StructuralComponents

A CLIENT-SERVER SOFTWARE ARCHITECTURE FOR FEM-BASED STRUCTURAL DESIGN EXPLORATION

Bastiaan Marinus van de Weerd

June, 2013

Delft University of Technology

COMMITTEE:

Prof.dr.ir. J.G. Rots

Dr.ir. J.L. Coenders

Ir. J.W. Welleman

Ir. A. Rolvink

This page is intentionally left blank.

Summary

The goal of StructuralComponents is to allow an engineer to quickly compose an early-stage structural concept and immediately analyse its performance. In a sense, it represents an opportunity in the gap between rough hand calculations and expensive analysis models (see Figure 1-a). At a high level, the tool follows the method of the *Conceptual Story* (Coenders, 2011): the idea of a design justification in the form of a narrative combining analytical methods and computer tools with reasoning and thought models, in an attempt to deal with imprecision and uncertainty (see Figure 1-b).

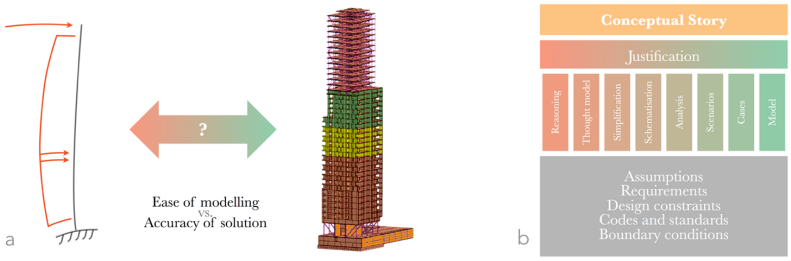


Figure 1: a) The gap between hand calculations (image courtesy of Wagemans), and b) the *Conceptual Story* (image courtesy of Coenders, 2011).

This thesis represents the third in a series of MSc theses on StructuralComponents (Breider, 2008; and Rolvink, 2010). Both previous theses allowed the user to create a structural concept using the parametric and associative design ('PAD') paradigm using *GenerativeComponents* (Bentley, 2013) and *Rhino Grasshopper* (McNeel, 2013), respectively. The prototype implementations would structurally analyse the user's model as it was being created and give immediate performance feedback. This thesis proposes its own prototype implementation with similar functionality, and further incorporating new concepts.

The general problem definition, of providing more adequate earl-stage design tools and dealing with imprecision and uncertainty, forms a guiding principle throughout this thesis. It is accompanied by specific problem statements that inform the thesis objectives (see Section 1.3): a limited adherence to formal research on design in engineering; a limited versatility of the *super-element method* for analysis; and the restriction of the previously plugin-based software architecture. To address theses, the three objectives for this thesis are:

- Bringing the StructuralComponents concept in adherence with the *characteristic design cycle* and **increase emphasis on the synthesis phase**, i.e. the generation of design alternatives.
- Allowing structural versatility beyond the previously used *super-element method* by **developing a Finite Element Method Analysis engine**.
- Removing external software and device dependencies by **reimplementing the tool using a client–server software architecture**.

CONCEPT AND METHODOLOGY

The major goal of the reimplemented system is an increased emphasis on design generation and exploration. To achieve this, it relies on a system of *abstraction*: a system of automatic generation of design variants and subsequent interactive visualisation and exploration of the resulting performance data. This system lets the user define measurable performance metrics, and uses data to show which parameters yield a good solution.

The proposed system alternates between *composition* and *abstraction* phases. During the composition phase, the user synthesises a solution to the design problem: they compose a PAD design concept out of various functional components. During the *abstraction* phase, they analyse the full potential of the solution: they abstract over the parameters to assess alternative solution instances. The transition from *composition* to

abstraction features the definition of *metrics* which indicates the performance of a solution. The transition back features the deduction of the most promising parameter ranges.

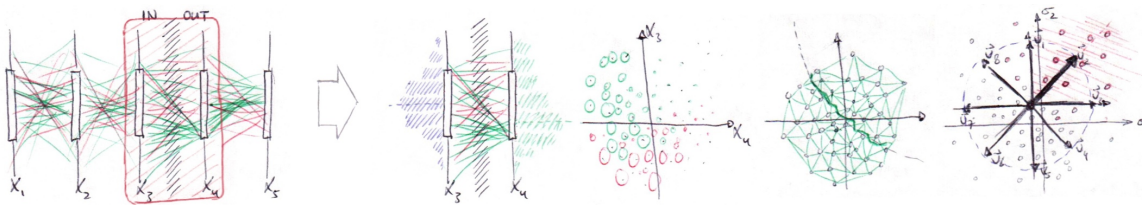


Figure 2: The various visusalisation types: parallel coordinates, frequency histogram, and various scatter plots.

The analysed data resulting from an *abstraction* is ready to be explored. A number of visualisation types are proposed which allow fairly deep exploration against acceptable ease of implementation (see Figure 2, also Section 3.2.3). The parallel coordinates plot is well-suited to visualise high-dimensional data, and serves as an overview. The frequency histogram shows data distribution. The scatter plot shows pairs of values as points on a plane, and can be adorned with *metric* sensitivity indicators, *parameter* relationship contours, and proximity to *constraints*.

FINITE ELEMENT ANALYSIS

The system incorporates a built-in *Finite Element Method* ('FEM') analysis engine. Its core algorithms employ the Cholesky decomposition matrix solving algorithm (Davis, 1996), employing typical stiffness matrix sparsity for more efficient computation. It further employs Reverse Cuthill-McKee node numbering to optimise matrix bandwidth (Cuthill & McKee, 1969).

The engine performs three-dimensional static analysis. In its prototype implementation it supports simple bar and beam element, though it is written with extensibility in mind. Further features of the engine include both rigid and flexible point, line, and plane constraints; user-defined element stiffnesses; and linear and torque forces.

SOFTWARE ARCHITECTURE

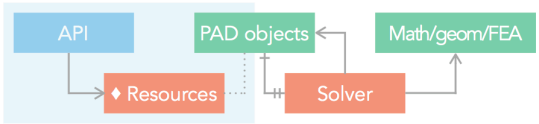


Figure 3: A class diagram showing the nested MVC architecture: simple *CRUD* operations for parametric and associative modelling, and job scheduling for solution and analysis.

The architecture of the software system can roughly be summarised as a two-part nested *MVC* (*Model-View-Controller*; see Appendix C) structure. Looking at the class diagram (see Figure 3, also Section 5.1), starting from the right, the system involves *FEM* analysis of a structural model generated by solving a parametric and associative design ('PAD') model definition. This part of the program is computationally intensive, and is therefore structured as an asynchronous job queue. Moving to the left, the system provides interaction by allowing the manipulation of the PAD *resources* through a *REST API* (*Representational State Transfer*; Fielding, 2000). This part of the program consists of an relatively ordinary object-relational model with *CRUD* (*Create, Read, Update, Delete*; see Appendix C) operations, and implements *asynchronous job queue* pattern (Richardson, 2007) for PAD solution jobs.

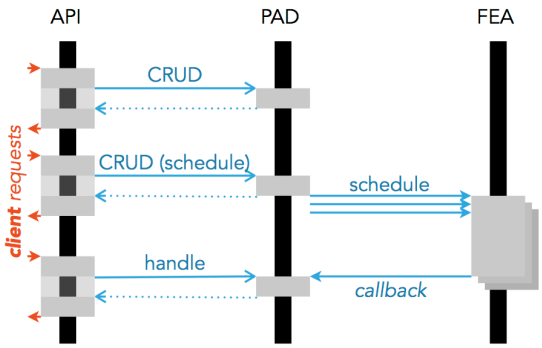


Figure 4: A sequence diagram showing *CRUD* manipulation and asynchronous scheduling operations.

The way the system works is shown by its sequence diagram (see Figure 4) and the accompanying flow chart (see Figure 5). Interactions with the user are represented by the red arrows on the left side. The top-most sequence represents the CRUD operations on the PAD resources through the REST API. This sequence is likely repeated several times for different resources as the user is busy creating the parametric and associative model.

The second sequence from the top represents the job scheduling operation. In the REST style, this takes the form of creating a job resource. The creation of the job schedules a series of asynchronous of computational tasks on a pool of worker queues, which will eventually complete. The created job itself can be polled periodically to check whether it has completed.

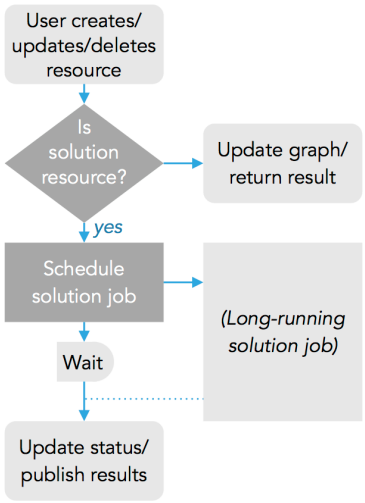


Figure 5: A flow chart showing the two basic interactions with the system: CRUD operations on PAD *resources* and scheduling *solution* jobs.

This page is intentionally left blank.

Preface

This is the final report of the Master's thesis of Bastiaan van de Weerd on the development of a client-server parametric and associative design modelling software system for conceptual structural design. This research is part of a collaboration between the faculty of Civil Engineering and Geoscienes of the Delft University of Technology, the BEMNext Laboratory, and Arup Amsterdam.

The graduation committee consisted of the following members:

Prof. dr. ir. J.G. Rots
J.G.Rots@tudelft.nl
+31 (0)15 278 4490
Delft University of Technology
Faculty of Civil Engineering and Geoscienes
Department of Design & Construction
Section Structural Mechanics

Dr. ir. J.L. Coenders
Jeroen.Coenders@arup.com
+31 (0)20 308 8500
Arup Amsterdam
Delft University of Technology
Faculty of Civil Engineering and Geoscienes
BEMNext Lab

Ir. J.W. Welleman
J.W.Welleman@tudelft.nl
+31 (0)15 27 84856
Delft University of Technology
Faculty of Civil Engineering and Geoscienes
Department of Design & Construction
Section Structural Mechanics

Ir. A. Rolvink
A.Rolvink@tudelft.nl
Arup Amsterdam
Delft University of Technology
Faculty of Civil Engineering and Geoscienes
BEMNext Lab

This page is intentionally left blank.

Table of contents

List of figures

This page is intentionally left blank.

1 Introduction

At a high level, the goal of *StructuralComponents* is to guide and capture the early-stage, conceptual design process in structural engineering. That is, it is a tool that allows the engineer user to build a structural concept. It tries to augment and complement the intelligence and creativity of the engineer. Simultaneously, it strives to capture that process and its decisions, and act as an information container for the resulting design. At this high level, the tool follows the method of the *Conceptual Story* (Coenders, 2011): the idea of a design justification in the form of a narrative combining analytical methods and computer tools with—importantly—reasoning and thought models, in an attempt to deal with imprecision and uncertainty.

The high-level idea of StructuralComponents is given shape by a series of MSc theses and PhD-level research projects, which all revolve around *parametric and associative design* ("PAD") modelling tools with integrated structural analysis. This MSc thesis is the third in the series, following Breider (2008) and Rolvink (2010). Rolvink is working on a doctoral thesis, *"Enhancing building safety assessment by computation"*, further developing the idea itself and incorporating methods of assessing structural integrity. Related to this research is the doctoral research of Coenders (2011)—which revolves around establishing a comprehensive infrastructure consolidating many previously existing and new concepts around the concepts of parameters and association.

The specific issues addressed in this thesis are threefold. First is the previously underdeveloped adherence to the available theory on the engineering design process, specifically its early stage. Second, the lack of general applicability of the employed *super-element method* of analysis. Third, the plugin-based (see Appendix C) software architecture, which acts restrictively from the perspective of information sharing and the overall goal of acting as a container for the captured design process.

This thesis is accompanied by a prototype software application. This prototype implements the proposed solutions to the defined problems to a degree necessary for their illustration and evaluation. The prototype presents a solution exploration concept which uses automatic parameter variation and data analysis. It is further reimplemented as multiple applications separated along a client–server architecture. The server application incorporates the PAD solver and the integrated FEM analysis engine.

1.1 Background

The research around StructuralComponents is based on the ideas proposed in the research around *Structural Design Tools* (Coenders & Wagemans, 2005; Coenders, 2008).

1.1.1 STRUCTURAL DESIGN TOOLS

The approach that originated from the research around *Structural Design Tools* ("SDT") identified the insufficient suitability of common CAD and analysis software to support the design process as a whole. The research reveals that while the design process does include the analysis and production stages that such software supports, it actually extends further into unsupported areas. The proposed ideas concern the goals of new design-suitable software, and its scope and development principles.

The SDT approach proposes that software packages should ultimately serve the engineer by providing support in design justification in order to inspire confidence. From this perspective, packages should be transparent and allow the engineer user to easily define, review and edit design solutions.

From a technical perspective, the SDT approach proposes that individual packages should strive to serve a few purposes well, rather than trying to encompass the entire design process. To this end, packages should use transparent methods and open data structures in order for its output to be compatible with and easily carried over to other packages, built for other purposes.

1.1.2 HISTORY

From its inception, StructuralComponents has revolved around creating software tools that use PAD modelling to let the user define the logic of a structural design. The tools simultaneously analyse the performance of the resulting structure to give immediate feedback using an information dashboard. Both previous theses were accompanied by prototype tools that accommodate the early-stage design process through the distinct phases of creation, analysis, and presentation.

2008 VERSION

Breider identified a problem stemming from difficult communication and the incompatibility of tools between different building industry disciplines. This thesis attributed the problems to the ongoing specialisation of these disciplines into separate professions, each with their specialised methods and tools. Specifically, Breider argued that in early-stage communication, the tools of the structural engineer were unable to work as fast as the methods of the architect. He proposed the combination of PAD modelling and schematic structural analysis methods as a solution.

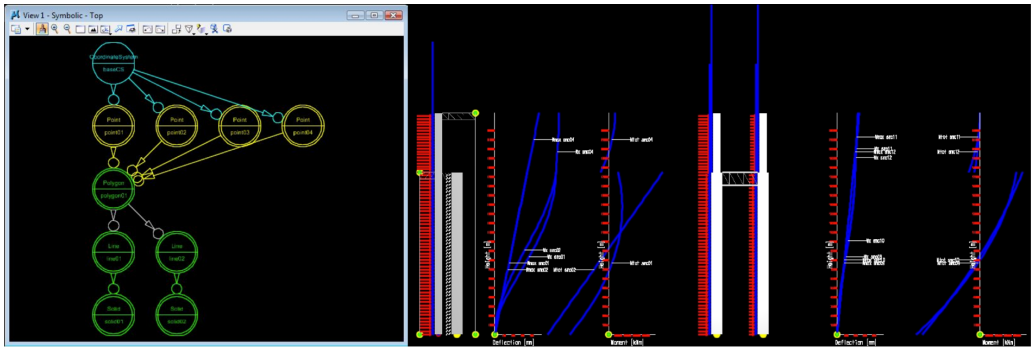


Figure 1.1: The modelling and dashboard interface of StructuralComponents 1.0
(image courtesy of Breider, 2008).

The prototype accompanying this thesis was a plugin for the Bentley GenerativeComponents (Bentley, 2013) software package. The package itself provided a PAD modelling interface with mathematical and geometrical modules, to which the prototype added a specialised structural module. The plugin was accompanied by a dashboard from which performance results could be read visually.

In conclusion, Breider upholds the choice for PAD modelling, which enables quick and easy composition of a structural concept. He further states that (near) real-time analysis is essential for design efficiency, and notes the specific value of comparing multiple structural variants, or load variants within the same model. Finally, he recommends implementing more structural versatility, and that additional effort be put into the dashboard and its visualizations.

2010 VERSION

Rolvink continued with the high-level objective of creating tools for the early conceptual design stage. She incorporated the *Conceptual Story* and the shifting balance between design freedom and availability of information, arguing that the tool would bring information to earlier stages, enabling better decisions at a time when the design is less constrained. She pointed to the gap between simple rules-of-thumb calculations and complex analysis as an opportunity for tools like StructuralComponents.

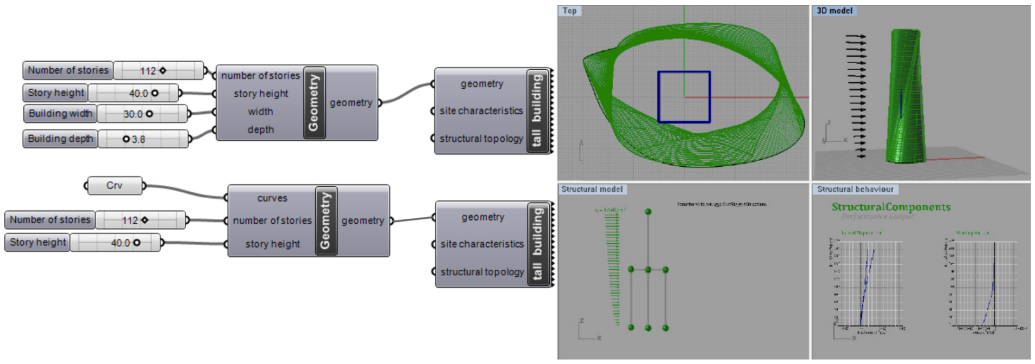


Figure 1.2: The modelling and dashboard interface of StructuralComponents 2.0 *(image courtesy of Rolvink, 2010).*

The prototype accompanying this thesis was written as an independent analysis library and separate bridge libraries for plugging into and visualising results in any suitable third-party software package. A bridge library is implemented to an advanced degree to plug into the *Rhino Grasshopper* (McNeel, 2013) package (see Figure 1.2). The analysis library implements an engine based on the *super-element method* (Steenbergen, 2007).

Rolvink restates the goal of the tool as augmenting and complementing the engineer. She stresses the importance of allowing for simple and efficient models, with a degree built-in design knowledge, and a focus on key drivers of a design concept. Rolvink states a broader objective of incorporating the research in a multi-disciplinary framework, and recommends including multi-criteria optimisation methods.

1.2 Problem definition

As described in the previous, the ongoing research around StructuralComponents revolves around creating more suitable structural design tools for the early-stage design process. This thesis proceeds under the umbrella of this general problem definition, which will be further described in the next section. On a concrete level, this thesis specifically addresses further the identified problems of adherence to design theory, general-purpose analysis and restrictions of the previous software architecture. These are described in the section after the next.

1.2.1 GENERAL PROBLEM DEFINITION

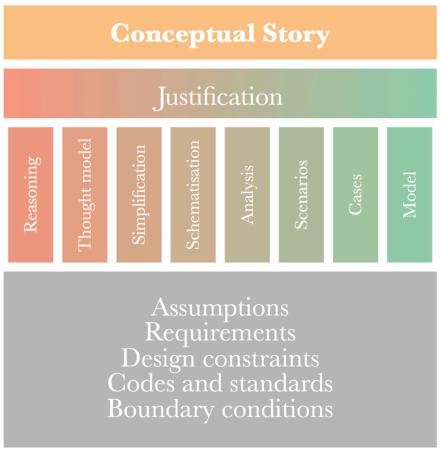


Figure 1.3: The Conceptual Story, envisioned as a structure basing a justification on pillars of early-stage methods and tools which each deal with imprecision *(image courtesy of Coenders, 2011).*

As argued and referenced in the preceding, the general problem that the ongoing StructuralComponents project revolves around is:

> ***The software tools available to the designing structural engineer insufficiently support the structural design process, and specifically its early stage.***

On a high level, the approach taken to solve this problem follows the strategy of the *Conceptual Story* ("Story"). The Story is (as defined) the idea of a design narrative based on a justification that combines the methods and tools available at the conceptual design stage. The term 'narrative' acknowledges the uncertainty and imprecision at this stage. It represents the goal of defining a design in such a way that it establishes an early qualitative framework that still encompasses the quantitative decisions of later, more certain stages. The referred methods and tools are reasoning, thought models, simplifications, schematisations, cases, scenarios, models, and analysis (see Figure 1.3).

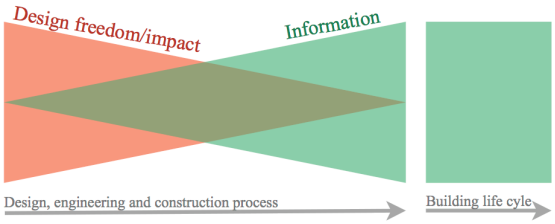


Figure 1.4: The impact of decisions is generally largest in the early-stage design process, while information is still lacking. This is reversed in later stages, which translates to higher costs (image courtesy of Coenders, 2011; in turn adapted from MacLeamy, 2004).

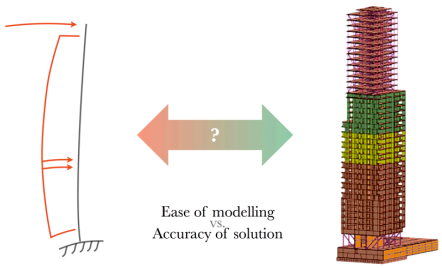


Figure 1.5: There exists a gap between simple rules-of-thumb methods and complex analysis software *(image courtesy of Wagemans)*.

The Story addresses the problem of designing with early-stage uncertainty and imprecision. At this stage, information is still largely lacking, while the decisions made here have a large impact on the eventual outcome of the process (see Figure 1.4). The lack of information is to be expected, but could be improved by suitable software tools at this stage. The imbalance is reflected in the gap between simple rules-of-thumb methods and complex analysis software (see Figure 1.5). Similarly, it is reflected in the way CAD software is suited to produce the outcome of a design process, but not necessarily its early stages. These imbalances pose opportunities for improvement.

1.2.2 SPECIFIC PROBLEM DEFINITIONS

REGARDING DESIGN THEORY

The MSc theses preceding this thesis followed the high-level views on structural design put forward by the work of Coenders and Wagemans (2005, 2008). However,—

The StructuralComponents tool lacks adherence to formal research on design in engineering and closer views on the design process.

From various publications on the topic of early-stage design processes in general, and those in engineering specifically (see Chapter 2), it can be distilled that design is generally an iterative process, each iteration of which is a cycle involving phases of generating new solutions, and assessing their performance. It can be argued that there is a *characteristic design cycle*, which in the early stage clearly shows a strong emphasis on the generation, or synthesis phase. This emphasis is insufficiently reflected in the previous implementations of the tool.

REGARDING ANALYSIS METHODS

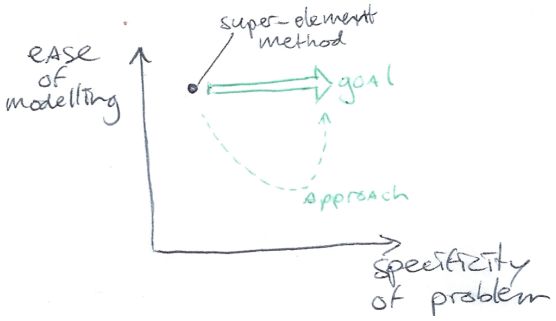


Figure 1.6: The super-element method is well-suited for specific designs, but not for general-purpose analysis. To come closer to the goal of general applicability, the inclusion of true general-purpose methods can be considered as a first step, at the initial cost of ease of modelling.

As described, the previous theses proposed an analysis system based on the *super-element method*. Though efficient, the method is limited to specific building types—tall structures, as previously implemented. Both theses include a recommendation for more structural versatility in further research, alluding to the following fact:

The previously-used analysis engine base on the super-element method has limited versatility and cannot be easily extended to other building typologies.

This method schematises a structural model as a series of large elements, each of which model the behaviour of a larger building part. For example, a building with an outrigger would consist of the lower and upper core elements, an outrigger element, and facade elements, each with specific behaviour (axial, bending, shear). While this method is fast and accurate, only a limited number of building types can be properly schematised for it (see Figure 1.6; the previous research focused on tall buildings).

REGARDING SOFTWARE ARCHITECTURE

The software architecture of the previous prototypes was designed for use as a plugin for other software packages. It serves its purposes well, but is entirely dependent on the host software.

The previous plugin-based software architecture is restrictive in light of the Conceptual Story and the Structural Design Tools approach.

The previously used host software can handle open data structures, and to a certain extent provides transparency within its plugin architecture, and through additional documentation. But, being closed-source, such transparency is limited, as is flexibility beyond the rigidly defined plugin architecture. Furthermore, the software runs on workstations, raising issues of scale. On the other hand, in spite to these issues the tool should continue to work in combination with specialised software packages, most of the functionality of which this tool

should likely never wish to copy.

1.3 Objectives

Based on the previously-defined problem definitions, a number of objectives were set. Regarding the overarching *general* problem definition, the general objective of this thesis is:

> ***To further improve the StructuralComponents approach to the early-stage design process.***

Such a high-level objective may be too non-specific to be achieved practically, or judged on its merits. However, it exists to guide decisions of the further, tangible objectives. These are based on the *specific* problem definitions.

In this early stage, it it proposed that in exploring possible design solutions, only low precision is required. Instead, this stage requires high flexibility. Later stages may determine the performance of a specific solution, but this stage should focus on seeking a solution for a desired performance. This stage, more than those following later, requires creativity and generation under limited constraints.

AN INCREASED EMPHASIS ON THE SYNTHESIS PHASE

It is proposed to further emphasise the synthesis phase, in which design solution alternatives are created. The parametric and associative design ("PAD") modelling paradigm is deemed suitable for this purpose, and it is proposed to extend it with an abstraction system. That is, a system of automatic parameter variation around the input of the user to contextualise the choices using statistical analysis and data visualisation.

THE DEVELOPMENT OF A FINITE ELEMENT METHOD ANALYSIS ENGINE

It is proposed to develop a Finite Element Analysis ("FEA") engine for the general-purpose analysis cases to which the *super-element method* is not suitable. To fit within the scope of this thesis, this engine will practically be limited to linear wireframe models with axial bar and Euler-Bernoulli bending elements. It will further make use of stiffness matrix symmetry and sparsity in order to accelerate computation.

THE REIMPLEMENTATION AS A CLIENT–SERVER SOFTWARE ARCHITECTURE

It is proposed to separate the functionality of the prototype software along a client–server divide. The server prototype contains the FEA engine, and additionally a PAD solver. It will be accessed through an REST-style HTTP API.

2 Design theory

The rationale for many of the choices and much of the reasoning of this thesis lies within the available research on design theory, especially that on design in engineering. Design is a very broad and complex topic, and accordingly has many definitions, plenty of which are conflicting. Although this thesis deals exclusively with the narrower topic of design within the engineering discipline, various conflicting views still remain. However, it is not the goal of this thesis to develop or unify these theories.

An effort has been made to put forward an internally harmonious collection of views on design theory. Section 2.1 discusses the views individually and attempts to show their coherence. Section 2.2 presents a number of maxims which form the guiding principles for the implementation choices of this thesis. To provide the argumentation for those choices, the following chapters will explicitly mirror the relevant aspects with the presented maxims.

2.1 Views on design in engineering

A collection of views has been chosen in service of the thesis argumentation. Obviously, this collection is only a subset of the entire body of design theory. The choices were made to serve the thesis requirements itself, as governed by the direction of the preceding theses and the newly introduced concepts. Views will exist that invalidate part of the argumentation, which will be discussed to a certain extent, where relevant, in the following chapters.

2.1.1 SYSTEMS DESIGN AND MANAGEMENT OF CONSTRAINTS

Among the various views, the common view of the design process is that its goal is to try to find the best solution satisfying a certain problem. This wording pertains to the view of Systems design (Chapman, 2001), which has specific definitions for the terms *solution* and *problem*. Similarly, the term *best* carries much implied meaning, as does the critical verb *to find*. The problem will generally be uniquely defined in terms of ambition, environment and financials (Dorst, 1995).

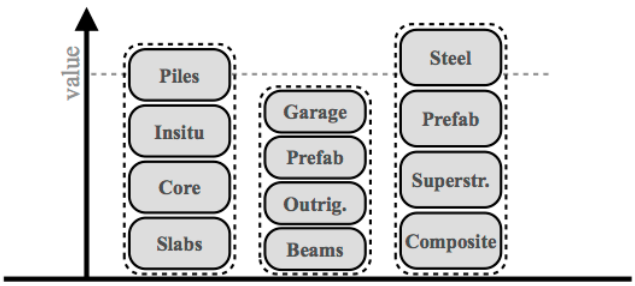


Figure 2.1: Solutions are sets of decisions and have variable value beyond merely satisfying requirements.

