referring to broad differences between the software. We anticipate future observations of actual student use over time would function both as a test of the particular choice of cognitive dimensions by making qualitative and quantitative comparisons to the observed learning curves shown here, and to allow extension of the scope of conclusions. Indeed if there were to be opportunities for future empirical studies into the way parametric design thinking is supported by available parametric design tools, then it is hoped that these studies would be concerned with more than basic usability issues as reported here, and would be able to focus on more interesting and substantive conceptual and practical issues. It is hoped that feedback from such future empirical studies will play an important role in informing future software design decisions and design educational strategies.

In this study the modelling exercise was chosen to present common and essential parametric modelling concepts and operations of increasing complexity particularly relevant to architectural design. As reported earlier, the indicative learning curves from the evaluation of the three parametric design systems suggest three quite different learning profiles. In some cases the cognitive challenges occur uniformly over the modelling task; in other cases the cognitive challenges are concentrated at the earlier and later phases. The differences between these curves could be used to select a suitable parametric design system for users who have different levels of skill, or to plan teaching strategies, or to anticipate where students might experience different cognitive difficulties.

It is also recognised that different exercises might result in different learning curves. For example, in cases where the use and conceptual understanding of 'collections' is not considered an essential aspect of parametric design thinking, then a modelling task could be constructed which excluded 'collections'. This might remove some of the learning barriers present in the initial stages of the Grasshopper and Dynamo curves and learning these systems by novice users might be far more rapid. A further investigation of how these curves might change with exercises which use different parametric design concepts may well shed light on why particular parametric design systems are chosen in different institutional contexts or by different user communities.

There are also important conclusions to be made about the evaluative methods developed for this study, and how these might inform future software design. As the original authors of the 'cognitive dimension' research suggest (Green & Blackwell, 1998), there is a need for practical usability tools by which everyday software developers and educators can assess cognitively-relevant system properties and identify important system design trade-offs. Therefore, we should consider the evaluation methods proposed here as the start of an open discussion to further develop and refine ways to measure the suitability of parametric design systems.

Acknowledgements
The authors would like to acknowledge the assistance of:
Marc Thomas in the review of GenerativeComponents.
Arthur Mamou-Mani in the review of Grasshopper.
Andreas Dieckmann in the review of Dynamo.
Alan Blackwell in discussions about Cognitive Dimensions.
The Reviewers who provided valuable feedback which contributed to the refinement of the paper.

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References

Aish, R., & Woodbury, R. (2005). Multi-level interaction in parametric design. In A. Butz, B. Fisher, A. Kruger, & P. Olivier (Eds.), Smart graphics, 5th International Symposium. Springer.
Auslander, P. (2008). Liveness: Performance in a mediatized culture (2nd ed.). Routledge.
Barr, P. (2003). User-interface metaphors in theory and practice. Victoria University of Wellington. http://www.pippinbarr.com/academic/Pippin_Barr_MSc_Thesis.pdf
Burnett, M. M., Baker, M. J., Bohus, C., Carlson, P., Yang, S., & Van Zee, P. (1995). Scaling up visual programming languages. Computer, 28(3). ftp://ftp.cs.orst.edu/pub/burnett/Computer-scalingUp-1995.pdf.
http://web.engr.oregonstate.edu/wburnett/Scaling/ScalingUp.html
or
Davis, D., Burry, J., & Burry, M. (2011). The flexibility of logic programming. In C. Herr, G. Ning, R. Stanislav, & M. Schnabel (Eds.), Circuit Bending, Breaking and Mending: Proceedings of the 16th International Conference on Computer Aided Architectural Design Research in Asia (pp. 29e38). Australia: The University of Newcastle. http://www.danieldavis.com/the-flexibility-oflogic-programming-parametrically-regenerating-the-sagrada-familia/
Dynamo issue 6528. (2016). Point.ByCoordinates (X, Y,Z) inconsistent behaviour using cross product lacing #6528. https://github.com/DynamoDS/Dynamo/issues/6528
Dynamo' http://dynamobim.org/