In the Systems design view, a solution is a set of decisions or choices (see Figure 2.1). A problem is a set of requirements and constraints. The problem can be satisfied by multiple solutions, of which satisfaction is easily verifiable. Importantly, some solution will have greater value beyond the point of satisfaction than others.

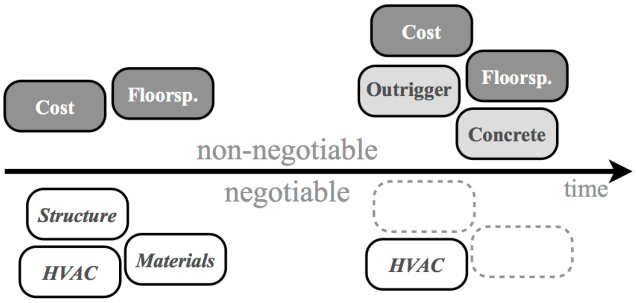


Figure 2.2: The design process involves gradually transforming negotiable constraints into non-negotiable constraints

Another view sees design as the management of constraints (Dini, 2009; see Figure 2.2). The design process starts with a few non-negotiable constraints, which specify the requirements. Furthermore, there is a large number of negotiable constraints. The design process involves the gradual reduction of this negotiability by way of making decisions (simplifying negotiability as either existing or not; see Box 2.1).

> The opposition of negotiable versus non-negotiable constraints is used here for simplicity, fully realising that in reality aspects will always remain negotiable to a certain extent (except perhaps laws of nature). Non-negotiable constraints may therefore be interpreted as *practically non-negotiable, i.e. difficult and expensive to change.*

Box 2.1: A further note on *negotiability*

The two views complement each other. The *solution* in the former is analogous to the *set of non-negotiable constraints* in the latter. Systems design puts forward the condition (satisfying the problem), while the management of constraints suggests the process (each state in the reduction process). The former introduces alternative solutions, while the latter contains differently negotiated states.

In combination, the two views suggest a process starting at the non-negotiable initial requirements state. This state represents both the design problem and the trivial solution which satisfies by definition. However, it is trivial since it rephrases the problem but is otherwise unspecified. The next state is further specified: it is more detailed in its negotiated decisions. But it exists alongside alternative states that are specified to the same extent, but different in its negotiations.

> The example of a building illustrates the combined view. The problem requires a building with certain characteristics, like some imagined *capacity*. The trivial solution is the question rephrased as an answer —'building which satisfies *X* capacity?' as 'building satisfying X capacity', *building* being a monolithic concept. A second state redefines the first—'building consisting of components A and B satisfying X capacity', where *building* is no longer monolithic but consisting of components. A third state redefines it again—'building consisting of components A, in turn consisting of components A1 and A2, etc…'.

Each subsequent stage consists of a wider array of possible alternative states, each negotiated differently, each specified to a further extent than the previous stages. As decisions are made, the scope of subsequent decisions is limited. Meanwhile, the set of requirements expands. The end product is a series of decisions forming a solution of non-negotiable constraints that specify an identity, the left-over integrity of the concept.

> Understood in terms of structural engineering, decisions relate to concepts such as structural layout, topology and member sizing. The initial set of requirements cover concepts such as loads, architectural shape or floor area ratio. Further requirements will be introduced during the design process, such as material stresses or geometry clashes.
>
> Returning to the example of the building, the product of the design process is a building satisfying the capacity *X*. One level down, the overall structure and architecture (which form the building) satisfy the capacity, and their own requirements. Another level down, the core and floor substructures together (which form the overall structure), together with the facade and routing concepts (which form the overall architecture), satisfy the capacity, the second-level requirements, and the new requirements at their own level.

The idea of the *Conceptual Story* has been a topic of the theses preceding this one (Breider, 2008; Rolvink, 2010; Coenders, 2011). It is the overarching narrative of a design solution on a high level, covering all the major design decisions but leaving many of the low-level details to be resolved in later stages. This relates directly to the solution states described above, of which each iteration is a further specification of those preceding it.

The design story is the solution state at the end of the early stages of the design process. It must be a solution that satisfies the problem, but which will only minimally clash with detail decisions later on in the process. That is, early decisions must leave enough room for sensible detail solutions. The narrative is formed by the body of assumptions, logic, calculations and experiments underlying the solution, which together form its justification (see also Figure 1.3).

The abstract design problem in structural engineering falls in the category of problems for which no known efficient way to find a good solution exists (Chapman, 2001), but for which it is relatively trivial to assess whether an individual solution is good. Problems in this category are referred to as NP-complete problems, and the complex and creative human decision-making process has proven very valuable in tackling them (Wigderson, 2009).

2.1.2 ITERATION IN DESIGN

The sequence of solutions states described in the previous section already alludes to the strongly iterative nature of the design process. Iteration is defined as cycles of proposal, testing and modification of an evolving design (Smith, 1998). Activities within a cycle can loosely be categorised as analysis, performance evaluation; or synthesis, alternatives generation.

The scale of a cycle can vary greatly in complexity and time length, although reductions in scale are often advanced by decomposing a problem into smaller pieces. As iterations progress, the duration of synthesis activities tends to decrease while the duration of analysis activities increases. This emphasis shift mirrors the shift in influence of design decisions, which is high early on but decreases along with the level of detail, and cost of making changes, which starts out low but strongly increases along with complexity (see also Figure 1.4).

The iterative nature of the design process can be both evolutionary and revolutionary. In the evolutionary sense, the process mirrors optimisation (Simon, 1955) in that it deterministically searches for an ever better solution in light of an established objective measure. In the revolutionary sense, the process continually overhauls the solution, redefining it on a deeper level of detail as the interdependent subcomponents constituting previous components.

2.1.3 CHARACTERISTICS OF EXPERT DESIGN PROCESS

One specifically broad study has revealed a range of common characteristics of expert design process (Sims-Knight, 2005). That is, these are characteristics of the individual processes its successful practitioners. Most of

these characteristics assume a high degree of repetition, pointing towards iteration as a key aspect of design.

- Taking **time to understand** the problem: Much care goes into meeting with clients and repeatedly discussing the interpretation of the design brief and outcome scenarios until confidence is gained into the true goals.
- **Frequent testing**: Right from the start, quick sketch designs are produced in order to eliminate unfruitful directions early. Throughout the process, the performance of intermediate solutions is frequently assessed.
- Creating **alternative concepts** from the beginning: Breadth in options is often preferred to depth in one specific solution. There is an understanding that there will be multiple satisfying solutions.
- Considering **feasibility and trade-offs**: Continuing from the alternative concepts principle, there is an understanding that additional objectives will come into play when considering the feasibility and trade-offs of and between multiple solutions.
- Using both **top-down and opportunistic** decision-making: There is a realisation that at times it is appropriate to follow a structured approach to design, from a top-down perspective. However, at other times it is more appropriate to opportunistically deal with the situation at hand and temporarily avoid structure.
- Considering **components' interrelations**: It is understood that a system consists of multiple components, which are variously interrelated. Interrelations lead to complexity and it is often advantageous to compartmentalise the problem early to reduce their total number.
- **Reviewing and reflection** in terms of requirements: The performance of solutions is continually brought back to the design requirements and objectives in order to avoid losing sight of the goals.
- **Prototyping**: The design process is structured around continuous prototyping to ensure that performance of a solution always remains assessable.

2.1.4 QUALITIES OF THE DESIGNING ENGINEER

In contrast to the characteristics of the expert design process, another study reveals the success-promoting personal qualities of the designing engineers themselves (Fricke, 1996).

- Good **spatial imagination**: Continually being able to keep track of spatial layout and the spatial interrelations between components in the mind, before producing a directly assessable spatial representation.
- **Routine sketching**: Naturally extending spatial imagination, obtaining a sense of layout and interrelations through representation of the design object when imagination falls short.
- Solid **engineering knowledge**: Solid knowledge in the engineering field at hand, and typically in the broader context of related sciences.
- High **heuristic competence**: The ability to derive dependable symbolic meaning in cursory observations in anticipation of more essential properties of a design that are more difficult to determine.

Analysis of the design goals is a specifically critical part of the process. Problems will often be formulated incompletely or excessively problems, respectively with present but unstated or over-emphasised or conflictingly described requirements. In summary, tactics for successful goal analysis are:

- Thoroughly **scrutinising 1the problem** structure: Uncovering the full set of requirements for incompletely formulated design problems, focusing on the structure of the problem, and making these requirements concrete.
- **Summarising** the information: Reducing the full list requirements of extensively formulated design

problems to the essential set through summarising, abstracting, prioritising, and disregarding (esp. subjectively over-emphasised objectives).

- **Not suppressing** solution ideas: Taking note of early ideas formed before the objectives are finalised, and later returning to them for reassessment.

The search for solution, too, is critical. Three main strategies that can be observed here are excessive expansion of the search space, balanced search, and narrowly restricted search. In summary, tactics that for successful solution search are:

- **Varying alternatives** at least for the main functions: Creating alternative solutions on principle, avoiding being satisfied by an initial idea. However, generating too many alternatives proves unsuccessful.
- **Generating and reducing alternatives** in a balanced manner: Expansively searching for variant solutions, but eliminating alternatives again when impeded by a lack of oversight. However, reducing to an unreasonably restrictive extent proves unsuccessful.
- **Evaluating solutions** frequently and according to their level of abstraction: Assess the performance of variant solutions often, but only in accordance with their level of concreteness, without getting lost in considering vaguely specified details.

2.1.5 CONFLICTING CONSTRAINTS AND MULTI-DISCIPLINARITY

The preceding sections have already implied that multiple constraints will typically apply to a design problem. Conflicting requirements have also been mentioned. Often, imposed design constraints will conflict others, in which case no solution will exist that fullfills every requirement to the full extent. In many cases, there will not exist a solution that satisfies even all constraints and requirements. Such cases call for compromising solutions.

The aspects of a design problem are often distributed over a multitude of disciplines. Design decisions often originate from the perspective of one discipline, based on an a performance benefit within that perspective. On its own, one perspective will only consider local aspects if there is an indication of local performance benefit. Such local benefit will typically be maximised to available resources. Wherever an aspects plays no role in local benefit, the possibility of global benefit (or benefit elsewhere) will not be considered.

Especially in the preliminary design it is both critical and still possible to coordinate the process such that global benefit is not overlooked (Klein, 2006). Available strategies to achieve this include rearranging components to untangle disciplines and minimise shared resources, and locally incentivising globally beneficial design aspects. As the design process moves on and decisions become more detailed, standard and established ways will become more prevalents and less opportunity remain to coordinate local aspects in service of global benefit.

The preliminary design stage ideally explores the problem space and finds global benefit early. It establishes a sufficiently untangled intermediary design solution that allows subsequent local optimisation to the extent that it does not influence global benefit.

2.2 Characteristic design cycle and maxims

Based on the preceding sections, a characteristic design cycle was distilled (see Figure 2.5), based on which a total of nine maxims of design theory were defined. It is proposed that the implementation decisions leading to a design tool, this being the product of this thesis, should reflect these maxims. They were established to be relevant to computational design tools only, an so do not aim to cover the design process as a whole. The maxims are subdivided in three categories of three maxims each, respectively concerning the inherent nature of the design process, strategies that promote its success, and tactics that promote its success.

In the following, the term ‘tool’ refers to a computational design tool for structural engineering, such as

StructuralComponents. Regarding the nature of design, the following maxims are proposed:

- The design process is **iterative**: a tool should capture the iterative process as a succession of evolving designs—perhaps not every trivial change but certainly those moments where a design decision was made to focus on one solution and forego another.
- Each iteration in the design process consists of phases of **synthesis, analysis, and inference**: a tool should provide means to both generate design solutions and measure their performance, as well as make explicit what are the key aspects driving the design.
- There is a **shift in emphasis** of iterations in the design process from synthesis to analysis: a tool focusing on the early stage of the process should emphasise the rapid generation of alternatives design solutions, to be analysed quickly using simple, perhaps less accurate methods.

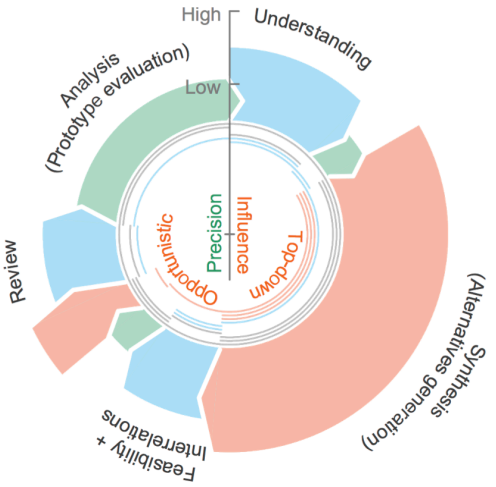


Figure 2.5: The characteristic design cycle: the early stage design process shows an emphasis on and high influence of *synthesis* (red), a de-emphasis on and low required precision of *analysis* (green); *inference* phases are found throughout the cycle.

Regarding strategies that promote success in the design process, the following maxims are proposed:

- Multiple **alternative solutions** to a design problem should be considered: a tool should allow the easy generation of design alternatives, to be evaluated simultaneously so that their performance can be compared.
- Alternative solutions should be **generated and eliminated** in a balanced manner: a tool should encourage its user to make quick progress while also providing trust that no potentially good solutions were missed.
- Solutions to a design problem should be **evaluated frequently**: a tool should provide the means to continuously evaluate the performance of solutions, to test whether they meet the requirements.

Regarding tactics that promote success in the design process, the following maxims are proposed:

- Indicators of solution performance should be encoded into **heuristics in terms of the problem requirements**: the tool should provide a method to declare what indicates good performance, to be encoded in heuristic performance functions which can be evaluated directly.
- When appropriate, there should be **opportunistic deviations from a top-down design process**: a tool should make it easy both to make the changes required to explore accidental ideas, and also to track back if and when such ideas prove unsuccesful.
- Solutions should frequently be **imagined, sketched, or prototyped** according to complexity requirements: a tool should provide visualisations and impressions of design solutions that closely resemble their real-world counterparts, in order to quickly evaluate performance at a glance without explicit mathematical definitions.

3 Concept and methodology

This thesis proposes modifications to the StructuralComponents design system, of which the major goal is an increased emphasis on design exploration. The modified system primarily relies on automatic generation of design variants and subsequent interactive visualisation of the resulting performance data. It prioritises exploration of the problem over accuracy of the solutions. It contains a design methodology based on posing measurable questions at each iteration, to which the data must provide an answer.

ADHERENCE TO THE CHARACTERISTIC DESIGN CYCLE

The proposed system alternates between composition and abstraction phases. During the composition phase, users synthesise a solution to the design problem: they compose a parametric design concept out of various functional components. During the abstraction phase, they analyse the full potential of the solution: they abstract over the parameters to assess the design's many value-substituted manifestations. The single design solution intuitively created in composition is viewed broadly, over the possible range of its parameters, in abstraction.

The composition and abstraction phases transition back and forth through inference phases. These phases are characterised by reasoning and deduction, relating the proposed design solutions and obtained performance data back to the problem constraints and requirements. The transition from composition to abstraction features the definition of metrics from which the value of a solution can be derived. The transition back features the deduction of the most promising parameter ranges, and the decision to refine and explore further—or to track back, if the desired performance is not found.

After the second inference phase, where the decision is made to refine or to track back, the system once again transitions to a new composition phase. This phase again segues into inference (defining metrics), then abstraction, and lastly back to inference (deducing potential). This repeating sequence follows the cyclical and iterative nature of the design process. Each new iteration might feature exploration of solutions in a new direction, or refinement of a direction showing promise. The resulting process shows a clear path from the last solution back to the initial problem statement, along the dead-end branches of rejected alternatives.

IMPLEMENTATION PROPOSAL AND PROTOTYPE

In terms of user interaction and use cases, the system firmly leaves the designing engineer responsible and in charge. It strives for full transparency in its inner workings and generated data, making use of the intuitively understood parametric and associative modelling paradigm and generating alternatives only within user-defined ranges. It is decidedly unintelligent with regard to supporting the design process, following the *"garbage in, garbage out"* principle of allowing both the sensible designs of the experienced designer or the meaningless results of uninformed input.

In terms of implementation, a prototype of the system was built along a client-server separation between user interface and computation. The separation yields greater potential regarding both usage on the client side, and performance on the server side. It must be noted that the client-server separation sacrifices the simplicity of the previous *StructuralComponents* prototypes. However, it ultimately allows multiple and weaker user applications and devices to take advantage of more powerful server hardware and infrastructure. Furthermore, it offers the beneficial network effects of centrally collecting design data for various problems.

OVERVIEW

The proposed system strives to adhere to the interpretation of design theory described in this thesis' introductory chapters. This chapter attempts to make the case for that adherence and gives the arguments for further implementation choices. Section 3.2 provides the theoretical background for the implementation

choices detailed in Section 3.3. The theory and implementation of the structural analysis contained within the system, and the software and system architecture shaping it are respectively dealt with in the following two chapters.

3.1 Theory

Design in engineering, at its core, entails the basic problem of finding a good solution to a given requirement, according to a certain set of metrics, respecting another set of boundary conditions. It could be seen as a virtually free-form process with an ambiguous end goal, built on foundations and assumptions that often change during the process itself. The performance of its resulting solutions is easy to assess, though the paths to those results are not usually laid out clearly ahead (Chapman, 2001).

The design process as a whole shares these characteristics with one of its possible subprocesses, optimisation. For that reason, it is common to see tools being equating optimisation with the design process itself. This may be fully justifiable in many cases, but least likely in that of the early, conceptual stage. Where the goal and product of optimisation is the final solution to the design problem, that of the conceptual stage is exploration of the problem space and familiarisation with the possible design solutions it holds.

A good aim for the designing engineer during the conceptual stage is to gain a broad overview of the collection of possible solutions (Fricke, 1996), while many details of and surrounding the design remain unknown; many design considerations remain ambivalent, not easily expressed in numerical values; many competing metrics exist together, and priorities remain unclear.

A good aim is to get familiar with the various parameters that govern a possible solution's behaviour: to see how they are individually related, how they influence the metric, and how close they come to crossing any boundary conditions. The goal and products of the early, conceptual stage of systems design should not be a final solution, but the insights gained on the path towards one, having travelled down and backed out of various branches.

3.1.1 SYNTHESIS: PARAMETRIC AND ASSOCIATIVE MODELLING

As was the case with previous iterations of the StructuralComponents project, the system proposed in this thesis is based on the parametric and associative modelling paradigm. This paradigm revolves around the modelling of design objects—geometrical, structural, or otherwise—through the visual composition of a network of parametric logic. In essence composing a family of visual programming languages, and generally adhering to the flow-based programming paradigm, the various examples of the parametric and associative modelling paradigm available within the architectural and engineering industries have become increasingly prevalent over the last decade.

3.1.1.1 Background and related work

Often cited as the origin of the term *parametric and associative design* is the early work on *GenerativeComponents* (Aish, 2005). Predating that work by some years are some explorations into the paradigm and computer-aided engineering methods in automotive and aerospace engineering (e.g. Aspettati, 2001; Ledermann, 2005). More recently, the paradigm has garnered success in the form of the *Grasshopper 3D* plugin for *Rhino* (McNeel, 2013). Furthermore, this thesis is part of the PhD dissertation *NetworkedDesign* (Coenders, 2011), which similarly proposes a design system encompassing parametric and associative modelling.

The parametric and associative paradigm closely resembles the *flow-based programming* paradigm in computer science (sometimes referred to as *dataflow programming*) in concept and typically in representation. This paradigm itself is described as "a programming language that defines applications as networks of 'black-

box' processes, which exchange data across predefined connections [and] can be reconnected endlessly […] without having to be changed internally" (Morrison, 1994). In similar terms (and simplified), parametric and associative models are defined as series of transformative components, through which a stream of input data flows and is transformed to output data.

The concept is related to the *map* and *reduce* higher-order functions familiar from *functional programming* (see Box 3.1; these are hereafter called *functors* to avoid confusion with other functions). The *map* functor takes an arbitrary transformative function and a list as input, and returns a list of the results of that function having been applied to each individual element in the list. The *reduce* functor similarly takes a function and a list, and additionally (and optionally) an initial value. This functor proceeds to apply the function to each subsequent list element and its own previous result, accumulating the list elements into a single return value.

Both *map* and *reduce* are known as such in e.g. *Clojure*, *Erlang*, *Python*, *Javascript*; both are included under different names in e.g. *C#* and *C++*. Furthermore, these functors represent the core components of *big data* solutions such as *Google MapReduce* and *Apache Hadoop*.

Box 3.1: Typical *functional programming functors*.

Forming an example of a *map* functor is the operation of doubling each number in a list. Forming an example of a *reduce* functor is the operation of summing up each number in a list. Together, in mathematical terms, using the input list of numbers $[1, 2, 3]$,

$$\begin{aligned}&\text{reduce}(g(y,z) = y + z, \text{map}(f(x) = 2 \times x, [1,2,3], 0)\\ &= \text{reduce}(g(y,z) = y + z, [2,4,6], 0) = 12.\end{aligned}$$

In terms of data flow, the operation resembles the sequence of states

$$\begin{aligned}&[1,2,3] \quad \text{(input)}\\ &\rightarrow [2\times 1, 2\times 2, 2\times 3] = [2,4,6] \quad \text{(mapped)}\\ &\rightarrow ((0+2)+4)+6 = 2+4+6 = 12 \quad \text{(reduced)}\end{aligned}$$

Representing these operations visually the resemblance to both a parametric and associative model definition and a flow-based programming diagram should be evident (see Figure 3.1).

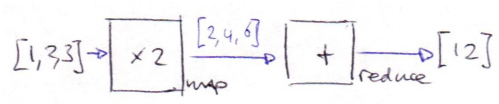


Figure 3.1: A visual representation of the described map-reduce operation

Design systems implementing the parametric and associative modelling paradigm generally feature a network of branching and conjoining data flows, a library of program objects which constitute the flowing packets of data, and model manipulation through the addition and removal of components and connections. In terms of advantages, as listed by Coenders (2011), these systems model the *logic* that generates models as opposed to modelling the models themselves, are flexible with regard to change, and provide insight by decomposing design logic to its essential arguments.

3.1.1.2 Argumentation for use

In terms of the proposed maxims which should be reflected in the computational design tool described in this thesis, the parametric and associative modelling paradigm provides primarily strategic advantages. Some caveats should be noted with regard to the tactical maxims.

> Repeated here for clarity, the strategic maxims are:
>
> - (IV) considering alternative solutions;
> - (V) balanced generation and reduction of alternatives;
> - (VI) frequent evaluation;
>
> and the tactical maxims:
>
> - (VII) heuristics as performance indicators;
> - (VIII) opportunistic deviations from a top-down design process;
> - (IX) frequent imagination, sketching, or prototyping.

The modelling paradigm is inherently generative—i.e. generates a design solution based on a network of logic. Obviously with regard to parameter variations, this nature makes the paradigm suited to the rapid generation of alternative solutions (IV). A single instance of a logic network can be used to generate wholly different solutions depending on the input parameters. One level of abstraction up, the paradigm similarly allows relative ease in the reconfiguration of its defined associations by reconnecting components.

As the paradigm is also inherently computational, its set of basic components can easily be used to further transform generated object properties into metrics for immediate evaluation (VI). Such a step—though its success requires a certain level of engineering knowledge from the user—allows the establishment of performance heuristics (VII). In the example of stress unit checks, the value and purpose of such metrics is self-evident. In more complex examples, where the available information may be limited, the designing engineer may find opportunities to express forward notions about actual performance as indicative metrics.

Benefitting from the generative nature itself and from the heuristic possibilities of the computational nature, the paradigm may enable a degree of balanced generation and reduction of alternative solutions (V). In support of this, additional facilities would be required for comparison of generated output, such as visualisations.

Some caveats must be noted with regard to the remaining tactical maxims. As noted by Davis (2011), it is not always possible to make quick changes to a parametric and associative model. Decomposing a design concept into its generative logical components is a top-down process. Changing some aspects of it—especially when those changes must be made in the middle or even at the start of a logic network—can be difficult, and may lead to other unforeseen changes. This may inhibit the proposed opportunistic deviations from a top-down design process (VIII), when the changes leading to those deviations cannot be made at the end of the logic network.

Similarly, the indirect modelling of the paradigm—i.e. modelling the logic, not the model itself—and associated decomposition step of a concept into logic components is not always easy and may not always be done at a whim. When this is the case, this indirection may remove the intuitive manipulation of a tactile and responsive sketch or prototype (IX). The paradigm may enable the user to logically construct an imagined design object, but may equally inhibit the user from making quick intuitive changes.

3.1.2 ANALYSIS: FINITE ELEMENT MODELLING

The analysis phase serves to measure the performance of solutions through simulation. Although in the early stage of the design process there is an emphasis on synthesis, or solution generation, it is important to determine whether those solutions satisfy requirements. Still, the analysis in this phase is not generally required to be highly accurate. An inaccurate, roughly estimated indication of performance will often suffice.

For the purpose of this thesis, a custom Finite Element Method analysis engine was developed. The theory

behind the engine is broad, and is accordingly discussed separately in Chapter 4.

3.1.3 INFERENCE: ABSTRACTIONS, COMBINATORICS & PROBABILTY, AND VISUALISATION

The inference phases serve to allow the designing engineer to attain knowledge about problem requirements and what they imply for the possible solution. In these phases, the engineer will determine which parameters are essential to, and what performance constitutes success of a solution. Furthermore, these phases will feature continuous checks of the performance of generated design solutions against the requirements.

SOLUTION SPACE

A specific choice for a design parameter can be thought of as a point on its respective domain of possible values. By extension, a specific combination of choices out of a set of design parameters—that is, a solution—can be thought of as a point on the total domain spanning all possible combinations. This domain will be referred to as the problem or design space—the former in the sense of opportunities for exploration, the latter in the sense of known regions.

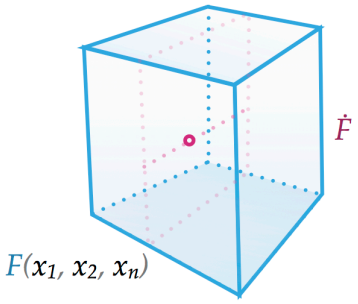


Figure 3.2: One solution in problem space

One perspective of this space—and its namesake—is that of an Euclidean space. In this perspective, each parameter corresponds to a physical dimension: two parameters form a plane and three form a volume. Furthermore, a design solution corresponds to a point on this plane or in this volume (see Figure 3.2). This spatial representation of the various design possibilities lends itself especially well to intuitive understanding—thanks to human spatial awareness (Victor, 2011)—and interactive manipulation—the goal of this thesis.

It is common to start the design process with a single starting solution of a set of interesting parameters (Fricke, 1996)—that is, for example, a point in the imaginary volumic design space of three parameters. Leaving two of the parameters fixed, the design problem can initially be explored by varying the remaining parameter; the design space explored along a single line. Using a computable metric that forms a good base for comparison, the various points along the line can assigned a value. Collectively, these values form a single broad, assessable view of a certain aspect of the design space.

ABSTRACTIONS

The transition from the concrete solution to the broad, collectively assessable view of the design space with respect to a free parameter constitutes an abstraction. Similar abstractions are possible between the broad view of a single parameter to that of multiple—in Euclidean spatial terms, transitions from linear to planar, or from planar to volumic.

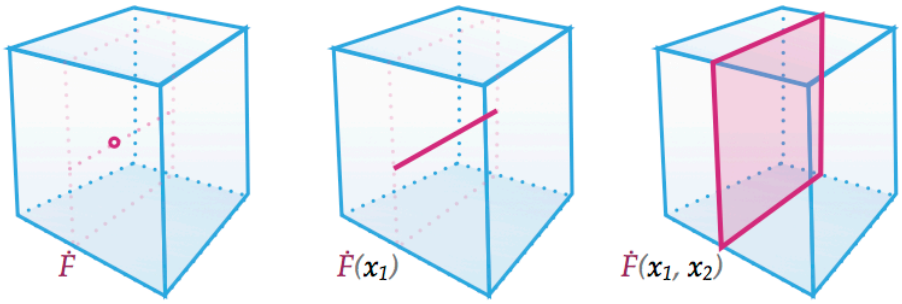


Figure 3.3: Various abstractions in problem space

Abstraction entails stepping from the narrow to the broader view, and back down to the narrow view—or possibly another. It allows for insights to be gained into the influence of a parameter on the metric, or the rate of change in that influence as the result of another.

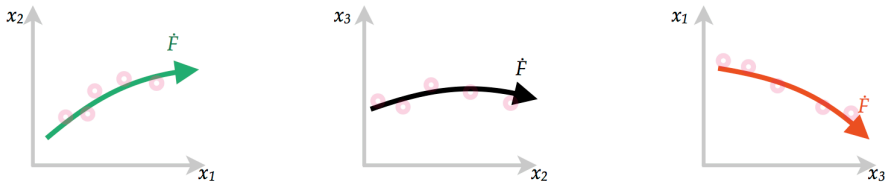


Figure 3.4: Parameter relationships

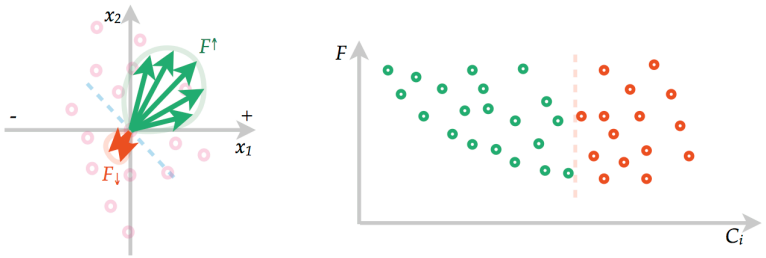


Figure 3.5: Metric sensitivity and constraint proximity

Looking at a single parameter, abstraction will reveal the course of the metric over the parameter's domain. Similarly for two parameters, but maintaining a constant value for the metric, abstraction will further reveal how they are related. Computing the metric values around a known point, the normalised vectors of parameter change will reveal the metric's directional sensitivities.

Through abstraction, insights can be gained into a metric's sensitivity to free parameters—the rate of change, the high points and the low—and consequently into the causes for the particular (un)suitability of individual solutions.

This thesis is based on an assumption that the best (see Box 3.2) method to gain these insights is use the intuition of the user as a starting point and reinforce it by verifying and contextualising using adjacent design space data. This assumption is further based on the premise that the user is best (see Box 3.2) knowledgeable about the problem, such that his or her intuitive guesses lead to sensible variations and consequently sensible data. However, it is proposed that by quickly verifying and contextualising an automatic system can inspire confidence and further exploration.

Verification can refer to the assertion that some particular requirement is or is not met for some design solution

(or parameter combination). That is, verification could be a passing unit check, asserting for example that a certain structure displacement is still acceptable. Verification can also refer to the assertion that an expected trend, relation or sensitivity does or does not exist. For example, it can confirm an anticipated inverse relation between the stiffness of an outrigger and that required of a building core.

Contextualisation can mean something like the latter, though it can further show that a particular trend, relation or sensitivity is dependent on yet another parameter. Referring back to the outrigger–core example, it can reveal that the bending stiffness relation further depends on the axial stiffness of the facade column. In this last case, such a dependency can pose an interesting new avenue for exploration in a next step.

> The first use of the term best 'best' is understood as the most suitable trade-off between feasibility in terms of ease of implementation and effectiveness in terms of correctly achieving the desired results in most cases.
>
> The second use of the term 'best' is understood as better suited to start exploring the problem in the right place using intuition than the heuristics underpinning any automatic system could feasibly be.

Box 3.2: 'Best'

3.1.3.1 Combinatorics & probability

The proposed system involves automatic variation around the parameter values specified by the user. Consequently, two important aspects of the system are the mechanism responsible for choosing these variations, and the considerations that should go into interpreting the resulting data. An important part of this interpretation is the visual representation (or visualisation) of the data itself, which is dealt with separately in the next section. This section will first discuss the variation mechanism in terms of its goals, the theory behind it, and the resulting implementation choices. Next, it will discuss how the data that is generated by the mechanism relates back to specific insights about the input parameters.

INTRODUCTION

The ultimate goal of the overall system is to provide insights into the influence of design parameters on a solution. That is, for example, how the parameters of a design solution are related, how sensitive the design goal is to parameter changes, or how close a certain solution is to some requirement constraint. This is done by deriving such properties from the data resulting from repeated simulation using automatically varied parameter values. In that sense, this mechanism is a Monte Carlo simulation as defined by Sawilowsky (2003; see also Box 3.3).

> Sawilowsky contrasts the Monte Carlo simulation and method, the latter using purely statistical methods without simulation—e.g. computing an area by randomly placing points and counting those inside.

Box 3.3: Monte Carlo simulation

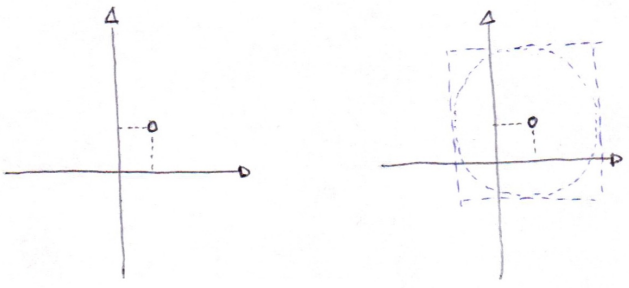


Figure 3.6: a) A single parameter value combination, and b) multiple parameter value combinations (non-stochastic)

Starting with a composed model, the user controls the simulation by defining a certain initial set of parameter values (see Figure 3.6-a) and the extent to which these may be varied (see Figure 3.6-b), in addition to some arbitrary performance metric. The system then repeatedly samples varying combinations of parameter values

and runs these as input values through a simulation. Finally, the measured performance metric values are analysed, the desired properties computed, and returned to the user for inspection. Should the user decide to adjust a parameter, the process is repeated with new samples.

Clearly, for the range of problems currently targeted by StructuralComponents, the properties that this system proposes to derive from simulation data are in truth deterministic (Chapman, 2001). That is, in general the relationship between a solution's performance metric and either of its significant parameters is somehow polynomial (see Box 3.4, 'Polynomial'), and could be derived exactly. However, it is exceedingly difficult to create a system that uses the smart approach (see Box 3.4, 'Smart') and still covers all occurring problem cases. The dumb approach naturally does—approximately, assuming sufficient sample size—using statistical methods.

Polynomial: the problems described here are polynomial in nature, though users may manually define other relationships—e.g. exponential.

Smart: here the term 'smart' is used for deriving properties deterministically and 'dumb' for doing the same using simulation. This follows the assertion that the smart approach would generally be computationally less expensive.

Box 3.4: 'Polynomial' and 'smart'

As with any general-purpose system, there is no guarantee that applying this system to an arbitrary specific design cases will yield sensible results. To an extent, statistical analysis can be used to ensure that some conclusion follows logically from a set of data. However, whether that conclusion is sensible depends on whether the data, and by extension the simulation model, is sensible to begin with. This system attempts to provide a measure of statistical soundness, and additionally—to a limited extent—qualifies results when mixing discrete and continuous input. Beyond that, it leaves any judgment of the physical soundness of a model up to the user.

STRUCTURED AND STOCHASTIC VARIATION

Perhaps the simplest mechanism of automatic variation would combine the values at constant intervals in one parameter range with values in another. For example, it would produce a hundred combinations of between 90 and 110 meters building height and between 1.3 and 1.7 core depth-to-width ratio, which parameters might be deemed essential by the engineer (assuming constant values for secondary parameters—building geometry, materials, load, etc.). Such a structured approach to variation should immediately reveal patterns concerning the relation between these parameters, such that they are likely visible on any bivariate plot.

Such structured variation serves to illustrate how the problem of automatic variation relates to the basics of combinatorics. The complete set of combinations consists of the leaves of a decision tree with a number of levels equal to the number of considered parameters. In the example, sets of values (e.g. 90, 100 and 110m height; 1.3, 1.5 and 1.7 ratio—although, the sample sizes would need to be higher in practice) are chosen and then permutatively combined:

- the tree root branches out into the possible height values to form the intermediary nodes of the first level (90m high, 100m high, 110m high);
- the intermediary nodes again branch out into the possible depth-to-width ratio values (90m high, 1.3 ratio; 90m high, 1.5 ratio; etc.).

The expected patterns resulting from structured variation can, on the other hand, also work restrictive. It is unlikely that the results evenly or entirely span their possible output range. This might be a problem from a standpoint of sensitivities and proximity to constraints, which require unbiased data. Another approach that would solve this issue is to use stochastic sampling, according to some probability distribution (see Figure 3.7). This can still be done in a structured way, combining the limited sets of stochastically sampled parameter values, or in a completely unstructured way, sampling new values for each combination.

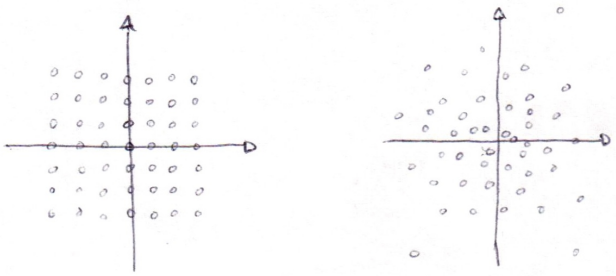


Figure 3.7: Structured and unstructured sampling

Another aspect to consider, regardless of whether sampling happens in a structured or stochastic manner, is the sampling distribution. The structured sampling example mentioned sampling from a range at constant intervals. The stochastic sampling equivalent would be sampling from a uniform distribution. An obvious alternative would be sampling stochastically or at predefined but variable intervals from a normal distribution.

3.1.3.2 Visualisation

A full assessment of available information visualisation theory falls outside the scope of this thesis. It should suffice to outline the principles of Tufte, who is an often-cited popular scholar on the subject, and provide some supporting and contrary positions by a few others. Importantly, the theory and principles discussed here will focus on the context of interactive software application graphics, and an audience of engineers with a presumed understanding of data and spatial relations.

GRAPHICAL EXCELLENCE

Tufte (1983) summarised the goal of visually displaying data, or quantitative information, in his concept of "graphical excellence, [which] gives the greatest number of ideas in the shortest time with the least ink in the smallest space". Visualisation serves to show data in such a way as to induce thought about its substance by way of making it coherent and encouraging comparison, while avoiding distortion. People are generally ill-equipped to understand large sets of numerical data when viewed as text. Visualisation enables someone to assess such data by engaging their visual pattern recognition.

Ideally, a graphic makes data coherent by showing it in its entirety and clustered—or not—around essential parameters. It should show different levels of detail to communicate both global trends and local subtleties. It enables or even encourages comparison by showing different data sets together, either in the same context or scaled and shifted to overlap. Graphics can show relationships between parameters and even provide a spatial language to think about and discuss expectations and check whether those match the data.

Graphics and visualisation can also easily distort data and suggest trends and correlations that do not exist. In general, small datasets (one or a few numbers) are best represented numerically, instead of graphically. When the choice is made to use a graphic, Tufted refers to its integrity and explains related concepts by listing examples of common mistakes:

- **proportionality**—exaggerating (or understating) a data measure in graphics (Tufte defines the *lie factor*—the shown-effect-to-effect ratio—and cites common exaggerations of two to five over the acceptable 1.05);
- **normalization**—for example not adjusting currency for inflation in time series, or showing building top deflection independent of height (data often depends on its context—two data points out of context cannot be compared);
- **dimensionality**—for example representing population size as circles or floor area as bars (numbers have both magnitude and order—difference in diameter is not the same as difference in area);

- **labelling**—using labels to hide facts, such as axes not starting at zero (labels may tell the whole truth, but visuals can still convey a different impression).

There is also the concept of data–ink ratio, or the ratio between graphical elements that carry data and those that don't ('ink' should be interpreted in the abstract sense when discussing computer graphics, where no actual ink is involved). Non-data ink can misrepresent data by distracting from or drowning it. Blasio and Bisantz (2002) find that high data–ink ratios improve reading accuracy, and to a lesser degree interpretation speed.

Purely illustrative elements—e.g. such as found in newspapers and other publications for non-technical audiences—will be left out of consideration[^deco]. Decoration on data elements, such as shape outlines, shadow or 3D effects, or skeuomorphic embellishments (such as the ubiquitous *speed gauge*), but also plot area outlines and dense grids are usually better omitted (even though it is found that 'useless' decoration does improve long-term recall of the information contained within a graphic; Bateman, 2010).

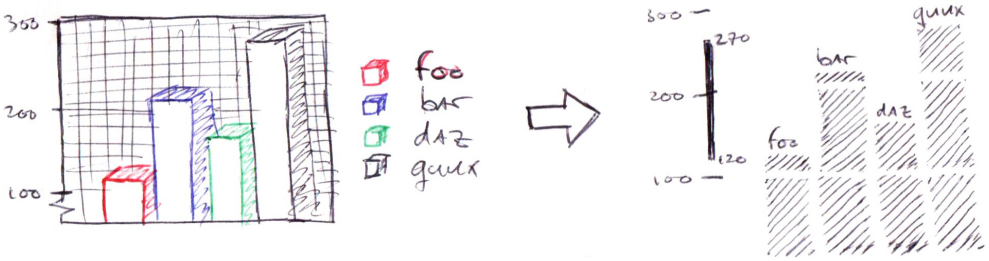


Figure 3.8: a) A bar chart with unnecessary decorative effects and colouring, a superfluous box and grid, external labelling, and a misleading axis offset; b) the same chart with an high data-ink ratio using multifunctional labelling (axis extents and positional labels) and bars (proportional, with tick lines).

Efforts to increase the ratio can also reinforce the data, such as when an axis also conveys a value range or distribution, when shapes are shaded instead of outlined, or when labelling is made multifunctional (see Figure 3.8). Within reason, the data–ink ratio should be as high as possible.

DASHBOARDS

More specific to this thesis and its primary function for data graphics is the theory behind software dashboard design. Building upon Tufte (which mainly target printed publications) Few (2006) sets out a dozen or so principles as 'pitfalls', to be avoided. Those dealing with aesthetics and accuracy are ignored here, while useless decoration is discussed above. Other pitfalls can be explained succinctly: exceeding the boundaries of a computer screen; displaying excessive detail or precision; or ineffectively highlighting what is important. Important pitfalls that merit further explanation are:

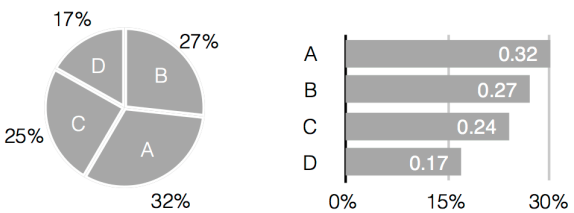


Figure 3.9: a) The pie-chart can convey relationship meaning through position and adjacency, but makes similar values look equal; b) the histogram shows differences clearly. In both cases, the exact values could be more easily read from a table.

- A common mistake is choosing inappropriate graphic types to convey data (e.g. like line graphs,

histograms, scatter plots, pie charts, and other *'widgets'*). Sometimes graphics variety is introduced just for the sake of it (conveying unintended meaning), other times for the sake of immediate appeal ('flashiness' at the cost of clarity). Each type conveys a different idea, which should match the intent: a pie chart might suggest *parts of a whole*, but inaccurately; a histogram will highlight even small differences, but doesn't show relationships (see Figure 3.9).

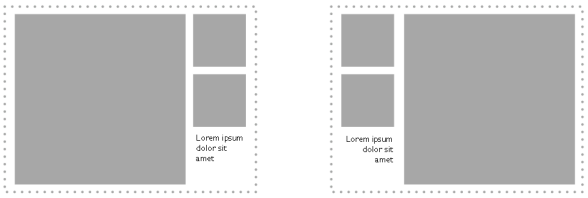


Figure 3.10: Large elements positioned towards the top-left demand most of the attention; positioning smaller elements to the left of, or 'before' large elements spreads attention more evenly.

- Another important consideration is the arrangement of graphics on the dashboard. Bateman (2010) shows that much higher attention is paid to the top and left areas than the bottom and right. Obviously higher attention is paid to larger and more prominent graphics. These phenomena can be exploited to bring the most important data to the front (important information large and in the top-left). Or, it can be used to spread attention out evenly over the available surface (small graphics in the top-left, larger ones off to the bottom-right; see Figure 3.10).

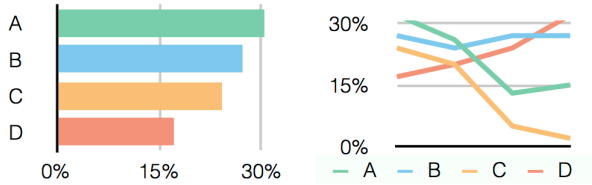


Figure 3.11: Green, orange and red colours category colours convey qualities such as 'good' or 'bad', which may be unintended; category colours in labelled histograms are redundant, but do allow the eye to easily follow line graphs.

- Colour must be used carefully. Using false-colour coding is prevalent in engineering (e.g. to show high/low or compressive/tensile stress), but has led to a strong subconscious *'green is good, red is bad'*-association. Introducing similar colours to differentiate ordinals or categories can convey unintended meaning (and may be redundant if properly labeled; see Figure 3.11). Colour blindness may call for brightness differences in addition to hue. Light backgrounds are less stressful, while dark backgrounds allow for higher contrast and better highlighting of important data.
- Graphics should convey context in addition to the data itself. It must be clear whether a lone number is good or bad, safe or unsafe. Likewise, it is easy to convey an unintended maximum or minimum by choosing the wrong graphic (e.g. the speed gauge, which implies a 'dangerous maximum' that may not exist). Labelling, axes, and grids can help here, but preferably in a way that preserves the data–ink ratio.

3.2 Implementation

This thesis puts forward an interactive tool based on an abstraction mechanism to allow the designing engineer to explore a problem or design space. In terms of the design cycle, it proposes a typical iteration as an initial synthesis phase and a subsequent analysis phase. The former features model composition and simultaneous deduction of essential parameters followed by the formulation of a metric based on those. The latter involves

performance computation and simultaneous steering towards meaningful areas in the design space. Each iteration revolves around posing a measurable question to which the data gathered throughout the problem space must provide an answer.

IMPLEMENTING THE CHARACTERISTIC DESIGN CYCLE

The synthesis phase consists of composition and deduction. Composition refers to the composition of computable logic, forming a parametric and associative model as described profusely in the preceding theses, and further expanded on here. Deduction refers to the deduction of which parameters are essential to the composition and a measurable metric as a function of those parameters, based on which a value can be ascribed to the composition.

The analysis phase consists of data gathering, visualisation and steering. Data gathering refers to the computation of the metric for various stochastically chosen value combinations for the set of essential parameters, the results of which are then presented in some meaningful way through visualisation. Steering refers to the navigation through the problem space, specifying points of interest for finer analysis and to form a basis for further investigation in successive iterations.

SYNTHESIS, ANALYSIS, AND INFERENCE SUBSYSTEMS

As implemented, the system relies on parametric and associative modelling of the design composition through the familiar box-and-wires metaphor. This will be referred to as the *synthesis* subsystem. As its implementation is familiar, Section 3.2.1 only discusses it briefly; software implementation choices are found in Chapter 5. The synthesis system features structural components which form the input for a Finite Elements-based analysis engine, which will be referred to as the *'analysis'* subsystem. Its implementation is very detailed, so it will be discussed only briefly in Section 3.2.2; its full implementation is discussed in Chapter 4.

Built on top of the modelling system is a system of user-driven, semi-automatic stochastic abstraction. This will be referred to as the *'inference'* subsystem, and takes cues from the user's parameter choices and stochastically varies them to generate a multitude of design solutions that are analysed simultaneously. The resulting data is then analysed and presented to the user through various visualisations. This system goes beyond the isolated performance results of the single solution to yield the abstracted broad view of the design space. The range of stochastic variation is left as a choice for the user, and visualised alongside the resulting range of the metric to yield a system of design by bandwidth.

COMPUTATION IN THE EARLY STAGE DESIGN PROCESS

At the early stage, readily available analytical power heavily outweighs the complexity of the typical problems of the first exploratory forays into a problem space. Theses problems are commonly tacked using a combination of simple schematisations, tules of thumb and intuition in the form of generalised experience. The analysis component of this process is relatively small and lends itself well to parallelisation. Although analysis is not in itself the object of performance improvement, the implementation proposes to exploit its features in order to automate calculations in a system of semi-stochastic abstraction, thereby reducing the interruptions of calculation and allowing greater attention for composition and exploration.

3.2.1 SYNTHESIS: PARAMETRIC AND ASSOCIATIVE MODELLING

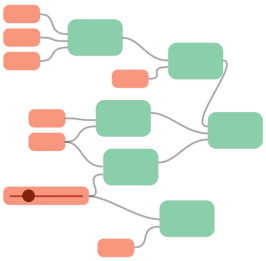


Figure 3.12: The synthesis subsystem makes use of the boxes-and-wires paradigm of parametric and associative design modelling

The synthesis subsystem makes use of parametric and associative design ("PAD") modelling. The system features structural components with basic building blocks like nodes and elements, and smarter compositions like trusses and building sections. The components are connected together, and to input parameters, forming a generative, directed data flow graph. The graph deterministically maps its input to output. Each generated result with unique parameter values constitutes a unique solution.

As part of the abstraction system—the implementation of which is discussed in Section 3.2.2—the modelling system features two special designations: parameters can be designated as *essential*, and components can be designated as a *metric* or a *constraint*. The essential parameters form the set of parameters for which the values are automatically varied, based on a given sampling distribution. Both metrics and constraints are the components for which the output values are collected, metrics determining a solution's scores and constraints its validity.

The system uses a custom-developed PAD engine which makes use of the boxes-and-wires paradigm. The engine has a limited feature set (one-directional, non-looping), which resembles familiar software that uses the same paradigm. Accordingly, its functional details are not discussed here. Relevant details about its software implementation are discussed in Chapter 5.

3.2.2 ANALYSIS: DESIGN EXPLORATION

The analysis subsystem makes use of a Finite Element Method analysis engine that was custom-developed for this thesis. Its theoretical background and functional implementation choices are discussed at length in Chapter 4. Its software implementation choices are discussed in Chapter 5.

3.2.3 INFERENCE: DESIGN EXPLORATION

The analysis phase results in a large number of solutions, each associated with value sets for both the input parameters and the output metrics.

3.2.3.1 Solution space

Coming back to the solution space analogy, each solution's set of input values represents a coordinate, and its set of output values its performance. All results represent a region of solutions whose performance—in the form of its output values—is now known. However, presuming arbitrary problems, the location and shape of the solutions regions will also be arbitrary. A distance between solutions in one dimension may be small or large depending on what the parameter represents. As such, comparing values is difficult.

NORMALISATION ACCORDING TO DISTRIBUTION

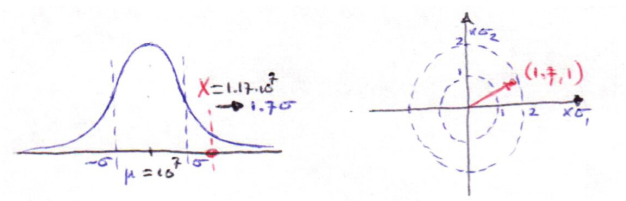


Figure 3.13: a) Normalisation according to value distribution, and b) distribution-normalised solution space

One solution to this problem is normalisation, i.e. establishing a common base within which values can be compared. A simple method is to normalise using the distribution of a value over all results (see Figure 3.13-a). Each input value is already associated with a given distribution, based on a mean and a variance, from which it was sampled. Output values may be distributed arbitrarily, but their same mean μ and variance σ^2 can be computed from the total number N of solution output values $X_1, X_2, \ldots X_n$ using the simple formulae:

$$\mu = \frac{\Sigma X_i}{N},\ \sigma^2 = \frac{\Sigma (X_i - \mu)^2}{N} = \frac{\Sigma X_i^2}{N} - \mu^2.$$

Using the mean and variance of a value, its actual normalisation can be done using the simple formula:

$$\mathrm{norm}_{\mu,\sigma}(X_i) = \frac{X_i - \mu}{\sigma}.$$

Normalisation results in the distribution-normalised solution space where the solutions region is centered at the origin and shaped more or less spherical (or rectangular, depending on sampling method; see Figure 3.13-b). In the context of a distribution, a solution will now become average or an outlier. Comparison between two arbitrary values becomes a comparison between average values, or an average value and an outlier, or two outliers, etc.

3.2.3.2 Data analysis

Having established a common base of comparison, the normalised data can already be represented in overview, such as in simple visualisations like parallel coordinates or scatter plots (more on those in Section 3.2.3.3). The solutions can be enumerated over, and for every solution, its associated input parameters and output metrics are known. However, the reverse—finding which solutions are (approximately) associated with some set of input parameters or output metrics—remains difficult.

Finding a solution approximately associated with a certain value set constitutes a *nearest-neighbour* search. In the naive implementation, this is an $O(N)$ operation: it takes in the order of N comparisons to find the correct solution (in reality, exactly N comparisons, as each solution must be checked). Derivative operations, such as sorting, querying, or continuous interpolation, are similarly expensive.

PARTITIONING AND TESSELATION

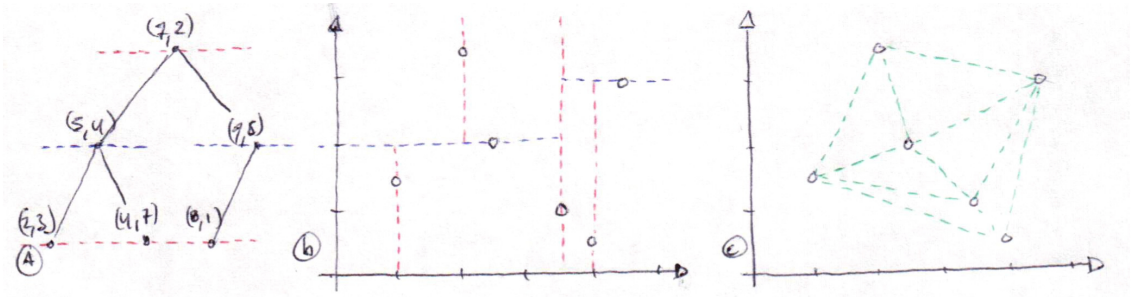


Figure 3.14: a) An example *k-D tree* partitioning, b) represented in space, and c) *Delaunay* tessellation represented in space of the points $(2,3),\ (5,4),\ (9,6),\ (4,7),\ (8,1),\ (7,2)$.

For sorting and querying operations, one solution to reduce computational cost is to precompute a spatial partitioning. Many such partitioning schemes are available, but for this thesis the *k-D tree* (for k-dimensional tree; Bentley, 1975) scheme was chosen for its ease of implementation and fit (it works best when the number of points N is much larger than the number of dimensions k, $N \gg k$). This is a binary tree partitioning scheme which associates each node with a data point and splits further nodes along the hyperplanes (see Box 3.5) in successive dimensions (see Figure 3.14-a,b). I.e., it will iteratively choose a splitting point P_0 (ideally the median) and partition each further data point P (all with associated values $X_{P,1}, X_{P,2}, \ldots X_{P,k}$) into one of two sets $\mathbf{P}_L$ and $\mathbf{P}_R$ on both sides of the hyperplane through value $X_{P_0,i}$ where $i = d \bmod k$ and d is the depth of the associated node (which increases from 1 at the root node):

$$P \in \begin{cases} \mathbf{P}_L & \text{if } X_{P,i} < X_{P_0,i} \\ \mathbf{P}_R & \text{otherwise} \end{cases} \text{ with } i = d \bmod k$$

For interpolation, a solution to reduce computational cost is to compute a continuous tessellation of the data points, on which each the output metric values for each arbitrary point can be interpolated from its surrounding vertices. Once more, many tessellation schemes are available, but for this thesis the *Delaunay tessellation* was chosen for its ease of implementation. This tessellation creates the unique (assuming *general position*, see Box 3.5; De Berg, 2008) non-overlapping and gapless piecewise continuum of *k-simplices* (see Box 3.5) with vertices at the data points, that satisfy the *Delaunay criterium* that no point is inside the circum-hypersphere (see Box 3.5) of any of the simplices (see Figure 3.14-c). I.e., it will create a tessellation $\mathrm{DT}(\mathbf{P})$ of k-simplices with vertices at data points $P_a, P_b, \ldots$ constituting the set $\mathbf{P}$ such that no point P is inside any of the simplices of $\mathrm{DT}(\mathbf{P})$.

Hyperplane: the concept of a hyperplane is the generalisation of a regular 1-D line in 2-D space, or a 2-D surface in 3-D space, to a $(k-1)$-D surface in k-D space.

General position implies that e.g. in 3-D space no line (or 1-D flat) contains 3 points (no 3 points are collinear), or in k-D space no $(k-2)$ -D flat contains k points. With stochastic sampling of input parameter values, this will be the case with sufficiently high probability.

Simplex: the concept of a simplex is the generalistion of a triangle in 2-D space, or a tetrahedon in 3-D space, to a geometric shape with $k+1$ vertices in k-D space.

Circum-hypersphere: the concept of a circum-hypersphere is the generalisation of a circumcircle in 2-D space, or a circumsphere in 3-D space, to higher dimensional k-D space. The circum-hypersphere uniquely circumscribes (touches the vertices of) a k-simplex.

Box 3.5: 'Hyperplane', 'general position', 'simplex', and 'circum-hypersphere'

COMPUTING METRIC SENSITIVITY

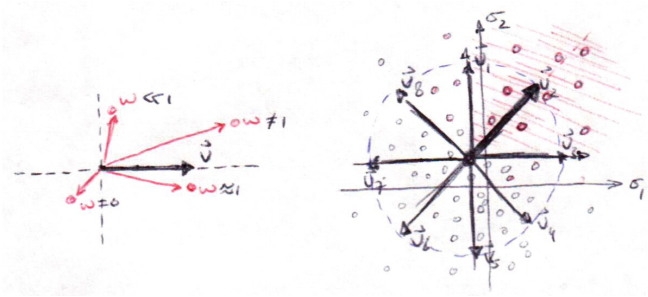


Figure 3.15: A visual representation of the sensitivity of a metric to a in two parameters

The sensitivity of a metric function is its rate of change in some arbitrary parameter direction. I.e., the change ∂_X of metric value X along some arbitrary vector $\mathbf{v} = (\partial_{X_1}, \partial_{X_2}, \ldots)$ with ∂_{X_i} = a change in parameter value X_i, relative to that of a center point P_0. The change can be derived by comparison with the point at the other end of vector $\mathbf{v}$. As this point is unlikely to exist, and surrounding points unlikely to lie exactly on the vector $\mathbf{v}$, the change must be interpolated using some weighting function w:

$$\partial_X(P_0, \mathbf{v}, w(\ldots)) = \frac{\Sigma_i \partial_{X,P_i} w(\mathbf{v}_{P_i})}{\Sigma_i w(\mathbf{v}_{P_i})} = \text{ for } P_i \text{ in } \mathbf{P},$$

where $\partial_{X,P_i} = X_{P_i,r} - X_r$. The weighting function might reduce the weight for values for points that lie more or less orthogonal to the vector, or completely discount those for points that in the opposite direction. It should be chosen based on the data density surrounding the center point, and applied consistently whenever comparing sensitivity for multiple directions.

COMPUTING PARAMETER RELATIONSHIPS

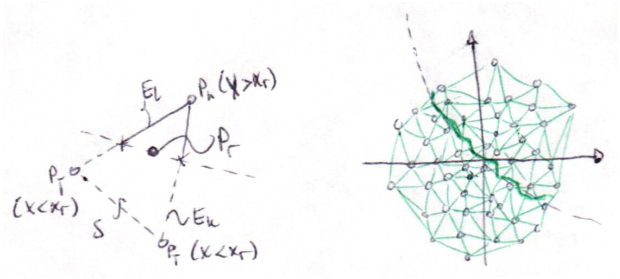


Figure 3.16: A visual representation of the relationship between two parameters with regard to a third metric

The relationship between parameters with regard to a metric is represented by the shape of the region in solution space where the metric values are constant (see Figure 3.16). I.e., the region around the set of points $\mathbf{P}_r$, of which each point P_r has variable input parameter values $X_{P_r,a}, X_{P_r,b}$ but a constant metric value $X_{P_r,r} = X_r = \text{constant}$ (and $X_{P_r,h} = X_h = \text{constant}, \ldots$ if there are more parameters that are not being considered in the relationship). As with sensitivity, it is unlikely that there will be points with a metric value at exactly X_r, and so the points P_r in $\mathbf{P}_r$ must be interpolated. This interpolation uses the tessellation, computing the unweighted average point P_r between the set $\mathbf{P}_{r,\mathbf{E}_{r,\mathbf{S}}}$ of intersection points of the k-simplex edges that cross the X_r plane $\mathbf{E}_{r,\mathbf{S}}$ for each of the set of k-simplices $\mathbf{S}$:

$$P_0 + \Sigma_j \left(\frac{\Sigma_i \mathbf{v}_{P_i} / \partial_{X_r,P_i}}{\Sigma_i 1/\partial_{X_r,P_i}} \text{ for } P_i \text{ in } \mathbf{P}_{E_j} \text{ of } E_j \right) / (\Sigma_j) \text{ for } E_j \text{ in } \mathbf{E}_S \text{ of } k\text{-simplex } S,$$

where $\partial_{X_r,P_i} = X_{P_i,r} - X_r$ and P_0 is some reference point (likely with metric value X_r). This set of points includes points whose proximity to X_r may be due to the influence of other parameters, which may duly be discounted using a weighting function.

COMPUTING CONSTRAINT PROXIMITY

The constraint proximity of a solution is that solution's proximity to solutions that are invalid with regard to a constraint. Is is computed similarly to sensitivity, although now it considers the weighted-average distance to the set of invalid points $\mathbf{P}_C$:

$$d_X(P_0, \mathbf{v}, w(\ldots)) = \frac{\Sigma_i d_{P_i} w(\mathbf{v}_{P_i})}{\Sigma_i w(\mathbf{v}_{P_i})} = \text{ for } P_i \text{ in } \mathbf{P}_C,$$

where $d_{P_i} = \|\mathbf{v}_{P_i}\|_2$.

3.2.3.3 Data visualisation and exploration

Once the data is computed and analysed, it is ready to be explored. A number of visualisation types are proposed which allow fairly deep exploration against acceptable ease of implementation.

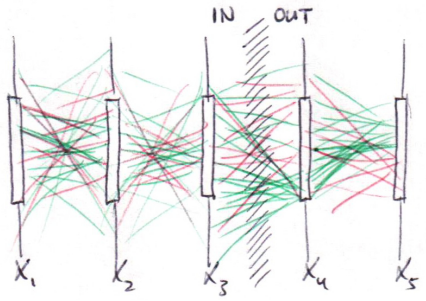


Figure 3.17: An example of a parallel coordinates plot

The first proposed visualisation type is the **parallel coordinates plot**. This visualisation displays k parallel axes for each of the k.$ dimensions of a data point. Then, it plots a polyline for each solution along its coordinate points on the axes, creating a series of diagonal lines spanning between the axes. This visualisation is well-suited to visualise high-dimensional data, as it isn't inherently constrained by the dimensionality of the display medium (Inselberg, 1985). The plot can display both input parameter and output metric values together, but should somehow be differentiated to avoid confusion.

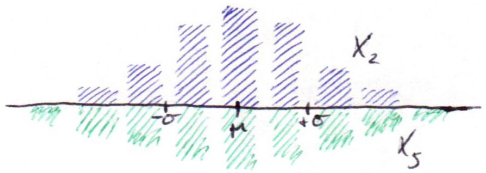


Figure 3.18: An example of a frequency distribution plot

The second proposed visualisation is the **frequency distribution plot**. This is a histogram plot of which the bar length is determined by the frequency with which solutions fall within the bar's value domain. Assuming sufficiently large data sets, the plot will reflect the distribution given for the input parameter values, and clearly show how metric output values are distributed (perhaps uniform or normal, as the input, but possibly more insightful, e.g. multiple peaks). The frequency distribution plot lends itself well to being overlaid on the axis of the parallel coordinates or scatter (see below) plots.

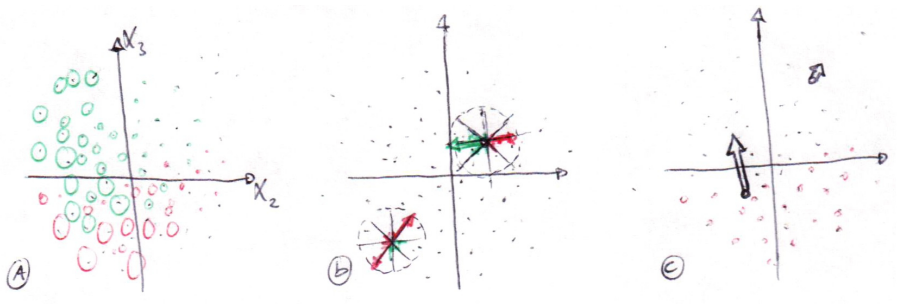


Figure 3.19: An example of a) an unadorned scatter plot visualisation, b) an overlaid metric sensitivity wind rose, and c) an overlaid constraint proximity (for an overlaid parameter relationship, see Figure 3.16)

The third proposed visualisation type is the **scatter plot**. This visualisation uses pairs of values as coordinates to plot points on a plane (normalised according to Section 3.2.3.1; see Figure 3.19-a). Further properties can subsequently be represented by the symbol, colour, or size of the points, which will show trends at a glance (gradients will be clearly be visible as e.g. shifting colours). The scatter plot can additionally be adorned by visualisations of the computed properties described in Section 3.2.3.2.

- **Metric sensitivity**: The difference in metric value is computed in several directions around a known solution. This results in an overview of metric sensitivity as expressed in rate of change per direction, visualised as radial positive and negative vector fields (colour represents positive of negative rate of change, magnitude represents rate magnitude; see Figure 3.19-b).
- **Parameter relationship**: Metric values are computed for several combinations of parameters, and an approximate continuous field is extrapolated and intersected with a target value. This results in a parameter relationship as expressed in the set of parameter combinations that result in the same target metric value (see Figure 3.19).
- **Constraint proximity**: The distance to surrounding invalid solutions is computed in several directions centered around a known solution. The results are then averaged and inverted, resulting in single evasive direction away from constraints (see Figure 3.19-c).

INTERACTIVE EXPLORATION

Beyond the visualisation types, which allow exploration through static imagery, it is proposed that an interactivity layer be implemented. Such interactivity could provide control over which plots are shown and which value pairs are being compared. It could provide control over axes domain, inversion, and location, or over data filters to show only subsets of all solutions. Interactivity could also allow control over the abstraction settings themselves, in order to make input parameter distribution adjustments and request computation of new or additional data points.

As an example, what follows is a brief overview representing the proposed interactivity (see Figure 3.20):

- The parallel coordinates ("PC") plot would serve as the user's starting point after the successful computation of an abstraction. It shows all input parameters and output metrics (differentiated), normalised according to distribution. The user might rearrange or adjust axes to compare different value pairs, and perhaps filter out invalid solutions.
- The user might choose to further inspect a pair. The PC plot will move off to one side, where every axis other than the inspected pair of axes would be hidden. On either side of the single-field parallel coordinates plot, frequency distribution histograms are drawn giving immediate insight into where values cluster together.

- Adjacent to the parallel coordinates plot, taking up much of the display, a scatter plot would be drawn. Unadorned at first, with only the natural gradients in the data showing. The user might choose to inspect sensitivity, and a wind rose is drawn around a reference point set at the origin. The point might be moved around, redrawing the rose at each location.
- After further inspection, the user might choose to refine some parameter, and shift another. After a moment, a new data set is generated and exploration begins again.

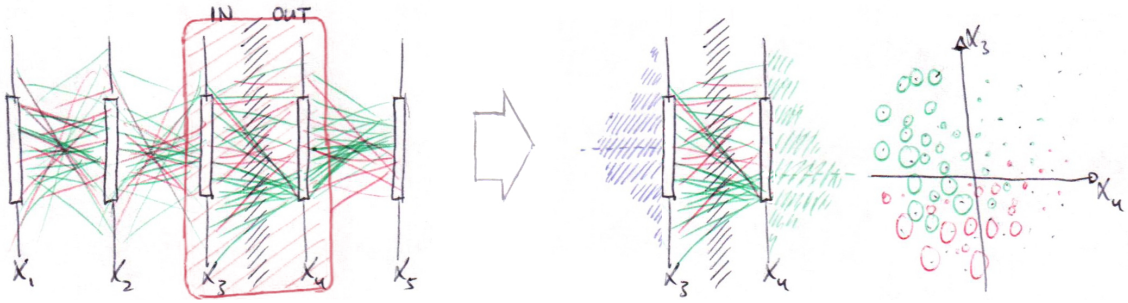


Figure 3.20: An overview of the proposed interactive exploration: the parallel coordinates plot forms the starting point, from which a values pair can be further inspected

This page is intentionally left blank.

4 Finite element analysis

4.1 Overview

The Finite Element method is a specific approach to numerically solving a continuous system that is governed by a partial differential equation. This equation, commonly referred to as the governing equation, is the product of a range of equations that model a real-world material. Ultimately, these equations describe the relationship between the material properties and its behavior as a response to external action.

The finite element method falls under the general class of numerical solution methods referred to as the Galerkin method, which revolves around a relaxation of the formal equation governing the problem. The formal, or *strong* form of the equation must hold absolutely, at every point within the problem domain. The Galerkin method only requires it to hold on the domain boundaries and cumulatively, summed up over the entire domain. It puts forward an approximate solution and assumes that if that solution holds cumulatively it is a good model of the exact solution.

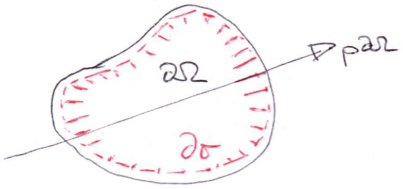
The specific approach taken in the Finite Element method is the systematic discretization and search for polynomial trial functions, which allows it to be implemented as a software program. Specifically for structural engineering, analysis using the Finite Element method typically follows the following steps:

1. Discretization into elements that are governed by the strong form of known ordinary differential equations modeled after physical characteristics.
2. For every type of element, derivation of the weak formulation of the governing equation over the element's subdomain.
3. For every element, numerical integration over the element domain resulting in a linear system of equations.
4. Assembling the global system by superposing the various element systems, typically transformed from local to global space, according to shared degrees of freedom.
5. Imposing Dirichlet boundary conditions (constraints) that prescribe solution function values and effectively eliminate degrees of freedom from the system, and Neumann constraints that prescribe forces and effectively preclude the trivial solution. Depending on the implementation, those Neumann conditions that themselves depend on the solution, such as spring constraints, are rewritten into the system of equations.
6. Solving the system of equations, typically using common numerical matrix solution methods. In the majority of cases, depending on the handling of constraints in the previous step, the final matrix can be symmetrical.
7. Post-processing the solution to compute relevant engineering values, such as continuous displacement and element force approximations between nodes, typically transformed back to local element space.

This chapter will cover these steps and the mathematical theory behind them, as they have been implemented within the structural analysis framework that is part of thesis.

4.2 Theory

At the core of the finite element method is the model of the material that is being analyzed. This model takes the form of a partial differential equation that governs the material behavior. This governing equation can be seen as the combination of three equations, commonly referred to as the kinematic, constitutive, and equilibrium relationships. Respectively, these equations define the relationships between space and displacement, displacement and force, and the equilibrium between forces.



At a given coordinate $\mathbf{x}$, the change of internal stresses $\nabla\boldsymbol{\sigma}$ is in equilibrium with the external forces $\mathbf{p}$ that are applied there. This internal stress is constitutively related through some ratio $\mathbf{D}$ to the local strains $\boldsymbol{\varepsilon}$, which in itself is a kinetic measure of the change in local displacement $\nabla\mathbf{u}$. In formula, this yields a typical governing equation

$$\nabla\boldsymbol{\sigma}(\mathbf{x}) + \mathbf{p}(\mathbf{x}) = \nabla(\mathbf{D}\boldsymbol{\varepsilon}(\mathbf{x})) + \mathbf{p}(\mathbf{x}) = \nabla(\mathbf{D}\nabla\mathbf{u}(\mathbf{x})) + \mathbf{p}(\mathbf{x}) = \mathbf{0}.$$

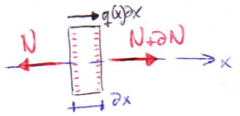
Throughout this section, the theoretically and conceptual explanation of the finite element method will be interwoven with the example of its application to the model of a simple bar under axial load. At the end of the section a summary is given for a similar derivation for an Euler-Bernoulli beam.

The kinematic, constitutive, and equilibrium equations also govern the simple bar element. As the element only extends in one direction, its kinematic relationship follows from the expression of strain along that direction. The relation should hold for each infinitesimal slice of the bar, yielding

$$\varepsilon(x) = \lim_{\partial x \to 0} \frac{\partial u(x)}{\partial x} = \frac{\mathrm{d}u(x)}{\mathrm{d}x}.$$

As the simple bar has a constant cross section area over its length, this can be incorporated in its constitutive relationship,

$$\sigma(x)A = N(x) = EA\varepsilon(x).$$



Lastly, from the fact that each infinitesimal slice of the bar should be in equilibrium, it follows that

$$-N + q(x)\partial x + N + \partial N(x) = 0 \;\Rightarrow\; \frac{\partial N(x)}{\partial x} + q(x) = 0$$
$$\lim_{\partial x \to 0} \frac{\partial N(x)}{\partial x} + q(x) = \frac{\mathrm{d}N(x)}{\mathrm{d}x} = 0.$$

Combining the three equations yields the strong form of the differential equation governing the simple bar,

$$EA\,\frac{\mathrm{d}^2u(x)}{\mathrm{d}x^2} = EAu''(x) + q(x) = 0.$$

Box 4.1: The derivation of the weak form from the strong form

The strong form of an element's governing equation is a differential equation that can be solved numerically using the Galerkin method of weighted residuals. This method discretizes the domain of the problem into one or more disjoint subdomains separated by boundaries and subsequently employs test and weighting functions to approximate the exact solution over the discretized domains.

The method is not applied directly to the strong form of the governing equation, which is typically a partial differential equation. Instead, the equation is converted to its weak form by first multiplying it by an arbitrary weighting function and then integrating over its domain. This yields the weak form of the governing equation, typically of the form

$$\int_{\Omega} [\nabla(\mathbf{D}\nabla\mathbf{u}(\mathbf{x})) + \mathbf{p}(\mathbf{x})]\delta\mathbf{u}(x)\mathrm{d}\Omega = \mathbf{0}.$$

The weighting function can be understood as a virtual displacement. The integration employs the fact that if the strong form holds at every point in its domain, it must also hold cumulatively, over its entire domain. In physical terms, the weak form of the governing equation is equivalent to energy. Specifically, it is a measure difference between internal and external work, or the energy contained within the domain, which must be minimal to approximate equilibrium.

The Galerkin method is based on the assumption that if a test solution holds cumulatively, it is a good approximation of the exact solution. The extent to which this assumption holds true depends on the soundness of the discretization and test and weighting functions.

The method ultimately employs equivalent parameterized coefficient series $\Sigma_i c_i \phi_i(x)$ as substitution functions for the test and weighting functions that represent the solution and virtual displacement. These functions effectively convert the problem to an ordinary differential equation problem that is numerically stable and approximately equivalent to the partial problem.

$$\int_{\Omega} [\nabla(\mathbf{D}\nabla(\Sigma_j\mathbf{u_j}(\mathbf{x}))) + \mathbf{p}(\mathbf{x})] \times \Sigma_i c_i \phi_i(x)\mathrm{d}\Omega = \mathbf{0}.$$

The strong form of the governing equation of the bar element is first multiplied by a virtual displacement, the product of which is subsequently integrated over the domain $\Omega = (x_1, x_2)$ of the bar, yielding

$$\int_{\Omega} [EAu''(x) + q(x)]\delta u(x)\mathrm{d}\Omega = 0.$$

This integration is then separated by parts, reducing the order of the derivative by explicitly defining the boundary $\Gamma = \{x_1, x_2\}$ of the bar (where n is the vector normal to the boundary—or the signed scalar, in the case of the one-dimensional simple bar). This then yields the weak for of the governing equation,

$$[nEAu'(x)\delta u(x)]_{\Gamma} + \int_{\Omega} [-EAu'(x)\delta u'(x) + q(x)\delta u(x)]\mathrm{d}\Omega = 0.$$

The Galerkin method can now be used solve the problem numerically. To this end, the test and weighting functions dependent on the coordinate x are substituted by approximations, each consisting of a coefficient series of shape functions,

$$u(x) = u_1\phi_1(x) + u_2\phi_2(x) \ + \ \cdots \ + \ u_r\phi_r(x) = \Sigma_j u_j \phi_j(x),$$
$$\delta u(x) = w_1\phi_1(x) + w_2\phi_2(x) \ + \ \cdots \ + \ w_r\phi_r(x) = \Sigma_j w_j \phi_j(x).$$

NB. Part of the boundary term, $nEAu'(x)$, is similarly substituted by a coefficient series, but shall from this point simply be represented by F_i.

Substituting these series approximations into the weak form of the governing equation yields

$$\Sigma_i w_i F_i \phi_i(x)|_{x=\Gamma} + \int_{\Omega} [-EA\Sigma_j u_j \phi_j'(x)\Sigma_i w_i \phi_i'(x) + q(x)\Sigma_i w_i \phi_i(x)]\Omega = 0.$$

Since $\Sigma_i w_i$ occurs in every term, and w_i is arbitrary, the above equation can be separated into n independent equations, each with a single unknown u_i. For each i in $1, 2, \ldots, n$, omitting $\Sigma_i w_i$ and rearranging to isolate the term containing the unknown yields

$$\Sigma_j [\int_\Omega EA\phi_i'(x)\phi_j'(x)\Omega] u_j = \int_\Omega q(x)\phi_i(x)\mathrm{d}\Omega + F_i\phi_i(x)|_{x=\Gamma},$$

or, in shorter form, $K_{ij}u_j = f_i$.

Box 4.2: The application of boundary conditions to the weak form resulting in the solvable differential equation

From the Galerkin approximation of the weak form of the governing equation—and in particular from its shorter form—it should become immediately apparent that the method results in a system of equations that can be represented as a linear equation involving what is commonly called the stiffness matrix, as evident in the progression

$$\Sigma_j [\int_\Omega \nabla\phi_i(\mathbf{x}) D_{ij} \nabla\phi_j(\mathbf{x})\mathrm{d}\Omega] u_j \rightarrow K_{ij}u_j \rightarrow \mathbf{Ku}$$

It should be noted that in the above systems that the independent variable $\mathbf{x}$ stands for a set of independent variables $x_1, x_2, \ldots$, that the test function parameters u_i could be any arbitrary set of degrees of freedom $u_1, u_2, \ldots$, and that the stiffness term D_{ij} varies for every type of element. As such, the Galerkin method represents a general solution method using discretized elements that lends itself to a wide range of partial differential equation problems. Following a series of steps similar to those used to derive the simple bar system of equations in the example, equivalent systems can be derived for various other elements.

The finite element method is a specific application of the Galerkin method that uses systematic discretization of the problem domain and selection of shape functions. These shape functions form the basis for the Galerkin coefficient series test and weighting functions used in the approximate solutions. They are parameterized to depend on a finite set of degrees of freedom, and somehow correspond to one of them on select boundaries—through direct equality or equality of a derivative.

The simple bar is a one-dimensional problem with two nodes. The weak form of its governing equation features the first-order derivative of the shape functions on the left-hand side, and no derivatives on the right-hand side. Consequently, the problem calls for two shape functions of *C1*-continuity, of the systematic coefficient polynomial form

$$\phi_i(x) = c_{i1} + c_{i2}x.$$

The values of the coefficients follow from the Neumann boundary conditions, which hold that the shape function corresponding to a particular node must have be 1 at the corresponding coordinate, and 0 at the other. Thus,

$$\begin{pmatrix} c_{11} \\ c_{12} \end{pmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & l \end{bmatrix}^{-1} \begin{pmatrix} \phi_1(0) \\ \phi_1(l) \end{pmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & l \end{bmatrix}^{-1} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{l} \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

and

$$\begin{pmatrix} c_{21} \\ c_{22} \end{pmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & l \end{bmatrix}^{-1} \begin{pmatrix} \phi_2(0) \\ \phi_2(l) \end{pmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & l \end{bmatrix}^{-1} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \frac{1}{l} \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Substituting the coefficients in the shape functions results in

$$\begin{pmatrix} \phi_1 \\ \phi_2 \end{pmatrix}(x) = \frac{1}{l} \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} 1 \\ x \end{pmatrix}, \text{ and } \begin{pmatrix} \phi_1 \\ \phi_2 \end{pmatrix}'(x) = \frac{1}{l} \begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

These functions can now by substituted into the left-hand side of the weak form to derive the stiffness matrix,

$$EA\begin{bmatrix} \int_x \phi_1'(x)^2 \mathrm{d}x & \int_x \phi_1'(x)\phi_2'(x)\mathrm{d}x \\ \int_x \phi_2'(x)\phi_1'(x)\mathrm{d}x & \int_x \phi_2'(x)^2 \mathrm{d}x \end{bmatrix} = \frac{EA}{l}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

Box 4.3: The shape function integration resulting in the axis-aligned bar element stiffness matrix

The individual terms of the stiffness matrix are ultimately computed using some form of numerical integration. In the case of simple elements, this is often done prior to the analysis, using individual parameterized element matrices that depend on the element geometry and properties.

Often, complex element geometry is further parameterized to a natural coordinate system with domains ranging from -1 to 1 for all dimensions. These natural, or isoparametric coordinates are related to real coordinates through the Jacobian matrix $J(\mathbf{x}) = \mathrm{d}f_i(\mathbf{x})/\mathrm{d}x_k$ and its determinant j. Integrating over the isoparametric geometry is generally a lot faster than integrating over the complex geometry itself.

Individual element stiffness matrices are usually computed within a coordinate system local to the element itself. The element matrices are subsequently transformed and permuted using relatively simple symmetric matrix multiplication operations, before they are summed up to form the global stiffness matrix. This assembly process can ultimately be summed up as

$$\mathbf{K} = \Sigma_i \mathbf{Z}_i^{\mathrm{T}} \mathbf{K}_i \mathbf{Z}_i.$$

The fully assembled global stiffness matrix is generally not invertible—at least not in the static case. The system must first be constrained in space. This requires the application of any number of Dirichlet boundary conditions to eliminate fixed degrees of freedom. Once the system is sufficiently constrained, it can be solved using some type of numerical matrix solution method.

Analogous to the simple bar element, the Euler-Bernoulli beam element again follows the establishment of the governing equation, the conversion to its weak form, the exposition of right-hand side boundary conditions, and the final computation of the local element stiffness matrix. The kinematic relation, assuming that the cross-section always remains perpendicular to the beam axis, and that $v(x) \ll 1$ so that $\sin v(x) \approx v$,

$$\lim_{\partial x \to 0} \frac{1}{\partial x}\begin{pmatrix} \partial v(x) \\ \partial u(x,y) \end{pmatrix} = \frac{1}{\mathrm{d}x}\begin{pmatrix} \mathrm{d}v(x) \\ \mathrm{d}u(x,y) \end{pmatrix} = \begin{pmatrix} \theta(x) \\ \varepsilon(x,y) \end{pmatrix},$$

and

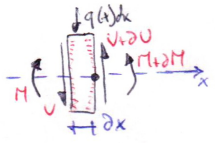
$$u(x,y) = y\theta(x).$$

The constitutive relationship, which once more follows from Hooke's law stating that $\sigma = E\varepsilon$, must hold for each infinitesimal longitudinal slice of the beam in vertical and lateral directions, giving

$$\sigma(x,y) = E\varepsilon(x,y) = E\frac{\mathrm{d}u(x,y)}{\mathrm{d}x} = E\frac{y\mathrm{d}\theta(x)}{\mathrm{d}x} = E\kappa(x)y.$$

Multiplying by the eccentricity y from the beam axis and the equivalent width of the beam $\delta z(y)$, and integrating over the height of the beam H yields

$$M(x) = \int_H y\delta z(y)\sigma(x,y)\mathrm{d}y = E\kappa(x)\int_H y^2\delta z(y)\mathrm{d}y = EI\kappa(x).$$



The equilibrium relationship states that each infinitesimal slice of the beam in longitudinal direction must on the one hand be in vertical force equilibrium,

$$-V(x) - q(x)\partial x + V(x) + \partial V(x) = 0$$
$$lim_{\partial x \to 0} \frac{\partial V(x)}{\partial x} = \frac{\mathrm{d}V(x)}{\mathrm{d}x} = q(x),$$

and on the other hand be in moment equilibrium around the lateral axis on a point on the beam axis on the right side of the infinitesimal slice, assuming that $\lim_{\partial x \to 0} \partial x^2 = 0$,

$$-M(x) + V(x)\partial x + ½q(x)\partial x^2 + M(x) + \partial M(x) = 0$$
$$lim_{\partial x \to 0} - \frac{\partial M(x)}{\partial x} = -\frac{\mathrm{d}M(x)}{\mathrm{d}x} = V(x).$$

Combining the various relationships yields the strong form of the governing equation for the Euler-Bernoulli Beam,

$$-EI\frac{\mathrm{d}^4 v(x)}{\mathrm{d}x^4} + q(x) = -EIv''''(x) + q(x) = 0.$$

The weak form is once more obtained by multiplying the strong form by a virtual displacement $\partial v(x)$ and then integrating over the domain Ω,

$$\int_\Omega [-EIv''''(x) + q(x)]\partial v(x)\mathrm{d}x = 0.$$

For the beam element, the integration is separated by parts twice to recover the boundary terms Γ for both force and moment Neumann boundary conditions. Subsequent substitutipn with the appropriate weight and trial functions, assuming that $n(x)EI\Sigma_j c_j \phi_j'''(x) = F_i$ and $n(x)EI\Sigma_j c_j \phi_j''(x) = T_i$, yields

$$\Sigma_j[\int_\Omega EI\phi_i''(x)\phi_j''(x)\mathrm{d}x]v_j = \int_\Omega q(x)\phi_i(x)\mathrm{d}x + F_i\phi_i(x)|_{x=\Gamma} + T_i\phi_i'(x)|_{x=\Gamma}.$$

The second-order derivative on the left-hand side calls for shape functions with C_2-continuity, and thus third-order polynomials. The right-hand side Neumann boundary conditions are independent, calling for two shape functions per node. They require that the first shape function for each node must be 1 on the coordinate corresponding to the node and zero on the other, while its derivatives must be zero on both. The second shape function must be zero for both coordinates, while its derivative must be 1 on the corresponding coordinate and zero on the other. Combining everything ultimately results in the left-hand stiffness matrix,

$$EI\int_L \phi_1''(x)\phi_j''(x)\mathrm{d}x = \frac{EI}{l^3}\begin{bmatrix} 12 & 6l & -12 & 6l \\ 6l & 4l^2 & -6l & 2l^2 \\ -12 & -6l & 12 & -6l \\ 6l & 2l^2 & -6l & 4l^2 \end{bmatrix}.$$

Box 4.4: The stiffness matrix derivation for the two-node Euler-Bernoulli element

4.3 Model declaration

NODES

As the atomic part in a finite element system, the node is located in N-dimensional space, and is associated with any number of degrees of freedom, each of which themselves is an M-dimensional value. In physical terms, taking the example of a three-dimensional truss, a node is located at a location (x, y, z), and its associated degree of freedom is the displacement (u_x, u_y, u_z).

FORCES

A force is applied to a degree of freedom, and forms the perturbing energy for which it is the analysis' goal to ultimately find an equilibrium state of the system's degrees of freedom *{MATH}*.

CONSTRAINTS

In mathematical terms, a boundary constraint forms a counteraction to a certain displacement in a degree of freedom. The counteraction is equivalent to an external force proportional to the inverse product of a deviation and the constraint's stiffness property. This stiffness may range anywhere between zero and infinity, yielding the expression

$$\mathbf{f}_c = -k_c \mathbf{u}_c$$

where $\mathbf{f}_c$ is the equivalent force vector, k_c is the stiffness property, and $\mathbf{u}_c$ is the deviation. Depending on the constraint properties, the latter may be unequal to, but still a function of the displacement $\mathbf{u}$ itself.

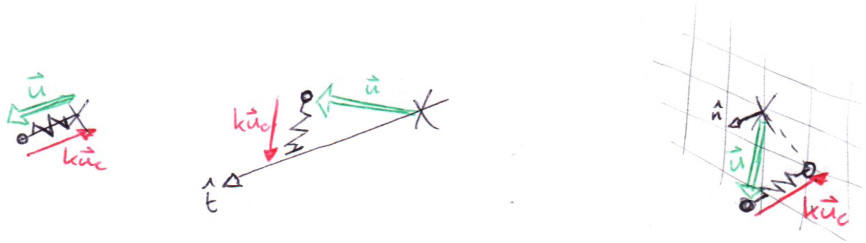


Figure 4.1: Point, line and plane constraints, resp. constraining 3, 2 and 1 degrees of freedom

In physical terms, again taking the three-dimensional truss as an example, a constraint's counteraction will be proportional to the inverse product of its stiffness and (also see Figure 4.1):

- in the case of a **point constraint**, a deviation equal to the total displacement—effectively fixing the degree of freedom in space;
- in the case of a **line constraint**, a deviation equal to the component of the displacement orthogonal to the line direction—effectively constraining the degree of freedom parallel to the line direction;
- in the case of a **plane constraint**, a deviation equal to the component of the displacement parallel to the plane normal—effectively constraining the degree of freedom orthogonal to the plane normal.

A (less than infinite-stiffness) constraint results in displacement-dependent force terms on the right-hand side vector in the global linear system. These can be expressed the product of a stiffness matrix and the displacements $\mathbf{K}_k\mathbf{u}$. Because of the dependence, these constraint stiffness matrices must be incorporated into the global stiffness matrix, $\mathbf{K} + \Sigma\mathbf{K}_{k,i}$. As such, these constraints can be considered equivalent to single-node elements (see Box 4.5).

The element equivalence becomes apparent from the stiffness matrix of the point constraint, which counteracts deviation from a point (including only the relevant portions of the displacement vector):

$$\begin{aligned}\mathbf{f}_c &= -k_c\mathbf{u}_c = -k_c\mathbf{u} \\ &= -k_c\mathbf{I}\mathbf{u} \\ &= -k_c\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{pmatrix}u_x\\ u_y\\ u_z\end{pmatrix}.\end{aligned}$$

The same stiffness matrix for the line constraint, which counteracts deviation from a line along $\mathbf{t}$, is

$$\begin{aligned}\mathbf{f}_c &= -k_c\mathbf{u}_c = -k_c[\mathbf{u} - (\mathbf{u}\cdot\mathbf{t})\mathbf{t}] \\ &= -k_c(\mathbf{I} - \mathbf{t}\mathbf{t}^{\mathrm{T}})\mathbf{u} \\ &= -k_c\begin{bmatrix}1-t_x^2 & -t_xt_y & -t_xt_z\\ -t_yt_x & 1-t_y^2 & -t_yt_z\\ -t_zt_x & -t_zt_y & 1-t_z^2\end{bmatrix}\begin{pmatrix}u_x\\ u_y\\ u_z\end{pmatrix}.\end{aligned}$$

And finally for the plane constraint, which counteracts deviation from a plane normal to $\mathbf{t}$,

$$\begin{aligned}\mathbf{f}_c &= -k_c\mathbf{u}_c = -k_c(\mathbf{u}\cdot\mathbf{n})\mathbf{n}] \\ &= -k_c(\mathbf{n}\mathbf{n}^{\mathrm{T}})\mathbf{u} \\ &= -k_c\begin{bmatrix}n_x^2 & n_xn_y & n_xn_z\\ n_yn_x & n_y^2 & n_yn_z\\ n_zn_x & n_zn_y & n_z^2\end{bmatrix}\begin{pmatrix}u_x\\ u_y\\ u_z\end{pmatrix}.\end{aligned}$$

Box 4.5: The finite-stiffness constraint stiffness matrices

As the stiffness property of a constraint increases and approaches infinity, the deviation will approach zero and the constraint will become rigid. Effectively this means that in the case of deviation normal to a plane, displacement will be constrained to that plane. It will be constrained to a line in the case of deviation orthogonal to that line, or constrained to a point in the case of deviation from that point. Unfortunately, as large variations in stiffnesses (or more precisely, large variations in stiffness matrix elements) cause numerical instability, it is not practical to represent (commonly occurring) rigid constraints this way.

To circumvent numerical issues, it may be realised that in mathematical terms a rigid constraint reduces the dimensionality of the degree of freedom it acts upon. And with that, it reduces the number of linearly independent equations that constitute the linear system. Consequently, this allows the elimination of one or more equations by incorporating linear combinations of them into other equations. In terms of the stiffness matrix, the elimination is achieved through a series of unipotent (or *elementary row-addition*) matrix operations of the form $\mathbf{K}_r = (\Pi_i\mathbf{U}_i^{\mathrm{T}})\mathbf{K}(\Pi_i\mathbf{U}_i)$, where Π_i implies a product series and $\mathbf{U}_i$ is a matrix of the form

$$\mathbf{U} = \begin{bmatrix}1 & 0 & 0\\ c & 0 & 0\\ 0 & 0 & 1\end{bmatrix}.$$

In the case of a rigid plane constraint, the dimensionality of the degree of freedom is reduced to two. In the case of a rigid line constraint, it is reduced to one. In the case of a rigid point constraint, it is reduced to zero. In these cases, the number of linearly independent equations in the whole system is reduced by one, two and three, respectively. For the derivation of the elementary matrix operations that achieve these reductions, see *{APP X}*.

ELEMENTS

Elements span nodes, and as such create a dependency between the individual nodes' degrees of freedom through which a change in one degree of freedom will directly translate to a proportional change in the other. The discretization into element is the central concept in the finite element method, and as such the element stiffness matrix has already been covered in the section on the method's theory.

As previously mentioned, it is common practice to derive a parameterized element stiffness matrix offline, prior tot analysis. This only leaves the fast computation of the final local element stiffness matrix using its properties and geometry for the analysis itself, and its rotation and permutation into the global matrix.

Element stiffness matrix rotation, which nominally follows the form $\mathbf{K}_e = \mathbf{R}_e^{\mathrm{T}}\mathbf{k}_e\mathbf{R}_e$, can be done quickest using a block-based method. In this method, each block of which the sum of the products of the various stiffness relations and associated orientation vectors' outer products

$$k\mathbf{v}\mathbf{v}^{\mathrm{T}} = k\begin{bmatrix} v_x^2 & v_x v_y & v_x v_z \\ v_y v_x & v_y^2 & v_y v_z \\ v_z v_x & v_z v_y & v_z^2 \end{bmatrix} = k\mathbf{V}$$

> As an example, the stiffness matrix for the simple bar element rotated to global coordinates is
>
> $$\mathbf{K} = \mathbf{R}^{\mathrm{T}}\mathbf{k}\mathbf{R} = \begin{bmatrix} k_{11}\mathbf{V} & k_{12}\mathbf{V} \\ k_{21}\mathbf{V} & k_{22}\mathbf{V} \end{bmatrix}$$

Box 4.6: The element stiffness matrix of the bar element in the global coordinate system

4.4 Stiffness matrix assembly

In terms of implementation, an intermediary step of the finite element method generally involves solving a matrix, often referred to as the stiffness matrix. The global stiffness matrix which describes the system as a whole is a summation of its constituent elements' individual stiffness matrices, permuted in such a way that its local degrees of freedom correspond to their global counterparts. This summation process is referred to as stiffness matrix assembly, and is mathematically expressed as

$$\mathbf{K} = \sum_i \mathbf{K}_i = \sum_i \mathbf{P}_i^{\mathrm{T}}\mathbf{K}_{e,i}\mathbf{P}_i = \sum_i \mathbf{P}_i^{\mathrm{T}}\mathbf{R}_i^{\mathrm{T}}\mathbf{k}_{e,i}\mathbf{R}_i\mathbf{P}_i$$

where $\mathbf{K}$ is the global stiffness matrix and each $\mathbf{K}_i$ is the permuted matrix of element i. Each $\mathbf{K}_i$ is the symmetrical permutation expressed in the binary permutation matrix $\mathbf{P}_i$ of the global element stiffness matrix $\mathbf{K}_{e,i}$. Each $\mathbf{K}_{e,i}$ in turn is the symmetric rotation to the global coordinate systems expressed in the rotation matrix $\mathbf{R}_i$ of the local element stiffness matrix $\mathbf{k}_{e,i}$.

4.5 Cholesky decomposition

As size increases, solving a matrix rapidly becomes a computationally expensive operation. The various algorithms that are in common use are all of cubic complexity with regard to matrix size. That is, they have an algorithmic complexity (i.e. the number of arithmetic operations) of $O(n^3)$, where n is the number of equations in the system (i.e. the number of rows in the matrix).

Using Cholesky decomposition or factorization to solve a positive-definite symmetric matrix involves computing the Cholesky factor, which is the lower-triangular matrix of which the product of it and its transpose is equal to the initial matrix. The system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$ with given positive-definite matrix $\mathbf{A}$ thus has a Cholesky factor $\mathbf{L}$ which satisfies $\mathbf{L}^{\mathrm{T}}\mathbf{L} = \mathbf{A}$. Consequently, $\mathbf{L}^{\mathrm{T}}\mathbf{L}\mathbf{x} = \mathbf{b}$ or—with the introduction of $\mathbf{y}$ as an intermediary

result—$\mathbf{Lx} = \mathbf{y}$ and $\mathbf{L}^{\mathrm{T}}\mathbf{y} = \mathbf{b}$. Using the factor, it becomes trivial to compute $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} = (\mathbf{L}^{\mathrm{T}})^{-1}\mathbf{L}^{-1}\mathbf{b}$, involving consequently a lower and an upper triangular matrix solution.

It should be noted that the Cholesky decomposition, with its fill-in, is best suited to linear structures. Its benefits are diminished with circular structures, that do not have a clear beginning or end, or star-shaped structures, that contain a multitude of lobe-like protrusions around a central connecting core.

4.5.1 IMPLEMENTATION

STORAGE

During the solution process, the stiffness matrix and force and solution vectors are entirely held in memory. Its elements are stored separately in a diagonals array and an off-diagonals, or envelope array. Exploiting the symmetry of the stiffness matrix, only the lower half of the envelope is stored, in row-major order (i.e. contiguously per row, one row after another; see Figure 4.2).

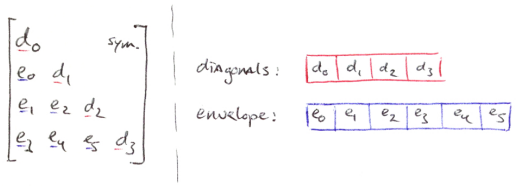


Figure 4.2: The separated diagonals and row-major symmetric envelope storage scheme.

Since stiffness matrices are commonly sparse (i.e. have many zero elements), this sparsity can be further exploited to store only the portions of the matrix rows starting from the first non-zero element. An additional indices array is now required to define rows' starting point in the envelope array (see Figure 4.3). In this scheme, the zero elements after the first non-zero will still be stored. Further savings could be achieved by eliminating these as well, but anticipating fill-in from the decomposition algorithm (i.e. the algorithm will turn zero elements into non-zero elements), this complicates matters greatly, and it was opted to accept the inner sparsity.

Figure 4.3: The same storage scheme as above, now exploiting sparsity to store the the non-zero portions of the envelope (and inner zeros, in anticipation of fill-in).

LOWER TRIANGULAR MATRIX SOLUTION

The lower triangular matrix solution algorithm solves a system of equations of the form $\mathbf{Lx} = \mathbf{b}$ or $\mathbf{x} = \mathbf{L}^{-1}\mathbf{b}$, where $\mathbf{L}$ is a lower triangular matrix of size n. Disregarding any sparsity, the kernel loop of the algorithm takes the form

$$\begin{aligned} &\textbf{for } i \textbf{ in } [0, n) \\ &\quad x_i \leftarrow \frac{(b_i - \mathbf{x}_{0\ldots i}\mathbf{L}_{i,0\ldots i})}{L_{i,i}}. \end{aligned}$$

Employing sparsity using the bandwidth of each of the matrix rows, the kernel loop is rewritten as

$$
\begin{aligned}
&\textbf{for } i \textbf{ in } [0, n) \\
&\quad j \leftarrow i - band_i \\
&\quad x_i \leftarrow \frac{(b_i - \mathbf{x}_{j\ldots i}\mathbf{L}_{i,j\ldots i})}{L_{i,i}}.
\end{aligned}
$$

For a total of n loops, with each loop taking $3i + 1$ arithmetic operations, this algorithm amounts to $O(n^2)$ operations. Employing sparsity, this becomes an upper limit, while the general case complexity will be a fraction of that.

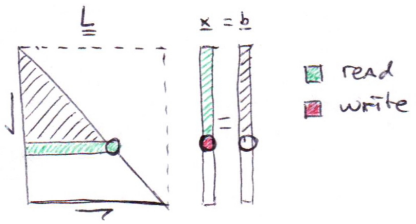


Figure 4.4: The progressive (optionally in-place) lower-solve algorithm access pattern.

This algorithm uses a straightforward access pattern (see Figure 4.4). It starts at the beginning of both the matrix and the vector arrays, and moves through each row, diagonal, and vector element in storage order. For every iteration i, the algorithm reads from the entire current matrix row $\mathbf{L}_{i,0\ldots i}$ and its diagonal element $L_{i,i}$, as well form as the solution vector up to the current element $\mathbf{x}_{0\ldots i}$ and the right-hand side vector's current element b_i. Then, it only writes to the current solution vector element x_i. This pattern, with one write at the end of the iteration and no reads from previous right-hand side vector elements, allows the algorithm to efficiently run in-place.

CHOLESKY FACTORISATION

The Cholesky factorisation algorithm computes a factor matrix $\mathbf{L}$ for a positive-definite symmetric matrix $\mathbf{A}$ such that $\mathbf{A} = \mathbf{L}^{\mathrm{T}}\mathbf{L}$. Disregarding sparsity first like before, the kernel loop of the algorithm takes the form

$$
\begin{aligned}
&\textbf{for } i \textbf{ in } [0, n) \\
&\quad \mathbf{L}_{i,0\ldots i} \leftarrow \mathbf{L}_{0\ldots i}^{-1}\mathbf{A}_{i,0\ldots i} \\
&\quad L_{i,i} \leftarrow \sqrt{A_{i,i} - \mathbf{L}_{i,0\ldots i}^2},
\end{aligned}
$$

where $\mathbf{L}_{0\ldots i}$ in the first step in the kernel is the computed factor up to, but not including, row i. This step uses the lower triangular matrix solution algorithm described above for the matrix solution that computes the off-diagonal factor row.

Again, employing sparsity using the bandwidth of the matrix rows, the kernel loop is rewritten as

$$
\begin{aligned}
&\textbf{for } i \textbf{ in } [0, n) \\
&\quad j \leftarrow i - band_i \\
&\quad \mathbf{L}_{i,j\ldots i} \leftarrow \mathbf{L}_{j\ldots i}^{-1}\mathbf{A}_{i,j\ldots i} \\
&\quad L_{i,i} \leftarrow \sqrt{A_{i,i} - \mathbf{L}_{i,j\ldots i}^2}.
\end{aligned}
$$

For a total of n loops, with each loop taking around $O(n^2)$ arithmetic operations, this algorithm amounts to $O(n^3)$ operations, but again a fraction of that when employing sparsity.

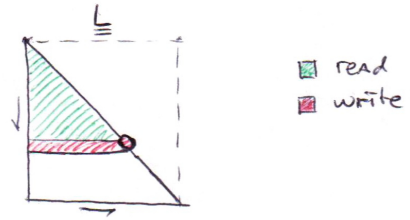


Figure 4.5: The progressive (optionally in-place) factorisation algorithm access pattern, which uses its partial factor in the lower-solve algorithm at every iteration.

This algorithm also starts at the beginning of the matrix envelope and diagonal arrays (see Figure 4.5). Its first iteration $i = 0$ only computes the first diagonal element $L_{0,0}$. Subsequent iterations use the above lower-solve algorithm to compute the current row $\mathbf{L}_{i,0\ldots i}$ from the partial factor up to that iteration $\mathbf{L}_{0\ldots i}$. Thus, these iterations all follow the access pattern described for that algorithm, reading from the current and previous rows and writing back to the current. The final step of an iteration reads from the just-solved row to write to the current diagonal $L_{i,i}$. Since both the lower-solve algorithm and this last step can be run in-place, so can this factorisation algorithm as a whole.

UPPER TRIANGULAR MATRIX SOLUTION

The upper triangular matrix solution, which forms the last step in the positive-definite symmetric matrix solution, solves a system of equations of the form $\mathbf{Ux} = \mathbf{c}$. Disregarding any sparsity, the kernel loop of the algorithm takes the form

$$\begin{aligned}&\mathbf{x} \leftarrow \mathbf{c}\\&\textbf{for } i \textbf{ in } (n, 0]\\&\qquad \mathbf{x}_{0\ldots i} \leftarrow \mathbf{x}_{0\ldots i} - \frac{x_i}{U_{i,i}}\mathbf{U}_{0\ldots i,i}\end{aligned}$$

where it should be noted that the solution vector is computed iteratively, instead of sequentially.

Again, employing sparsity using the bandwidth of the matrix rows, the kernel loop is rewritten as

$$\begin{aligned}&\mathbf{x} \leftarrow \mathbf{c}\\&\textbf{for } i \textbf{ in } (n, 0]\\&\qquad j \leftarrow i - band_i\\&\qquad \mathbf{x}_{j\ldots i} \leftarrow \mathbf{x}_{j\ldots i} - \frac{x_i}{U_{i,i}}\mathbf{U}_{j\ldots i,i}\end{aligned}$$

For a total of n loops, with each loop taking around $2(n - i)$ arithmetic operations, this algorithm amounts to _ $O(n^2)$ total operations, but again a fraction of that when employing sparsity.

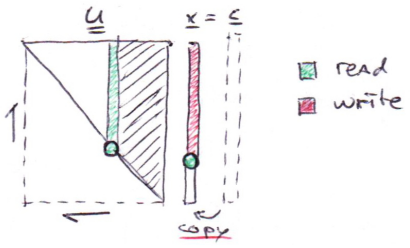


Figure 4.6: The iterative in-place upper-solve algorithm access pattern.

This algorithm departs from the straightforward access pattern of the previous to and starts at the end of the matrix envelope and diagonal, and solution vector arrays (see Figure 4.6). This algorithm naturally runs

iteratively, starting with the right-hand side vector $\mathbf{c}$ as the initial solution vector $\mathbf{x}$. Subsequently, the algorithm reads from the current matrix column $\mathbf{U}_{0\ldots i,i}$ and diagonal $U_{i,i}$, and the current solution vector element x_i to update $\mathbf{x}_{0\ldots i}$ at every iteration i. This access pattern allows the algorithm to use a lower triangular matrix storage scheme, but forces it to run in-place (or otherwise requires an initial *copy* step).

4.5.2 NODE NUMBERING

The number of computational operations per equation can vary greatly depending on the solution algorithm, and the way it is implemented. This is where node numbering plays an important role.

In addition to being symmetric, the matrices that are solved in the finite element tend to be sparse. That is, an overwhelming majority of the—off-diagonal—elements are zero. In mathematical terms, given that each element is at the intersection of a row and a column, and that each row and each column relates to one of the system's degrees of freedom, a zero off-diagonal element implies that the the two related degrees of freedom are independent of each other. In physical terms, it implies that one related displacement does not directly—though usually still indirectly—translate to another.

In terms of computational cost, an arithmetic operation involving a zero element counts as any other. But given that it involves a zero element the result is already known to be zero as well. Ultimately, the goal of node numbering is to populate a matrix in such a way that as much of these operations can be avoided as possible. This can reduce the computational cost of solving a sufficiently sparse matrix by an order of magnitude.

(REVERSE) CUTHILL-MCKEE

Different numbering strategies are suitable to different matrix solution algorithms. In the case of Cholesky decomposition, where it is known that the Cholesky factor matrix will have the same bandwidth profile as the corresponding triangular half of the original matrix, the foremost goal is to reduce what is referred to as 'fill-in'.

Fill-in is the occurrence of non-zero elements in the factor where there were none in the original matrix. The elements in the factor contain summations of the multiplied elements of two preceding rows, which may contain non-zero elements even when the element in the target position in the original matrix is zero. Fill-in is guaranteed to only occur after the original row's first non-zero element, and as such will not alter the original profile.

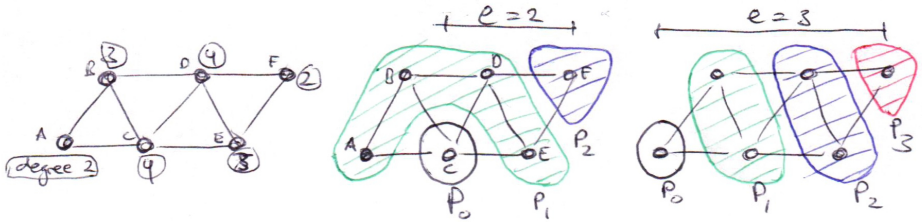


Figure 4.7: Node degree, partitions and eccentricity

A suitable node numbering is attained using the so-called (Reversed) Cuthill-McKee (1969) family of algorithms (the reversed variant usually results in better fill-in). In pseudocode, the algorithm goes through the following steps:

1. **Let** U **be** the set of unnumbered nodes
2. **Let** n_c **be** a suitable root node in U *(note 1)*
3. **Let** N **be** a list initially containing only n_c
4. **While** U is not empty
    i. **Let** A **be** the subset of U adjacent to n_c
    ii. Move nodes from A to N in order of increasing degree *(note 2)*
    iii. **Let** n_c **be** the next numbered node in N
5. Number the nodes of N in reverse order

The above algorithm contains two unexplained references, to root node suitability and node degree. The latter, the degree of a node, is proportional to the number of nodes adjoining that node. With regard to the former, a suitable root node is loosely defined as a node with high eccentricity. The eccentricity of a node is proportional to the maximum number of nodes between that node and any of the others.

Logically, a most suitable root node would be any nodes with highest eccentricity, the search for which is an optimization problem. For many cases, there will only be a small performance difference between the most suitable root node and another with relatively high suitability. For the general case, it is more important to find a relatively suitable root node in a short time. The following algorithm has proven to reliably meet these goals, yielding a decently suitable root node in relatively short time.

1. **Let** n_r **be** any node in U
2. **While** no root node has been found
    i. **Let** P_0 **be** the partition (i.e. set) containing only n_r
    ii. **For** increasing i, **while** unpartitioned nodes remain
        a. **Let** P_i **be** the set of nodes adjacent to those in P_{i-1}
    iii. **Let** the eccentricity e_r of n_r **be** the number of partitions
        a. **If** e_r **is** lower than previous eccentricities, **yield** n_r
        b. **Else**, **let** n_r **be** the the highest-degree node in P_e

It should be evident that the above algorithm can prematurely yield a non-optimal root node upon encountering a local optimum. In the general case, however, these nodes will still result in sufficiently slim matrices that its speed becomes an increasingly important factor.

4.6 Post-processing

After solving the reduced linear system, an additional number of post-processing steps is required to compute the full, non-reduced system. These steps involve restoring the eliminated displacements, then computing the eliminated reaction forces, and finally computing the element forces.

DISPLACEMENTS

As implemented, the vector resulting from the solution of the reduced stiffness matrix–force vector problem is not yet the final displacement vector. This is due to the way in which the rigid constraints were applied, where linearly dependent equations were eliminated resulting in the associated displacements to be set to zero. These eliminated displacements must be restored (except for the case of the point constraint, where the displacements remain zero). This is trivial, since by definition the eliminated displacements are linear equations of the remaining ones.

To restore the eliminated equations, the elementary matrix operations that were first applied to the stiffness matrix and force vector must be applied to the solution vector (or *reduced displacement vector*) $\mathbf{u}_r$ in reverse. This operation takes the form $\mathbf{u} = (\mathbf{\Pi}_i \mathbf{U}_i)(\mathbf{\Pi}_i \mathbf{U}_i^{\mathrm{T}})\mathbf{u}_r$ (see also *{SECTION 4-3c}* on constraints application, and Appendix A on the derivation thereof).

REACTION FORCES

As implemented, the force vector that is used to solve the displacements is reduced to contain only the known, applied forces. The full force vector also contains the reaction forces at the constrained nodes, and is trivial to compute once the final displacement vector is known. First considering the case with rigid constraints only, this computation involves multiplying the non-reduced global stiffness matrix with the non-reduced displacement vector, $\mathbf{f} = \mathbf{K}\mathbf{u}$.

When finite stiffness (spring) constraints are applied, their associated reaction forces must additionally be computed by summing up their associated stiffness matrices and multiplying the result with the non-reduced displacement vector, $\mathbf{f}_k = (\sum_i \mathbf{K}_{k,i})\mathbf{u}$.

ELEMENT FORCES

Element forces are trivially computed by individually multiplying the elements' associated stiffness matrices (rotated, permuted into the global coordinate system) with the non-reduced displacement vector, $\mathbf{f}_{e,i} = \mathbf{K}_i\mathbf{u} = (\mathbf{P}_i^{\mathrm{T}}\mathbf{R}_i^{\mathrm{T}}\mathbf{k}_{e,i}\mathbf{R}_i\mathbf{P}_i)\mathbf{u}$. This force vector can be easily rotated to align with the element's local coordinate system using the inverse rotation $\mathbf{R}_e\mathbf{R}_e^{\mathrm{T}}$.

4.7 Validation

The FEA solver will be validated using four structural models: two bar models, the elements of which have only axial stiffness, and two beam models, the elements of which additionally have flexural and torsional stiffnesses. The first pair of cases validates the extension bar element, using a simple single-bar model and a full truss model. The second pair validates the extension, flexion and torsion beam element, using a simple single-beam cantilever model and a full frame model. The simple cases are validated by rules of thumb, while the complex cases are validated by comparison with the analysis results of the certified structural analysis software package *GSA* (Oasys, 2013).

Each model will be solved twice: once aligned with the $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ axes of the cartesian coordinate system, and once aligned with the $\mathbf{x}'$, $\mathbf{y}'$, and $\mathbf{z}'$ axes of a rotated coordinate system. Obviously, the resulting solution vectors should be (triplet-wise) rotated, but equivalent (and the invariants equal). A 3-vector in the global coordinate system $\mathbf{v}'$ is related to its rotated coordinate system counterpart $\mathbf{v}$ through the identity $\mathbf{v}' = \mathbf{R}'\mathbf{v}$ (e.g. $\mathbf{x}' = \mathbf{R}'\mathbf{x} = \mathbf{R}'(1,0,0)^{\mathrm{T}}$), which uses the transpose $\mathbf{R}' = \mathbf{R}^{\mathrm{T}}$ of the rotation matrix $\mathbf{R}$:

$$\mathbf{R} = \begin{pmatrix} \mathbf{x}' & \mathbf{y}' & \mathbf{z}' \end{pmatrix}^{\mathrm{T}} = \frac{1}{6}\begin{bmatrix} x'_x & x'_y & x'_z \\ y'_x & y'_y & y'_z \\ z'_x & z'_y & z'_z \end{bmatrix}.$$

For each of the validation cases, the same randomly rotated coordinate system defined by the rotation matrix $\mathbf{R}$ was used in addition to the global coordinate system (rounded off to six significant digits):

$$\mathbf{R} = \begin{bmatrix} 0.953463 & 0.095346 & 0.286039 \\ -0.135233 & 0.983142 & 0.123062 \\ -0.269483 & -0.156017 & 0.950283 \end{bmatrix}$$

ISOLATED DISPLACEMENT MODES

The following validation cases are chosen to validate both relatively simple unit models with unit forces affecting various degrees of freedom, and more complex cases involving a multitude of forces. However, in order to ascertain that the analysis is correct in principle, it is important to validate each element's individual displacement modes.

A displacement mode corresponds to an displacement $\lambda\hat{\mathbf{u}}_j$ with exactly one displaced degree of freedom j, or

$\lambda\hat{u}_i = \lambda\delta_{i,j}$, where $\delta_{i,j} = 1$ when $i = j$ and $\delta_{i,j} = 0$ otherwise (see Figure 4.8). An element has exactly one displacement mode for each of its degrees of freedom. Each unit displacement mode $\hat{\mathbf{u}}_j$ corresponds to a unique combination of external forces $\hat{\mathbf{F}}_j = \mathbf{K}_e\hat{\mathbf{u}}_j$ (i.e. it corresponds a column of the element stiffness matrix).

Validation requires that an element constrained to allow displacement only for the appropriate degree, to which the isolated corresponding force is applied, elicits the appropriate reaction forces. The solver satisfies the requirement.

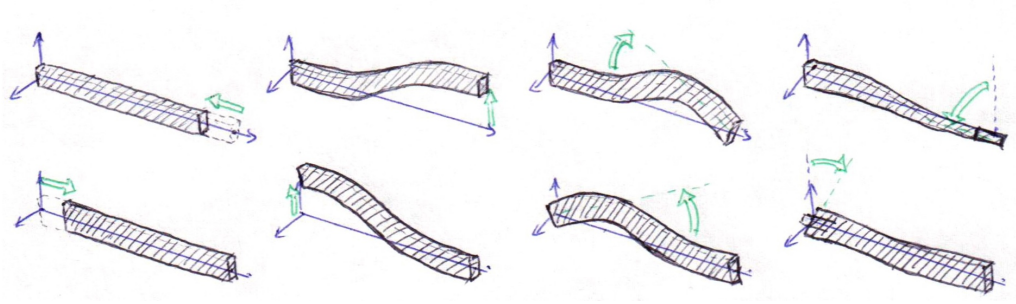


Figure 4.8: Unit displacement modes for (the minor-axis flexural modes are omitted)

4.7.1 EXTENSION BAR

The first validation case is a simple extension bar model. This case validates the extension bar element and the fixed and parallel displacement constraints, as well as the nodal force.

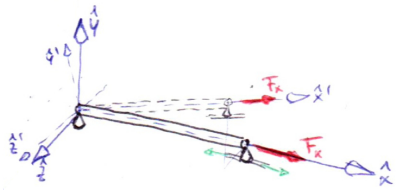


Figure 4.9: The axis-aligned and rotated extension bar models

The model consists of one extension bar element (see Figure 4.9), rigidly constrained (with respect to displacement) at the origin of the coordinate system, with a unit length of $1\,\mathrm{m}$, aligned with and constrained at the far end along the $\mathbf{x}$ vector. Its axial stiffness EA_{xx} is $1\,\mathrm{kN}$. A single force F_x is applied to the far end of the bar, with a magnitude of $1\,\mathrm{kN}$, directed along the bar axis away from the origin.

The displacement at the far end of the bar follows from the simple rule of thumb:

$$u_x = \frac{F_x l}{EA_{xx}} = 1\,\mathrm{m}.$$

Obviously, the same value must result from the rotated analysis in its local coordinate system, which can be related to the un-rotated coordinate system using $\mathbf{R}$:

$$\mathbf{u}' = \mathbf{R}'\begin{pmatrix} u_x \\ 0 \\ 0 \end{pmatrix} = u_x\mathbf{x}' = \mathbf{x}'.$$

Such a result indeed follows from the solver (see *{TABLE ii-3-2-7-1}*), validating it at least for this simple case.

	u_x	u_y	u_z
Axis-aligned	1.0	0.0	0.0
Rotated	0.953463	0.095346	0.286039

Table 4.1: Solver results for the extension bar model

4.7.2 TRUSS

In addition to the simple extension bar model, a larger truss model is used to validate the interaction between multiple bar elements, and the orthogonal constraint.

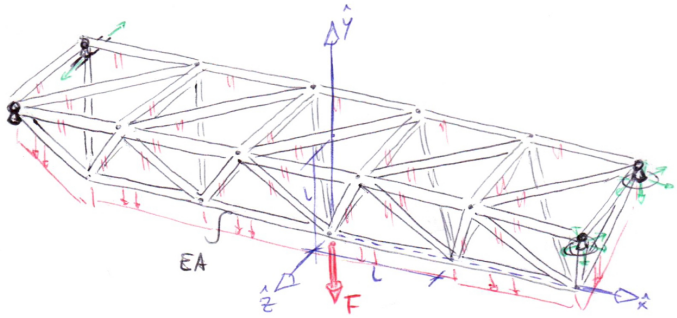


Figure 4.10: The truss model

The model consists of a horizontal prismatic three-dimensional truss, supported at its extremes and loaded by distributed downward member (gravity) forces and a single downward force acting on the middle node (see Figure 4.10). Again, the model is analysed both in the axis-aligned and rotated coordinate systems. Where the previous simple case was validated using rules of thumb, this case is validated by comparison with results from GSA, found in Appendix B.

4.7.3 CANTILEVER BEAM

The first of the second pair of validation cases is a simple cantilever beam model. This case validates the Euler-Bernoulli flexion and torsion beam element and the fixed displacement and rotation constraint, as well as the element force and nodal torque.

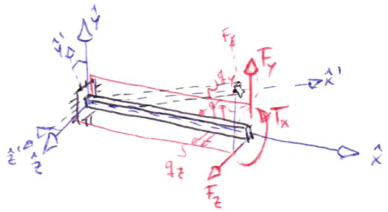


Figure 4.11: The axis-aligned and rotated cantilever beam models

The model consists of one cantilever beam (see Figure 4.11), rigidly constrained (with respect to displacement and rotation) at the origin of the coordinate system, with a unit length of $1\,\mathrm{m}$, aligned with the $\mathbf{x}$ vector. Its major stiffness axis is aligned with the $\mathbf{z}$ vector, and its minor stiffness axis is aligned with the $\mathbf{y}$ vector. Its major and minor bending stiffnesses EI_{zz} and EI_{yy} are 2 and $1\,\mathrm{kNm}^2$, respectively. Its torsional stiffness GA_{xx} is $1\,\mathrm{kNm}^2$. Two forces F_y and F_z, and a torque T_x are applied to the free end of the beam, the forces with a unit magnitude of $1\,\mathrm{kN}$, and the torque with a unit magnitude of $1\,\mathrm{kNm}$. Two additional distributed forces q_y and q_z act along the full length of the beam, both with a unit magnitude of $1\,\mathrm{kN/m}$. Both pairs of forces are directed

perpendicular to the major and minor stiffness axes (and the beam axis), respectively. The torque revolves around the beam axis.

The displacement and rotation at the free end of the beam follows from simple rules of thumb:

$$u_x = 0,\ u_y = \frac{F_y l^3}{3EI_{zz}} + \frac{q_y l^4}{8EI_{zz}} = \frac{11}{48}\ \text{m},\ u_z = \frac{22}{48}\ \text{m};$$
$$\theta_{xx} = \frac{T_x l}{GA} = 1\,\text{rad},\ \theta_{zz} = \frac{F_y l^2}{2EI_{zz}} + \frac{q_y l^3}{6EI_{zz}} = \frac{1}{3}\ \text{rad},\ \theta_{yy} = -\frac{2}{3}\ \text{rad},$$

where the sign of the last rotation is due to the right-handed convention of the coordinate system. Obviously, the same values must result from the rotated analysis in its local coordinate system, which can be related to the un-rotated coordinate system using $\mathbf{R}$:

$$\mathbf{u}' = \mathbf{R}' \frac{1}{48} \begin{pmatrix} 0 \\ 11 \\ 22 \end{pmatrix} = \frac{1}{48}(11\mathbf{y}' + 22\mathbf{z}'),$$
$$\boldsymbol{\theta}' = \frac{1}{3}(3\mathbf{x}' - 2\mathbf{y}' + 1\mathbf{z}').$$

These same values follow from the solver (see Table 4.2), validating it at least for this simple case.

	u_x	u_y	u_z	θ_{xx}	θ_{yy}	θ_{zz}
Axis-aligned	0.0	0.229167	0.458333	1.0	-0.666667	0.333333
Rotated	-0.154504	0.153796	0.463748	0.95379	-0.612087	0.520759

Table 4.2: Solver results for the cantilever beam model

4.7.4 FRAME

In addition to the simple extension bar model, a larger truss model is used to validate the interaction between multiple beam elements.

The model consists of a vertical prismatic three-dimensional frame (see Figure 4.12), supported at its base and loaded by distributed downward member (gravity) forces, and lateral forces at the top. Again, the model is analysed both in the axis-aligned and rotated coordinate systems. Again, this case is validated by comparison with results from GSA, found in Appendix B.

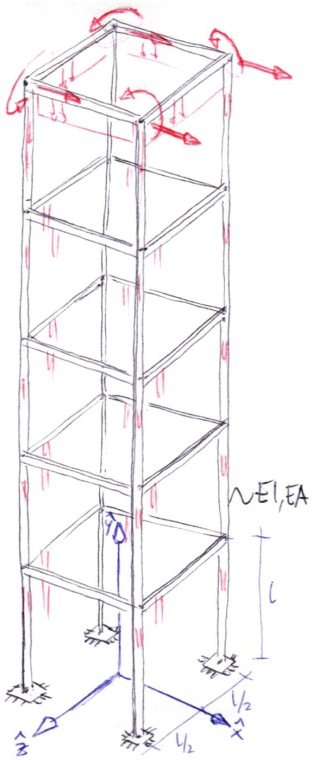


Figure 4.12: The frame model

This page is intentionally left blank.

5 Software architecture

5.1 Overview

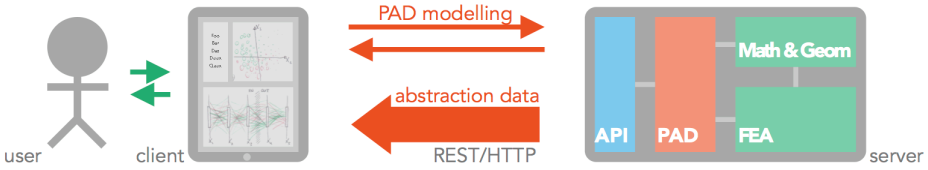


Figure 5.1: The client-server architecture.

The implemented software system accompanying this thesis is architected as a client–server separated application (see Figure 5.1). The server-side system handles finite element analysis and computation, and central parametric and associative model manipulation and storage. The client-side system (the implementation of which falls outside the thesis scope) would provide the user interface.

The next subsections describes the full system, fully detailing the server-side system and summarising the role of a fictitious client-side system. Subsequently, it describes two use cases that illustrate the way users would interact with the system. After that Sections 5.2, 5.3, and 5.4, give further detail of the server-side system, as referenced from the overview.

Frequent use is made of domain-specific software concepts, patterns, and terminology. These are explained in Appendix C.

5.1.1 SERVER-SIDE SYSTEM OVERVIEW

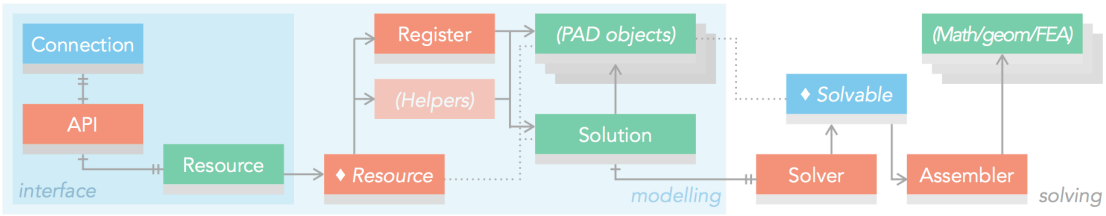


Figure 5.2: A comprehensive class diagram showing the nested MVC architecture (a ♦-diamond prefix indicates an interface; parentheses—possibly in stacked classes—indicate a collection of classes).

The server system is architected in the object-oriented paradigm, following the *model–view–controller* separation of roles. The overall system is a nesting of subsystems separated by the MVC separation. Three distinct subsystems can be identified, each of which in turn is similarly separated (see Figure 5.2; from right to left): the *solving* subsystem, the *modelling* subsystem, and the *application interface* subsystem.

The following overviews give brief descriptions of the role of each subsystem, and how they are related. They refer to more detailed descriptions given in Sections 5.2, 5.3, and 5.4.

SUBSYSTEMS OVERVIEW

Solving subsystem

The first and primary subsystem is the solving system (see Figure 5.3, which is a detail of Figure 5.2), which handles the primary computational function of the system and consists of:

- (model) the finite-element analysis ('FEA') engine and its associated mathematical and geometrical libraries;
- (view) the parametric and associative design ('PAD') modelling graph, and specifically its *Solution* class which represents an instance of a PAD model definition with a concrete set of input parameters and resulting output;
- (controller) the parametric and associative solver engine which interprets the PAD logic to translate and route its input into the FEA engine and the resulting output back into the logic.

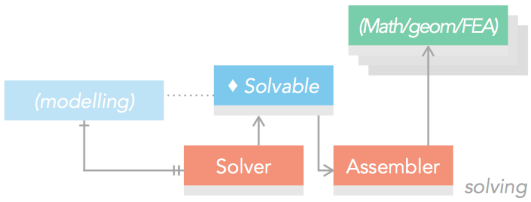


Figure 5.3: Class diagram showing the solving subsystem (and its relation to the *modelling* subsystem).

The FEA engine solves a structural model, which itself consists of entities like elements and nodes, and forces and constraints, all of which rely on vector and matrix computations. Various geometrical entities, such as lines and points, help position the structural entities in space. Each of these entities neatly contains its own logic, and all are therefore implemented in strict object-oriented fashion. This logic is presumed to be self-evident here. The design of the FEA engine itself was described in detail in Chapter 4; the associated mathematical and geometrical libraries are briefly discussed in Section 5.4.

The *solution* resource contains and conveys the user input and computed output of a PAD graph instance. It represents the outcome of applying the logic described by a PAD model to user-defined input. In the simple case, the input may consist of a combination of parameter values that deterministically corresponds to a combination of specified output values. Abstractions, as described in Chapter 3, are achieved by defining the parameters as probability distributions, stochastically sampling a large number parameter value combinations form these distributions, solving the PAD graph instance for each of these combinations, and analysing and interpolating the outcome data to form outcome probability distributions.

The PAD solver system is described below using a sequence diagram, and in further detail in Section 5.2.2.

Modelling subsystem

The *Solution* class, which fulfills the view role in the primary system as described above, is part of the secondary *modelling* subsystem (see Figure 5.4), which defines the PAD graph and handles its associated manipulations. the classes of which themselves are functionally separated along an MVC separation:

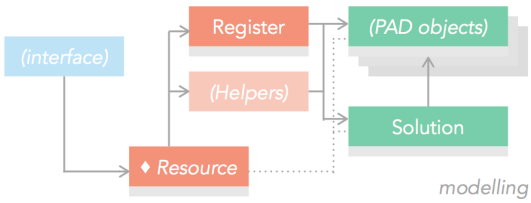


Figure 5.4: Class diagram showing the modelling subsystem (and its relation to the *application interface* subsystem).

- (model) the hierarchical object-relational model, consisting of several related object types that together form the non-looping, directed PAD graph according to the *boxes-and-wires* model (described in Section 5.2.1);
- (view) the application interface which allows manipulation of the PAD graph (see next heading);
- (controller) the per-model interface implementations which interpret the manipulations and the associated storage and serialisation-related helper classes (the latter described in Section 5.2.3).

The primary function of this subsystem is to allow manipulation of the PAD graph through *CRUD* ('create–read–update–delete') operations, which is described below using the sequence diagram. As mentioned, the controller role is fulfilled primarily by model *extensions*, which implement the resource model *interface*. Although these are extensions of the same classes which fulfil the model role, the extension pattern separates these concerns out (over multiple source files), while avoiding the overhead and complexity of an extra layer of

controller classes.

Application interface subsystem

Finally, the view role of the second system is fulfilled by the third *interface* subsystem (see Figure 5.5), which functions as the application interface, or API. This API is designed in the REST architectural style, following a subset of API design guidelines which establish a loosely-coupled interface with goals in mind such as component independence, scalability, and flexibility. The REST API relies on a built-in HTTP/TCP server (described in detail throughout Section 5.4).

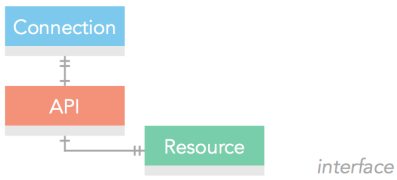


Figure 5.5: Class diagram showing the interface subsystem.

SEQUENCES OVERVIEW

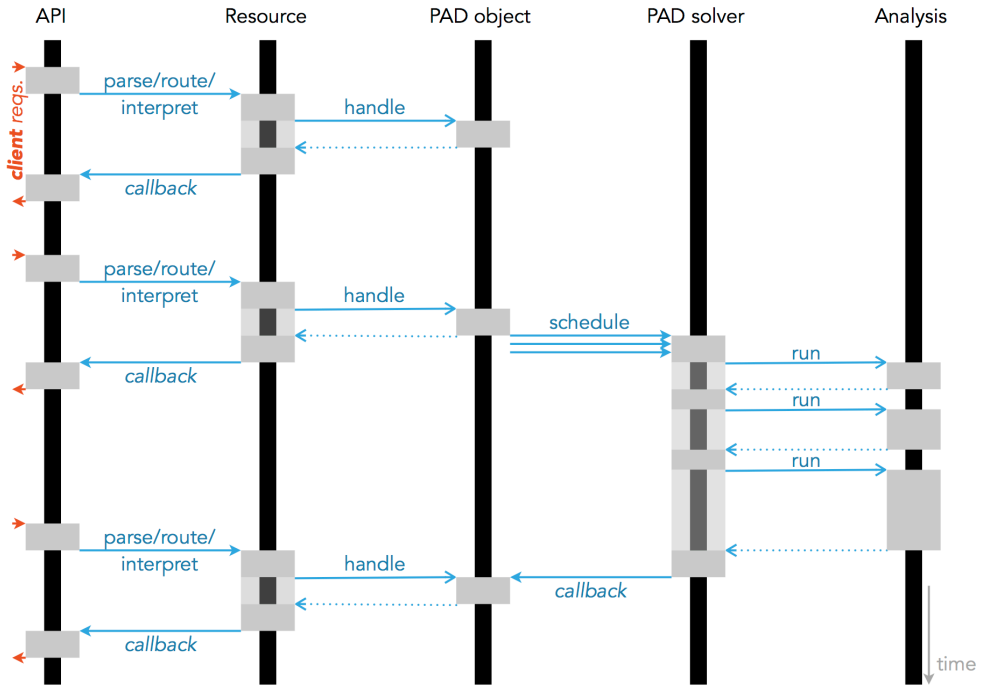


Figure 5.6: A comprehensive sequence diagram showing the (semi-)synchronous *CRUD* and asynchronous scheduling operations following *API* request from a client.

CRUD manipulations

The way the system works is shown by its sequence diagram (see Figure 5.6). Interactions with the user are represented by the red arrows on the left side. The top-most sequence represents the CRUD operations on the PAD resources through the REST API. This sequence is likely repeated several times for different resources as the user is busy creating the parametric and associative model. Each client-side interaction, such as creating a component, or connecting two components together, is followed by a manipulation of the relevant server-side data

Each CRUD manipulation (e.g. reading a definition, creating a model or a parameter, updating an input, deleting a connection) requires an API interaction. Using *cURL* to send HTTP requests, these interactions might be:

```
$ curl -X GET /definition/<id>
$ curl -X POST -d @model.json /definition/<id>
$ curl -X POST -d @parameter.json /model/<id>
$ curl -X PUT -d @input.json /input/<id>
$ curl -X DEL /connection/<id>
```

In this example, *cURL* is used as the client; other clients would interact with the server by generating and exchanging the same HTTP requests.

objects. This sequence runs more or less synchronously (i.e. it returns more or less immediately with results).

Box 5.1: CRUD manipulations using (simplified) *cURL* to generate HTTP requests with *JSON* data.

PAD and FEA computation jobs

The second sequence from the top represents the job scheduling operation. In the REST style, this takes the form of creating a job resource. A REST identifier of the resource is returned immediately, through which its status can be polled periodically. Simultaneously, the creation of the job resource results in the scheduling of a computation job on a queue, which will be asynchronously executed in due time.

Each job consists of solving the PAD model given a set of input parameters, which in turn involves FEM analysis and eventually returns a set of solution values. Once the job completes, its outcome is post-processed, analysed, and finally made available on the job resource, updating its status and allowing access on subsequent polls (represented by the bottom sequence).

Concurrency and parallelisation

The system is designed with multiple jobs (i.e. multiple PAD solutions, each with different sets of input parameters) in mind. As such, an effort was made to accelerate computation by making use of multi-threading processors. To start, portions of jobs (such as PAD solving), as well as the pre- and post-processing of batches of jobs, run concurrently. Overall, however, the nature of the computations demands that each individual job runs more or less serially, so that high efficiency is gained by running a multitude of jobs in parallel.

5.1.2 CLIENT-SIDE SYSTEM OVERVIEW

Implementing a complete client-side application ('client') to interact with the server-side application ('server') described above expressly falls outside the scope of this thesis. That is, while parts of a possible client were implemented in service of illustrating the overall system, it was not a goal of this thesis to integrate those in a complete application. What follows here is a brief discussion of the roles of a client, and the ways in which it interacts with the server.

The role of the client within the overall system is to act as the user interface, and provide the user experience. In narrow terms, the user activities and thus the interactions that the client should enable are modelling and exploration (in terms of the design cycle, *synthesis* and *analysis*, resp.; see Chapter 2). A third user activity is defining the problem's essential parameters and performance metrics (*inference*). More trivial activities which will receive no further discussion are standard application activities like file management (loading and saving), etc.

Modelling

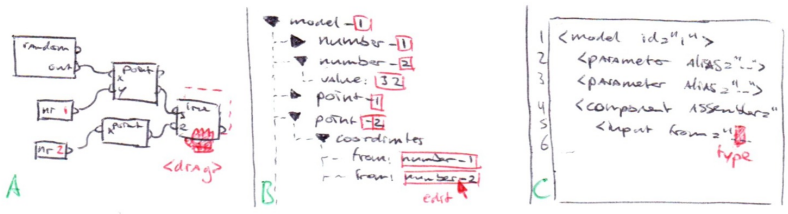


Figure 5.7: Three mockups of possible modelling user interfaces: a) visual *boxes–and–wires* graph canvas; b) hierarchical, collapsable tree; c) representational text editing.

Modelling (*synthesis*) follows the directed-graph paradigm of interconnected parameters and components. As such, a visually very literal possible user interface is the *boxes-and-wires* model, where the user drags components and parameters onto a canvas, and subsequently drags between their outlets to create

connections between them (familiar from e.g. *Grasshopper 3D*; see Figure 5.7–a). Another interface could be the collapsable hierarchical tree (familiar from e.g. outliners or file systems). Possible text-based interfaces could be direct editing of the JSON or XML representations of the PAD graph, or even an interpreted scripting language.

Common to all these possible interfaces is the translation of *view models* into the API-specified representation of a graph object, and the subsequent communication (or transfer) of that representation to the server, where the equivalent object is created, updated, or deleted.

Essential parameters and the metric

A client would have to provide a user interface to define essential parameters (which the abstraction system could vary), and metrics (the outcomes of which the abstraction system would compute). Parameters may be defined as distribution type (e.g. uniform or normal), a mean and variance. Metrics may be defined as direct values of the PAD graph (e.g. a nodal displacement) or a function thereof (e.g. a unity check of a nodal displacement divided by the maximum allowed displacement). This could be implemented property editing forms (e.g. pulls-down menus or input fields), or more visually (e.g. positioning a distribution on a scale by dragging).

Exploration

Data exploration could at its simplest be implemented as series of value tables. However, data visualisations on a comprehensive dashboard would be much more insightful. Such visualisation techniques and the user's interactions with the dashboard are described at length in Section 3.1.3.2 and Section 3.2.3.3.

Miscellaneous

The following is a list of miscellaneous points that serve to further illustrate capabilities enabled by the client–server separation:

- During the modelling phase, after each manipulation, clients may schedule a solver job and receive back feedback illustrating what a change in logic brought on in a solved instance.
- Some client may implement continuous polling to check if an object was externally changed, enabling collaboration.
- Full or partial models (such as a truss model, or a high-rise story section) may be stored in a central library, allowing users to share work.
- To show the potential of a client-server software architecture, a prototype interface of a client implementation for a mobile device was implemented (see Figure 5.8). The prototype was implemented for an iPad device, and allows the manipulation of a PAD graph, where the user drags objects onto a canvas and connections between them using a touch interface.

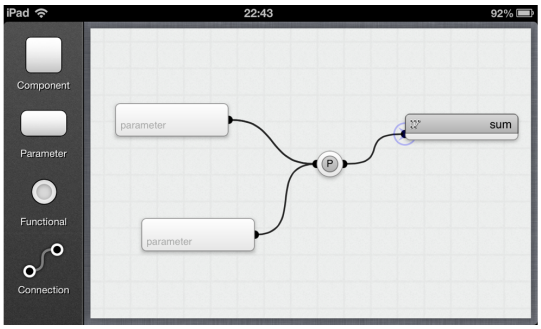


Figure 5.8: An prototype for an iPad client was developed to show the potential of a client-server software architecture, allowing a touch interface to PAD modelling.

5.1.3 USE CASES

To further illustrate how the system works, two use cases are presented. The purpose of these is to define the intended users (termed actors) and their objectives in working with the system. In order to establish continuity, the same use cases are used that were previously described by Rolvink (2010). These illustrate normal use, and adaptation for future purposes. Appendix D describes how the use cases are designed and the template of how they are reported.

USE CASE 1: COMPOSITION AND DEFINITION OF A CONCEPTUAL STRUCTURAL DESIGN

This use case **describes** how a structural engineer can compose and define a conceptual structural design using StructuralComponents 2.0. The goal is to build a model for a tall building structure, assess its behaviour, and adapt the parameters until it fits the design criteria (taken literally from Rolvink, 2010).

> An engineer with a pre-specified structural design problem starts a client application that is connected to the server, starts a new PAD definition, and is presented with the modelling interface. She defines a model, which synchronises with the server, giving her immediate feedback. At one point content with the model, the engineer defines some essential parameters and a metric or two, which she judges are representative of a design goal.
>
> The engineer switches over to the exploration interface, which prompts a computation job on the server. Dashboard graphs are populated with incoming solution data, which interactively let the engineer explore different regions of the solution space. Accepting one particular solution, the engineer switches back to the modelling interface again. She keeps some parts of the model and refines other parts, starting a new iteration with a better insight into the problem.

Box 5.2: The narrative for the first use case.

The **actor** in this use case is a designing structural engineer ('engineer'). The **assumptions** made in this use case are listed below:

- a server instance is up and running, and waiting for interactions;
- the engineer has a working client application on their device;
- the engineer has basic knowledge of PAD modelling and of the client application;
- the engineer has a design specification detailing goals and requirements.

The **course of events** in this use case is:

1. ♦ The engineer starts the client application, which automatically recognises the broadcasted server.
2. ♦ The engineer starts a new PAD definition, and is presented with the modelling interface.
3. The engineer manipulates (e.g. creates, changes) a PAD graph object (e.g. a parameter, a component).
4. The engineer observes an indication that the manipulation was synchronised and accepted by the server.
5. ♦ The engineer observes an immediate-feedback change in the PAD graph instance (e.g. a change in a 3D model; *repeat steps 3–5 a number of times*).
6. The engineer defines two or three essential parameters and one or two metrics which they judge are representative of a design goal.
7. The engineer switches over to the exploration interface, which prompts a computation job on the server.
8. The engineer observes as the dashboard graphs are populated with incoming solution data.
9. The engineer interacts with the graph to explore different regions of the solution space (as they reach

not-yet-computed regions, corresponding jobs are prompted on the server; *repeat steps 8-9 a number of times*).

10. ♦ The engineer accepts one particular solution, which prompts the application to switch over to the modelling interface again, showing the same model (greyed out).
11. ♦ The engineer indicates which parts of the model are to be kept, and refines the other parts (*repeat steps 3–11 a number of times*).
12. ♦ The engineer is left either with a design that achieves the objectives and a sense of the right parameters.

Alternately, the engineer may at some points have found that the requirements could not be achieved. At this point, they might first have gone back one or more iterations to attempt another solution. In the worst case, they might have concluded that the requirements could not be achieved.

The obvious **issue** with this use case is that some of the advanced client-side interactions have not been implemented, and the corresponding steps are therefore partly speculative (indicated with a diamond: ♦).

USE CASE 2: ADAPT AND EXTEND THE SYSTEM FOR FUTURE DESIGN PURPOSES

This use case **describes** how the users of the toolbox can implement new design and analysis methods in the toolbox.

> Some structural design problem requires a non-standard adaptation of the system. As a first attempt, an engineer (with help of a researcher) emulates the desired functionality using standard components, and exposes the solution as a *library* model. It is determined that the solution works and a validation shows that it yields correct results.
>
> However, the engineer concludes that the solution takes the server too long to compute. Together with the researcher, she approaches a computational specialist with the specification of the solution and the objective of improved performance. The specialist implements a native *assembler* to replace the *library* model, which is again validated and made available for use.

Box 5.3: The narrative for the second use case.

The **actors** in this use case are a structural engineer ('engineer'), a researched in the field of structural engineering ('researcher'), and a computational design specialist ('specialist'). The **assumptions** made in this use case are as in the previous case, and further listed below:

- the engineer has a design specification requiring additional, non-standard functionality;
- the engineer has expressed the additional requirements to the researcher, who has come up with a possible solution;
- the specialist has deep knowledge of the system and is capable of extending it with new functionality.

The **course of events** in this use case is:

1. The engineer, with help of the researcher, emulates the desired functionality using standard components, and exposes the solution as a *library* model (following steps similar to those of the first case).
2. The engineer observes that the solution works, the researches validates it correctness, but they conclude that it takes the server too long to compute.

- (Outside the system) the engineer and researcher approach the specialist with the specification of a better-performing solution.

3. The specialist implements a native *assembler* to replace the *library* model, and observes acceptable performance.
4. The engineer and researches again verify that the solution works.
5. The engineer uses the new *assembler* in the design problem.

Alternately, the performance of the *library* model solution can be deemed acceptable at step 2, at which point steps 3–4 can be skipped (unless the solution is observed to affect the system, at which point the specialist can decide to implement the native *assembler* after all).

The same **issues** apply as for the previous case. This case deals less with the core functionality of the system, but illustrates the possibilities when standard functionality is somehow found insufficient.

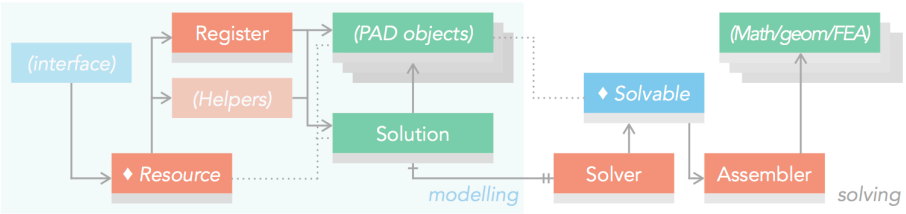


Figure 5.9: PAD engine class diagram: the modelling subsystem allows PAD graph manipulation. It serves as view to the solving subsystem, which computes instantiated design solutions by solving the graph.

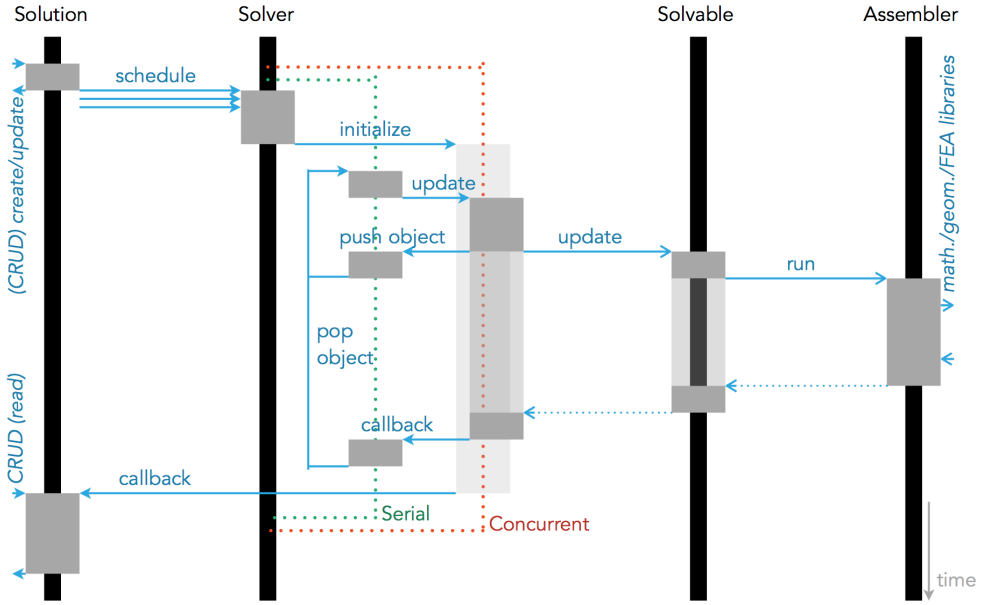


Figure 5.10: PAD engine solving sequence diagram (the vertical time scale is abstract: it indicates sequence durations, though not to exact proportions).

5.2 Parametric and associative engine

The parametric and associative design ('PAD') modelling engine is functionally divided over two subsystems: the modelling subsystem and the solving subsystem (introduced in Section 5.1; also see the class diagram of Figure 5.9). The modelling subsystem consists of the class hierarchy that describes and stores PAD model graphs and allows their manipulation. The solving subsystem takes a graph and uses it to compute instantiated design solutions from given input values (further detailed in Section 5.2.2).

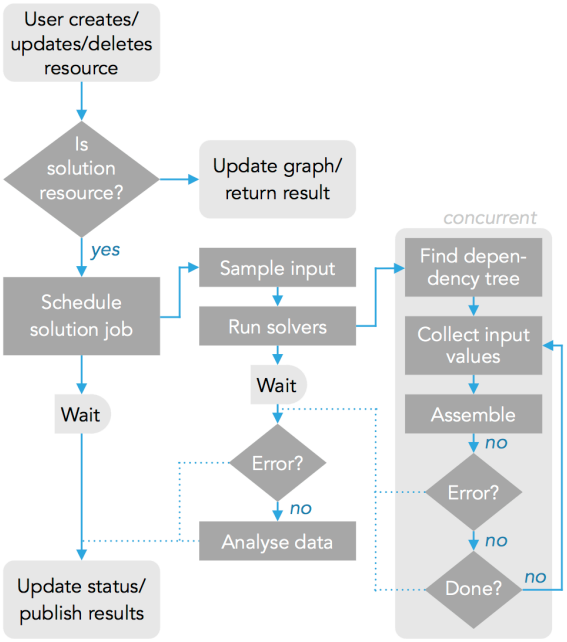


Figure 5.11: PAD engine operational flow chart.

As seen in the flow chart in Figure 5.11, any interaction with either of the systems starts with the creation, update, or deletion of an object. In the case of all object classes except the *Solution*, such an interaction forms a *CRUD* manipulation of a PAD graph. For example, a *Definition* object may be read, or a *Connection* object may be created (see also Box 5.4-a). The result of these operations is returned immediately (see the "Update graph/return result" box in the flow chart). Section 5.2.1 gives a full description of the graph object classes and their properties.

The creation of, or an update to a *Solution* object (see also Box 5.4-b) prompts the scheduling of a PAD solution job (see the "Schedule solution job" box in the flow chart; also see the top-left "schedule" operation in the sequence diagram in Figure 5.10). A *Solution* object is given zero or more *Parameter* objects and one or more *solvables* (i.e. *Component* or *Modifier* objects). It is additionally given *abstraction* settings for each *Parameter* object, such as mean and variance. A solution job will first sample input values for its *Parameters*, then concurrently run a number of PAD solvers, and finally upon finishing store the analysed results to the *Solution* object. Section 5.2.2 gives a full description of the solver architecture.

a) Retrieving the *Definition*:

```
$ curl -X GET /definitions/<id>
```

Might result in a response:

```
HTTP 1.1 200 OK
...
{
  "uri": "/definitions/<id>",
  "alias": "Example definition",
  "models": [...],
  "meta": {
    "author": <name>,
    "creation": <date>
  }
}
```

b) Creating a *Solution*:

```
{
  "abstractions": [{
    "parameter": "/parameters/<id>",
    "mean": 3.14159,
    "variance": 2.61803
  }],
  "solvables": ["/components/<id>"],
  "count": 100
}
$ curl -X POST -d @- /components/<id>
```

Might result in a response:

```
HTTP 1.1 201 Created
Location: /solutions/<id>
```

Box 5.4: Two interaction examples using (simplified) *cURL* and *JSON*: a) reading a *Definition* object, and b) scheduling (creating) a *Solution* job (object; the latter returns a *Location* which can be polled for status changes).

5.2.1 GRAPH OBJECTS AND HIERARCHY

The parametric and associative ('PAD') model is represented a non-looping, directed graph (see Figure 5.12-a). The graph-level object is named a *Model*. Each graph node is represented by a *solvable*, i.e. by either a *Component*, a *Modifier*, or a *Parameter* object. Each graph edge is represented by a *Connection* object, which connects two *solvable* objects through *Input* and *Output* objects. One or more *Model* objects are grouped together in a document-level *Definition* object.

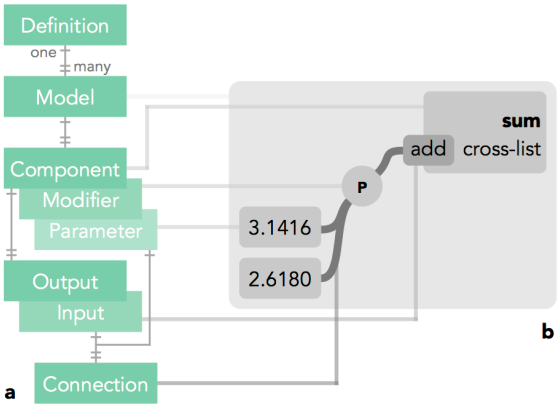


Figure 5.12: a) The graph object class hierarchy tree, and b) a *box-and-wires* representation (the *Connection* class can be seen to laterally connect the *Component*, *Modifier*, and *Parameter* classes).

The hierarchy of objects follows a tree structure, which runs orthogonal to the graph direction (see Figure 5.12-b). That is, its leaves leaves are laterally connected. The tree structure has five basic successive levels. Each class has at least two properties: *type* and *URI*, and may further optionally have a human-readable *alias* property. In addition, some classes may have a *subtype* property. The role of each class, and the additional related properties, is described below in order of hierarchy level (more detailed descriptions can be found along with the complete API documentation in Appendix E):

1. **Definition**: Document-level object which contains one or more *Model* objects. The *Definition* additionally keeps typical document metadata such as creation and modification date stamps, author names, etc.
2. **Model**: Represents the PAD graph describing a design concept, which can be solved to generate an instantiated design solution. The *Model* contains *solvable* objects, of which the *Component* object can itself contain another *Model* object.
3. A *solvable* object classes forming a node in the graph:
    - **Component**: Atomic element of logic that maps input to output. Has a *combinator* type to determine cartesian input combinations (short list, long list, and cross list). The *Component* class has two possible *subtypes*:
        - **Assembler**: Interface to a pre-compiled piece of logic, procedurally described in native source code (e.g. the *sum assembler* adds numbers together, the *unit-vector assembler* computes the unit-length vector and original magnitude of another vector, the *beam assembler* computes an FEA beam from nodes and a section profile; see Appendix E for an overview).
        - **Model**: Encapsulates a *Model* object (for the purpose of graph readability and re-use).
    - **Parameter**: Contains the user-provided input values: either numbers or strings (no difference is made between integers or floating point numbers), either single or multiple values (e.g. comma-separated). Abstractions will multiply the given values (or assume 1 if no value is given).
    - **Flow modifier**: Modifies dimensionality of value streams of incoming connections. Contains downstream connections. Can be one of five *subtypes*:
        - **Pass** (default): does nothing (combines multiple connections);
        - **Fold**: converts individual values into an array (multiple arrays for multiple connections);

- **Flatten**: converts an array (or more, for multiple connections) of values into single values;
- **Merge**: converts arrays (for multiple connections) of values into one;
- **Collect**: converts arrays (for multiple connections) of values into single values.

4. An *outlet* contained within a *solvable* object:
    - **Input**: Provides value access into (and out of) the object. Can have a *modifier* property, which modifies value streams of incoming connections (cf. *Modifier* object). Contains *upstream* (towards the back of the directed graph) connections.
    - **Output**: Provides value access out of the object. Receives connections from *downstream* (towards the front of the directed graph) nodes.
5. **Connection**: Connects two *outlet* objects (and thus their *solvables*, i.e. graph nodes) together to allow value flow, forming an edge in the graph.

5.2.2 SOLVING

As mentioned, the creation of or an update to a *Solution* object forms the starting point of a solution job. The *Solution* object controls a number of *solver* object. At the starting of execution of a solution job, the object asynchronously populates its solvers with sampled input parameter values. Then, it directs them to solve for the results of the *solvable* objects.

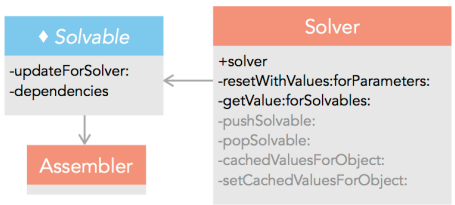


Figure 5.13: The solver and *Solver* interfaces

The solvers contain the main logic through which a PAD graph is solved. They schedule *solvable* objects for updates and serve as storage for the resulting output. In this way, a solver determines the model solution strategy and order. *Solvable* objects are implemented to contain only logic data, and specifically no computed values. The graph is expected to be immutable during the solution process. This architecture is reentrant (see Box 5.5), allowing multiple solvers to share the same instantiated logic model using different input parameters.

> **Reentrancy** signifies that a subroutine can continue after it is interrupted during execution. Often, this is due to independence of global state. Another thread calling the same function constitutes an interruption, necessitating reentrancy for multi-threaded execution.

Box 5.5: 'Reentrancy'

A *solver* class object communicates with *solvable* objects through a common interface, which is implemented differently for each kind of object. The interface consists of an update method and a dependencies getter. Intermediate results are stored on the solver object itself, and injected into the *solvables'* update methods.

THE CONCURRENT STACKLESS SOLVER

The one solver that has been implemented has two important characteristics: it is concurrent, and it is stackless. Concurrency refers to the fact that graph nodes are updated simultaneously whenever possible, making good use of modern multi-core hardware (see Box 5.7). Stacklessness refers to the fact that the logic model is not traversed in one single call stack, but a more complex strategy is used that allows for larger models (or inversely, more efficient memory use; see Box 5.8).

The solver keeps a list with a count for every graph node requiring an update. The list at first only contains the *solvables* that the solver received at initialisation (or reset), typically near the back of the graph. The solver will attempt to solve these, but will typically fail as the results of *upstream solvables* are not yet available. The *upstream solvables* will be *pushed* onto the list, prompting their update attempts (see the first loopback in Figure 5.14).

The solver will continue to attempt to update *solvables* further *upstream* until it encounters one for which the input values are available (typically a *Parameter* object, at the graph front; or in the middle if data from previous solutions has not yet been invalidated). When an update succeeds, the list count for any dependent *solvables* will be decreased (they will be *popped*, see the second loopback), and updates will be prompted whenever a count reaches zero. Box 5.6 gives a simple example.

For concurrency, the solver implementation makes use of both serial and concurrent queue types. The serial queue (the green queue in Figure 5.14) is used for tasks that need to be synchronised, such as *pushing solvables* on the update list (which is not thread-safe, meaning that behaviour is undefined when multiple threads attempt to place objects on the list at the same time). Storing and retrieving cached update results is also done on the serial queue.

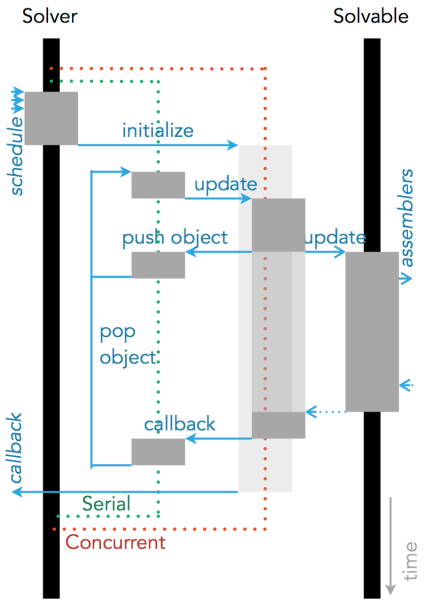
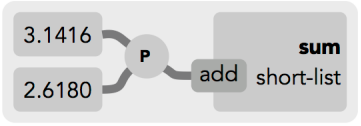


Figure 5.14: A detail of the solver-related sequences from the previous sequence diagram.



i. Solver attempts to update *sum component*
   - Solver attempts to retrieve *pass modifier* values
     - Fails; *component* counter is set to 1

ii. Solver attempts to update *pass modifier*
   - Solver retrieves both *parameter* values
   - Update executes and succeeds
     - *Component* is popped, counter set to 0

iii. Solver attempts to update *sum component*
   - Solver retrieves *modifier* values
   - Update executes and succeeds
     - No dependencies to pop; *callback*

Box 5.6: Example of the solver strategy in operation.

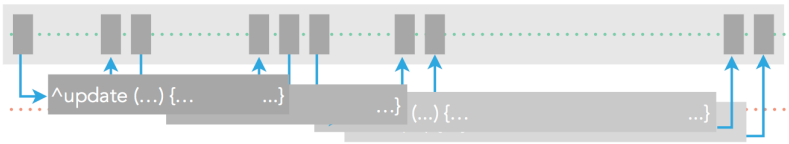
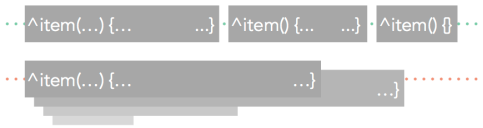


Figure 5.15: Concurrent solver implementation: short-running tasks are serial queue-bound, while long-running tasks are concurrent queue-bound

The concurrent queue (the red queue) is controlled from the serial queue to actually update *solvables* (see also Figure 5.15). These tasks generally run some order of magnitudes longer than the serial tasks, resulting in great gains in solver speed when executed at the same time. When the update is done (successfully or not), the item calls back to the serial queue to store the results and prompt further updates.

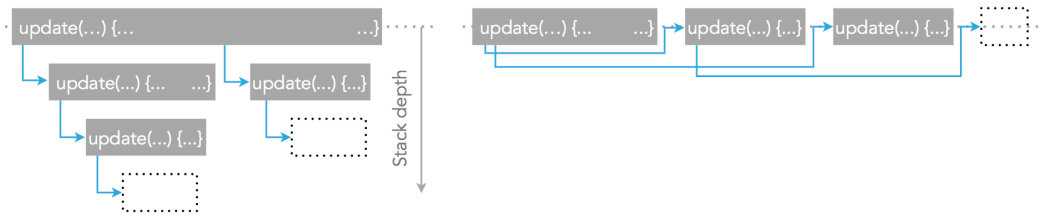
It is important to note that each solver instance will have its own queues, allowing multiple instances to run alongside one another, possibly sharing the same logic model. On top of the individually concurrent solvers, these additional concurrency gains result in very fast and highly scalable parametric and associative solver code.

Concurrency is achieved through the use of so-called queues, which are an abstraction layer over classic multi-threading. Work items can be placed on one of two types of queues, from where they will at some point be executed. Serial type queues execute items one after another, making sure that an item has finished before executing the next. Concurrent type queues execute items simultaneously whenever resources are available (but still on a first-in, first-out basis).



Box 5.7: Serial and concurrent type queues

The call stack is the stack of function contexts that is being created for each subsequent function call. When a function is called, a context is created which holds the local variables and external variable references. When a function is called from within another function, the context for the new function is stacked onto the current context. With extensive function nesting, the call stack will grow higher, until it finally outgrows the memory available. Contrast with a more complex stackless strategy, which does not simultaneously keep multiple levels in memory.



Box 5.8: Stacked and stackless calling

5.2.3 REPRESENTATION AND SERIALISATION

The PAD graph objects, as well as the *Solution* object, need to be communicated to and from the server, as well as saved for persistence and later retrieval. For this, the objects need to be serialised, for which in turn they need to be represented in a serialisable format. The system contains a number of helper classes for loading and saving data from and into system memory.

The serialisable representation forms the translation layer between memory objects and any serial format. It consists of only the information contained within, stripped of any algorithmic logic and implementation details. This information is represented using raw numbers and strings, and their hierarchy using basic collection data structures, such as sequences (arrays) and key-value maps (dictionaries, or map tables).

Two serial formats are supported: XML and JSON (the latter is used before in the text for examples). Both are document-type, typically saved to the file system, as opposed to record-type formats typically found in SQL-like databases. Database persistence falls outside the scope or requirements of this thesis, and therefore remains unimplemented. For the XML format, only loading is implemented, as it is best suited to be created directly by the user as a raw file. The JSON format is fully implemented as it is best-suited for server communication using the REST API (see Section 5.4).

```
<example format="XML">
  <keyword>Full-featured</keyword>
  <keyword>Widely supported</keyword>
</example>
```

XML stands for e**X**tensible **M**arkup **L**anguage (W3C, 2013), a markup language being any system for distinguishably annotating text with meta-level semantics—i.e. annotations that contain meaning about the text they accompany. As such, is is designed to be both human- and machine-readable, very full-featured, and widely supported.

```
{
  "example": "JSON",
  "keywords": [
    "simple",
    "concise"
  ]
}
```

JSON stands for **J**ava**S**cript **O**bject **N**otation (JSON, 2013), and is derived from the JavaScript programming language syntax. As such, valid JSON can be evaluated directly by any JavaScript interpreter, to which it owes much of its popularity. JSON is a simple notation system with limited features, but concise and very easily parsed.

Box 5.9: JSON and XML definition and examples

Full specifications for both file-type formats can be found in the API reference in Appenfix E.

5.3 Values

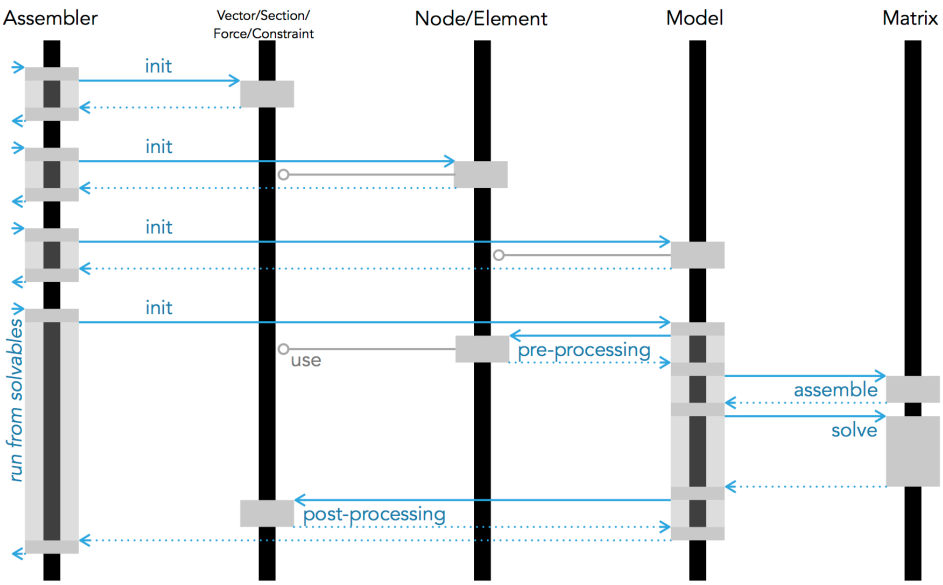


Figure 5.18: A sequence diagram detailing the interactions between different parts of the values subsystem.

The *solving* subsystem results in the generation of *values* inside the compiled functions of *assemblers*. *'Values'* here refers to the objects that are instantiated from the PAD logic, such as numbers, vectors, or FEA elements. An *assembler* (see Section 5.2.1) typically takes *values* of one type as input and of another type as output. For

example, a *sum assembler* may take numbers as input and give the added- and subtracted-together result as output. A *line assembler* may take a series of points as input and give a series of line segments as output.

As the complete PAD graph is traversed, typically ever more complex *values* are created (see the sequence diagram in Figure 5.18). Specifically for the process of creating a structural FEA model (see Box 5.10 for an example), vectors and numbers are used to create forces and constraints, which are used to create nodes, which are used to create elements, which are used to create the structural model. The structural model itself, in turn, makes use of the mathematical matrix and vector libraries to assemble and solve the stiffness matrix given a force vector, after which point the displacements and member forces become available.

> As an example, the assembly of a cantilever beam can be considered, (see Figure 4.11). Disregarding the third dimension, this beam would be rigidly constrained (supported) on one side and free on the other, were it would be loaded with a force. The supported node would be constructed with the rigid constraint and no forces, while the other would be constructed with the force. Subsequently, both nodes would be using in combination with a section profile to construct the element, which would then form the entire model. At this point, the model may be solved, resulting in its displacement vector and member forces.

Box 5.10: Example assembly process for a single-element cantilever beam model.

All *values* are immutable, and two *values* with the same properties are considered equal. This avoids looping dependencies while maintaining downstream integrity. This section describes the logic contained within the various *values* classes, respectively from the mathematical, geometrical and structural categories. All arithmetic and computation has been implemented using 64-bit, double-precision floating point numbers (at a slight loss of performance, though offset by the use of hardware-accelerated *DSP*—Digital Signal Processing—and *MAC*—Multiply-Accumulate—functions). The following sections list the main *values* classes

5.3.1 MATHEMATICAL

Used mainly in the FEA engine described in the previous chapter, the two implemented mathematical *values* classes are the vector and the matrix. A number of helper classes are implemented, such as those representing the main elementary matrix operations (permutation, multiplication, and addition).

> **Dictionary matrix**: the dictionary form (in the case of matrices sometimes referred to as the triplet form) stores the location and value of matrix elements in one-dimensional arrays, and assumes zero elements for any missing locations.
>
> **Envelope matrix**: a modified from the compressed sparse row form, the symmetric compressed sparse row-major envelope form deduplicates symmetric elements, storing an integer index array which describes the envelope structure (the non-zero bandwidths to the diagonal), an envelope array with the off-diagonal matrix elements, and a separate diagonals array for quick access.

Box 5.11: 'Dictionary matrix' and 'envelope matrix'

Both the vector and the matrix are implemented as class clusters, with an abstract class acting only as interface, and a number of specialised variant concrete classes providing actual implementations. For example, the full-fledged arbitrary-length vector class is accompanied by a three-element vector (i.e. 3D vector), which is used far more frequently and allows optimised operations in those cases.

Importantly, both the vector and matrix class provide sparse and symmetric implementations, which are optimised to use the non-zero elements in both storage and in operations. The sparse vector implementation uses the dictionary form, which only uses non-zero elements (see Box 5.11). The sparse matrix implementation is a symmetric compressed sparse row-major envelope matrix (see Box 5.11), which uses only the elements starting from each row's first non-zero element. This type is neither the fastest and most compressed, nor the easiest to implement, but provides a good trade-off between both.

5.3.2 GEOMETRICAL

Extensive geometrical capabilities fall outside the scope of this thesis, but a certain set of classes was implemented. Some of these, such as the point and the line, were implemented out of necessity, as structural classes inherit from these. Others, such as the circle and other simple shapes, were implemented to aid in the assembly of simple structures. The geometrical classes implement basic utility methods, e.g. to find the point on a line closest to an arbitrary point in space, or the point on a line at a certain unit parameter, etc.

5.3.3 STRUCTURAL

Chapter 4 on *Finite Element Analysis* described the structural analysis algorithms implemented as part of this thesis. This section describes their actual object-oriented implementation, which encapsulates the distinct FEA concepts into separate structural *values* classes. In order of assembly (see Figure 5.19), the *values* classes implementing the FEA engine are listed below.

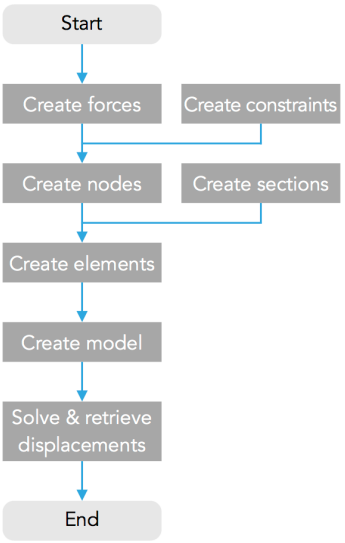


Figure 5.19: A flow chart detailing the process of assembling a structural model.

- The **Force** class inherits from a Vector. It contains exactly six members, one for each global degree of freedom. The two groups of three members are independent, the first three being the vector elements corresponding to the Cartesian force components, and the last three to those of the torque.
- Multiple concrete **Constraint** classes are actually implemented behind an abstract base class interface. Each constraint either constrains a node's displacement or rotation (for simplicity, both will be referred to as 'displacement'). The one exception is a conflated constraint, which represents any arbitrary combination of one of the other constraints, each of which either (see Figure 4.1):
    - fully constrains a node in place;
    - when given a stiffness, constrains a node with a force inversely proportional to its displacement;
    - when given a direction vector, constrains a node to displacement either parallel to a line or normal to a plane;
    - when given both a stiffness and a direction, constraints a node to with a force inversely proportional to its deviation either parallel to a line or normal to a plane.
- The **Node** class inherits from a Point class, having coordinates in Cartesian space. In addition, any number of forces and constraints may be applied to it.
- The **Section** class encompasses all element stiffness logic (currently limited to a double-symmetric, isotropic bar or beam). It may be given any arbitrary stiffness configuration (e.g. axial, minor- and major-flexural, torsional), or its stiffnesses may be retrieved from a library (e.g. a "unit", "HE300B", or "CHS300x12" section).
- The **Element** class forms the core FEM concept (the logic is there to support any element type, but only bars and beams are implemented). It is composed of any number of nodes and a section. For convenience, a force may also be applied to it to represent a distributed load (which is automatically converted to nodal forces).

- The **Model** class is composed of any number of elements, and implements the FEM solution logic (currently limited to linear-elastic analysis only). That is, it handles matrix assembly and pre- and post-processing (e.g. node numbering, constraints application and displacement derivation), although the actual solution is handled by the Matrix class.

5.4 Networking and the REST API

An important part of any distributed software system is the communication between its various components. Over the past few decades, all kinds of technologies have been developed that enable systems to exchange information. On the software side, various communications protocols have been standardised and widely implemented and are readily available as operating system-level libraries (e.g. `urllib` in Python, `cURL` in PHP, `NSURLRequest` in Objective-C/*Cocoa*). For applications' networking functionality, an interface is needed between available libraries and application-specific logic.

As its communications protocol, the StructuralComponents system uses HTTP over TCP. As the architectural style for its interface (its API), it uses the REST style. A brief summary of these terms and their implications is given in Box 5.12; a more detailed description is given in Appendix F.

> **REST** stands for *Representational State Transfer* (Fielding, 2000). It is a style (a *software pattern*) rather than an implementation, and might be implemented on any underlying transfer protocol. REST prescribes manipulations to *resources* through the transfer of representations of the resource, rather than communicating an intention. For example, updating a resource by sending its updated representation, rather than telling the server which properties to update. This philosophy, supported by the so-called *REST constraints*, constitutes a flexible and robust interface.
>
> **HTTP** stands for *HyperText Transfer Protocol* (Berners-Lee, 1991). It was invented for the *WorldWideWeb* project and currently underpins much of the internet. It revolves around client requests identifying certain resources to which servers respond with a status code and the requested resource data. Nominally, the server only responds to requests, and does not itself initiate communication.
>
> **TCP** stands for *Transmission Control Protocol* (DARPA, 1981). Its specifics are not relevant within the scope of this thesis, so it suffices to state that the protocol emphasises reliability and data integrity over transmission speed and latency.

Box 5.12: Explanation of the *REST*, *HTTP*, and *TCP* terms and implications.

5.4.1 REST API AND RESOURCES IMPLEMENTATION

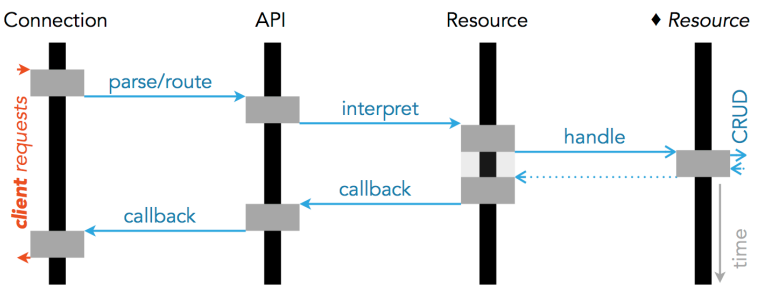


Figure 5.20: A sequence diagram detailing the interactions between different parts of the networking subsystem.

The implemented REST API consists of two main classes: the *API* class handles the main HTTP logic and

configuration, listens for incoming connections, and routes requests to the appropriate resources; the *Resource* class, in turn, wraps around any *model* class that implements the *Resource* interface, handles resource-specific configuration, and translates and forwards a request to its model's pertinent class methods. Figure 5.21 shows a class diagram illustrating the class relations.

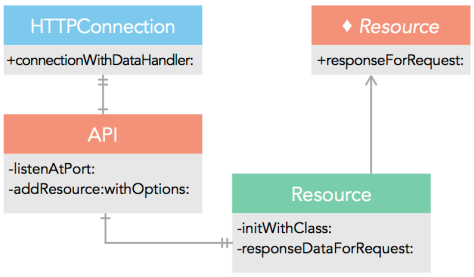


Figure 5.21: A class diagram detailing the networking subsystem.

The *API* class is largely based on a fully functional third-party open source routing HTTP server implementation (Hanson, 2008). The implementation is lightweight and fast, though not necessarily designed for use as a web server on the scale of *Apache* (2013) or *nginx* (NGINX, 2013). As such scale falls outside the scope of this thesis, it is left here as a recommendation that the implementation should prove adequate when used in combination with a load-balancing proxy—in which case the independence of a full-scale web server might well be considered a benefit.

As mentioned, the *Resource* class wraps around a model class—i.e. around one of the parametric-associative logic classes (and the *Solution* class, implementing the *asynchronous job queue* REST pattern; Richardson, 2007). A resource is added to the *API* for each model class, along with configuration options such as which protocol interactions are supported and whether the resource is available as a collection (versus only ever referred to by another resource). Upon an incoming request, the resource forwards a protocol-agnostic translation to its *Resource* class.

The *Resource* class interface allows the model classes to implement the interface access logic themselves. In the general case, this involves retrieving the identified resource object and translating it to a transmittable representation, or translating a sent representation data to a memory object. For the parametric-associative logic classes, these translations make use of the described (de)serialisation functionality. The interface methods also specify the result of a request and handle failure cases (to the extent where it is relevant to the scope of this thesis).

Appendix E shows the full API specification, including accepted data formats and the implemented resource endpoints.

6 Example

To serve as an example of the system proposed in this thesis, this chapter describes how it would be used to approach an actual design problem. The problem revolves around a building which must be realised with a circular floor plan and a certain nett floor area.

6.1 First iteration: building height and dimensions

The first iteration serves to find rough dimensions of the building, and starts with the following information:

- **Problem**: dimensions (height and radius) must be found for the building.
- **Information**: The following requirements and *rule-of-thumb* quantities are relevant:
  - Minimum nett floor area: $200,000\text{m}^2$
  - High-rise space efficieny (Sev, 2009): 70%
  - Leasing depth: 11.0m
  - Floor-to-floor height: 4.0m

The nett floor area of a floor follows simply from the given efficiency ratio. The ratio can be used in the formula $n * \phi * \pi * r^2$, where n is the number of floors, $\phi = 70\%$ is the efficiency ratio, and r is the building radius. This formula contains two unknowns, and thus will likely yield a range of combinations that result in the required floor area. Having to gain insight into these combinations yields an opportunity for an *abstraction*.

The same nett floor area also follows from the given leasing depth. The depth can be used in the formula $n * \pi * [r^2 - (r - \Delta r)^2]$, where $\Delta r = 11\text{m}$ is the leasing depth. This formula also contains two unknowns, and will consequently be used in the *abstraction* as well.

Figure 6.1 shows a PAD graph that models the problem using both formulae; Box 6.1 shows a selection of *API* interactions to create the PAD model; Appendix G gives its full *JSON* representation.

Creating the `pi` *parameter*:

```
$ curl -X POST --data\
'{"type":"parameter","alias":\
"pi","values":[3.14159]}'\
/model/<id>
```

Creating the `mult` input for the `n * pi` *component*:

```
$ curl -X POST --data\
'{"type":"input","name":"mult",\
"modifier":"collect"}'\
/component/<id>
```

Creating the *connection* to the `pi` *parameter*:

```
$ curl -X POST --data\
'{"type":"connection","from":\
"@pi"}' /input/<comp-id>-mult
```

Box 6.1: *API* interactions to create the model, using *cURL* and *JSON*.

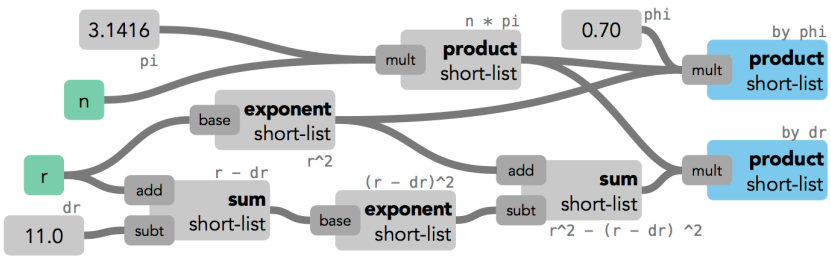


Figure 6.1: The PAD graph for the first iteration. The (green) *parameters* 'n' nad 'r' are designated as *essential parameters*; the right-most (blue) *components* 'by phi' and 'by dr' are designated as *performance metrics*.

In order to run an *abstraction* on the model, the last step is to create a *solution* resource. The *solution* designates *essential parameters* (in this case the *parameters* representing the unknown variables n and r—green in the Figure). For each *essential parameter*, it designates abstraction settings (the sampling mean μ and

variance σ^2, in this case $\mu_n = 60$ and $\sigma_n{}^2 = 20^2$ for n, and $\mu_r = 25.0$ and $\sigma_r{}^2 = 7.0^2$ for r). Finally, it designates *performance metrics* (in this case the *components* 'by phi' and 'by dr' representing the formulae—blue in the Figure).

Creating the *solution*:

```
$ curl -X POST --data '{"type":\
"solution","abstractions":[{\
"parameter":"@n","mean":60,\
"variance":400},{"parameter":\
"@r","mean":25,"variance":49}],\
"metrics":["/comp/<id>",\
"/comp/<id>"]}' /model/<id>
```

Box 6.2: *API* interaction to create the *solution*, using *cURL* and *JSON*.

Box 6.2 shows the *API* interaction to create the *solution* job; Figure 6.2 shows the resulting *abstraction* visualisatons (parallel coordinates and valid/invalid scatter plot). The *abtraction* shows a clear trade-off scenario. At this point, a user of the system might explore the visualisations: first by comparing different values and adjusting domains, then by adjusting the *abstraction* sampling distributions, and finally by tweaking the model and reiterating. For now, it might be assumed that this first iteration results in a choice for $n = 75$ (and thus the total height $H = n * x = 75 * 4 = 300\text{m}$), and $r = 26.0\text{m}$.

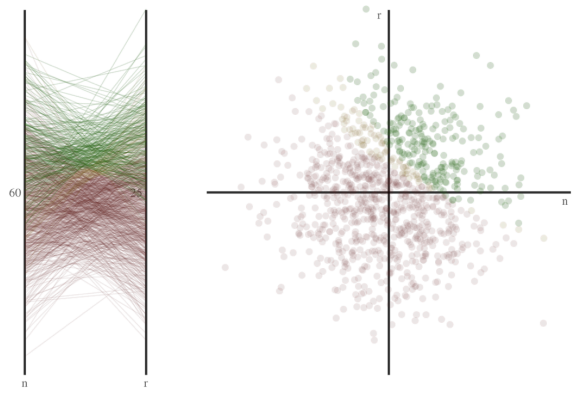


Figure 6.2: The *abstraction* (with 1000 computations) shows a clear trade-off scenario, where floor space can be realised either by height or by width.

7 Discussion

The objectives of this thesis were set in Chapter 1, based on the preceding general and specific problem definitions (respectively concerning the StructuralComponents project as a whole and the opportunities for this thesis). All progress made in this thesis served to advance either of the three specific objectives. This section discusses the degree to which these objectives were achieved, and the remaining limitations.

The specific objectives were guided by the general objective of *"further improving the StructuralComponents approach to the early-stage design process"*. This objective is too general to meaningfully discuss its satisfaction. Accordingly, it will be left to the satisfaction of the above-mentioned specific objectives to give a measure of the success regarding the general objective.

> As noted in Rolvink (2010), "it is not enough, just to develop new and advanced technologies. […] The use of these technologies requires a change in current working methodologies and the engineer needs to innovate in order to use and understand the new technologies." It is unclear that this 'change' has happened, although there indications that the use parametric and associative (and related) tools is becoming more prevalent in the AEC industry.
>
> Rolvink further notes that to gain confidence in new (software) systems, and have the ability to adapt these to the unique requirements of every project, a certain amount of software and programming skills is important. The proposed implementation of this thesis makes use of even lower-level technologies. In cases where the user is exposed to these (e.g. when writing *assemblers*, which involves extending the *C* codebase; or when building custom a client, which involves direct interaction with the REST API), such a skill recommendation arguably becomes even more important.

Box 7.1: As an aside, notes on software design tools in general

7.1 Regarding the increased emphasis on the synthesis phase

The first specific problem concerned the lack of the previous tools' adherence to the formal research on design in engineering, which highlights a significant emphasis on alternatives generation in the early stage of the design process. Accordingly, the first objective proposed to incorporate *"an increased emphasis on the synthesis phase"* (see Section 1.3).

- The proposed abstraction mechanism, i.e. the automatic stochastic variation of selected essential input parameters, each combination of which leads to a solution alternative with selected performance metrics, strives to give the user an awareness of the sensitivity of the metrics to the parameters, the parameter relationships, and the proximity of parameter combinations to the problem constraints.
    - The sensitivity of performance metrics to input parameters indicates in which direction parameter changes will have the most (or least) effect. It helps to find better (or worse) solutions, and can intuitively be perceived at a glance using the wind rose visualisation in the scatter plot, or from the directions of lateral lines in the parallel coordinates plot. Exploring the solution space guided by positive sensitivity most resembles optimisation.
    - The relationship of input parameters indicates which parameter combinations have similar performance according to some metric. It helps to find a parameter's role in the system, reveal opportunities for cheap gains, or indentify multiple non-dominant solutions when searching for an optimum. The interpolation visualisation reveals such parity at a glance.
    - The proximity of solutions to problem constraints indicates in which direction parameter changes come closer to constraints, which is a measure of risk and flexibility in later stages. Clearly highlighting invalid solutions in both the scatter plot and the parallel coordinates plot reveals the

proximity at a glance.

- The parametric and associative design ("PAD") modelling paradigm has already proven to be successful in previous theses. It is used here again to provide the modelling capabilities of the system.

Achievements

The proposed *abstraction* mechanism is built on the parametric and associative design modelling paradigm that previously proved successful. It involves the automatic stochastic variation of selected essential input parameters. Each combination of parameters leads to a solution alternative with selected performance metrics. The aggregate data of the *abstraction* can subsequently be explored, in principle by eye through the various proposed visualisations.

As a whole, the *abstraction* system provides the opportunity for a detailed view and awareness of the design problem at hand. This conclusion follows from the assumed sensibility of user input, from the tested correctness of the analysis results, and from the proven strength of data visualisation. As implemented, the system concerns the *sensitivity of the metrics to the parameters*, the *parameter relationships*, and the *proximity of parameter combinations to the problem constraints*.

Limitations

As mentioned, the *abstraction* system assumes sensibility of user input. To a degree, any design system might have to make that assumption. Correct and sound methods and processes will accept 'bad' input and return 'bad' output (cf. the 'garbage in, garbage out' principle). Still, 'good' input might also lead to 'bad' output, for example when false correlations are shown, or output patterns can be recognised that in reality originate from input patterns. The current implementation gives little measure of the statistical soundness of an *abstraction*.

7.2 Regarding the developed Finite Element Method Analysis engine

The second specific problem concerned the limitations of the *super-element method*-based analysis engine, and the difficulty in extending it to building typologies other than tall structures. Accordingly, the second objective proposed to *"develop a Finite Element Analysis ('FEA') engine"* (see Section 1.3) to serve as a basis for a more versatile building composition system.

- The custom-developed FEA engine provides the previously recommended structural modelling versatility, although it does come at the cost of ease of modelling. It is light-weight and fast, making use of sparse linear solving algorithms. It supports rigid and spring constraints, both nodal and element forces, and bar, bending, and torsion elements, and is extensible with further types.
    - Due to the Finite Element Engine, the versatility objective is arguably met. Most of the type of problems targeted by StructuralComponents ("SC") can be schematised as wireframe models, and beyond that even plate elements can be emulated.
    - The term *versatility* could be contested. It can be argued that an ability of an engineer to schematise a problem as simple rules of thumb makes those rules of thumb *versatile* at a low cost of complexity. As used in the above point, the term refers to the more accurate structural models allowed by an FEA engine. It can express nuances in a schematisation that rules of thumb cannot.
    - From the above points, it becomes apparent that the versatility of the analysis method comes at a cost of ease of modelling. Constructing a complex model, perhaps using bars and beams to emulate plate elements, will likely be a very time-consuming exercise, which goes against the overall goals of SC. Part of this problem can be alleviated by providing built-in shortcuts which model composite building parts in the hardcoded PAD layer, or even as a compiled *assembler*, to an extent removing the need to model individual elements. It could also be argued that more complex problems are simply not what SC is intended for.

- The engine supports wireframe elements (axial bar, flexural beam, and torsional beam), in a fully 3D modelling environment. It further supports both rigid and spring constraints, constraining either or both displacement and rotation, possibly along a direction or on a plane. It supports normal and torque forces, applied either to the nodes themselves or to the elements.
- The developed engine exploits the typical sparsity of structural stiffness matrices, resulting in a light-weight code base and relatively fast performance. At its mathematical core, the engine was built to be extensible and to allow additional plate or volume element types (not implemented). The pre- and post-processing steps can also be extended to allow additional load application and stress analysis methods.

Achievements

The custom-developed FEA engine provides the previously recommended structural modelling versatility. It is light-weight and fast, making use of sparse linear solving algorithms. It supports rigid and spring constraints, both nodal and element forces, and bar, bending, and torsion elements, and is extensible with further types.

Limitations

The increased modelling versatility comes at a the cost of ease of modelling. To an extent, the previously allowed complexity of structural models can still be approximated by built-in composite building parts. Further, ease of modelling forms a user interface problem, to be solved by a client implementation.

As the engine is custom-developed, it is very limited in functionality compared to available software packages, both commercial and open-source. This is purely a result of limited resources, the engine being a one-person development, and not the main subject of this thesis. The choice to custom-develop an engine was primarily made for its educational value. The development itself has yielded many valuable insights in analysis engine and software system architecture and numerical computation.

7.3 Regarding the reimplementation as a client–server software architecture

The third specific problem concerned the restrictions of the plugin-based software architecture of the previous versions of the tool. Accordingly, the third objective proposed the *"reimplementation of the tool [as stand-alone applications, functionally separated] along a client–server divide"*.

- The reimplementation of the tool as a stand-alone client-server software architecture provides the full data structures openness, full transparency, flexibility, and scalability that the previous implementations lacked. The server-side prototype implementation provides the functionality and speed required by the abstraction mechanism, without limiting the capabilities of an (unimplemented) client application.
    - The server component that was implemented for the tool provides a RESTful API, which fully support the open data structures principle of the *Structural Design Tools* strategy. Transparency, inasmuch as it does not stem from access to the source code, is provided by documentation.
    - Access to the source code guarantees full control regarding further development. The source code repository can be forked and fully modified to extend its functionality. Beyond that, the PAD library models provide directly accessible customisation.
    - As implemented, the prototype provides the functionality required by the overall goals of StructuralComponents, and by the abstraction mechanism specifically. It includes the finite analysis engine, as well as a custom-developed parametric and associative solver, with its mathematical, geometrical, and structural libraries, and data analysis methods, in addition to the API component.
    - The client-server architecture ensures that system performance is largely independent of the client device. The implemented prototype already employs parallel processing wherever possible, clearly identifying and supporting opportunities for scaling by spreading out computation over multiple

machines.

- In contrast to the full-featured server prototype, a light client prototype was implemented. Its primary goal is to demonstrate the system in action, as well as serve as an implementation of the data visualisation techniques. However, no effort was spent on basic application features, making the prototype suitable only for highly prepared demonstrations.

Achievements

The reimplementation of the tool as a stand-alone client-server software architecture provides further data structures openness, transparency, flexibility, and scalability than the previous implementations. The server-side prototype implementation provides the functionality and speed required by the abstraction mechanism, without limiting the capabilities of any client application.

Limitations

The client-server architecture is a step towards better performance capabilities. However, such an architecture is complex, and the source code less accessible for users with limited programming knowledge. This limited accessibility might be a major hinderance to the extensibility of the system (in spite of there being far fewer technical limitations).

The system no longer uses the *Rhino* and *Grasshopper* (or *GenerativeComponents*) software packages that Rolvink (2010) and Breider (2008) relied upon. Although this removes a dependency, it also poses a hinderance as these packages are (relatively) widely known and understood, provide easy visualisation and integration with other CAD software, and contain many useful built-in geometry algorithms. It should be noted that an implementation integrated with these packages is possible, but remains unimplemented.

8 Conclusions and recommendations

The previous chapter discussed the initial objectives for this thesis, the degree to which they were achieved, and the limitations that were uncovered in the approach taken. The following section will concisely state the conclusions that were drawn in the process, and the final section will give a number of recommendations for further research and development.

8.1 Conclusions

From the preceding, the following conclusions are drawn (in order of occurence):

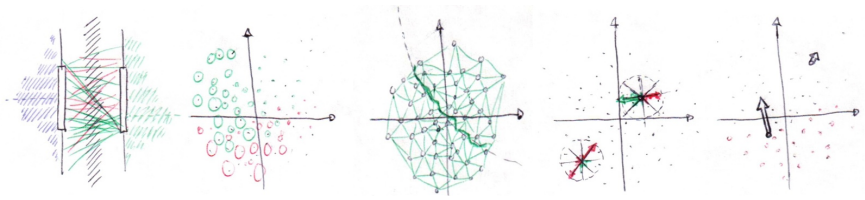


Figure 8.1: The proposed system of abstraction computes variations around user-specified parameter and provides a view into and possible awareness of the specifics of the problem through interactive visualisations.

- The system of *abstraction* achieves the emphasis on the early-stage *synthesis* phases through the automatic generation of design alternatives. The alternatives are generated from parameter variations based on user-defined value ranges. The alternatives are scored by user-defined performance metrics.
- The system takes a *naive* approach of alternatives generation, computing only what the user commands, many times. It does not use any *black-box* method to automatically narrow searches or optimise based on some performance function. Such exploration is left completely to the user.
- Interactive data visualisations allow exploration of the *abstraction* results, ultimately providing insights in such properties as performance *sensitivity*, parameter *relationships*, and constrain *proximity* (see Figure 8.1).
- A full *Finite Element Analysis* engine provides the previously-recommended structural modelling versatility and schematisation accuracy, but at the cost of ease of modelling. Furthermore, as it is custom-written the FEA engine might prove too limited, compared to third party-developed engines.

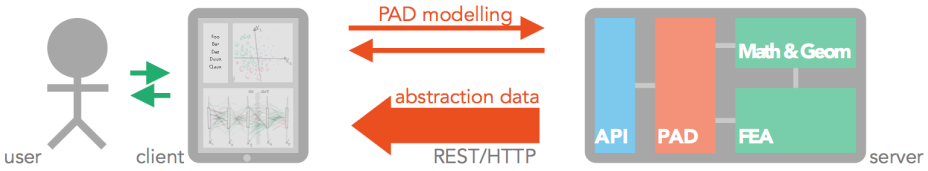


Figure 8.2: The proposed system (and its prototypes) is functionally separated along a client-server software architecture.

- The separation of system functionality along a client-server architecture (see Figure 8.2), yields openness, transparency, flexibility, and scalability by implementing a common and widely-supported interface, and decoupling user interface from analysis logic and computation.

{TODO: 'negatieve puntjes' in discussie?}

- The complexity of the client-server architecture and its current implementation details are beyond the

easy comprehension of a large portion of the intended target audience (the designing structural engineer). This complexity is hidden when using standard functionality, but poses a problem when considering extending the tool.

- The loss of integration with a software package such as *Rhino Grasshopper* (which, it must be noted, is not proposed, but follows from constraints in time and scope) reduces usability, easy visualisation, and integration with CAD software.

8.2 Recommendations

REGARDING THE INCREASED EMPHASIS ON THE SYNTHESIS PHASE

- **Data and statistics**—The techniques proposed for the abstraction system are first steps in a direction. It is recommended that more effort be spent on improvement of these techniques and the development of further functionality:
    - better measures for determining the statistical soundness of an abstraction;
    - automatically adjusting dataset (sample) size based on e.g. statistical soundness or density of solutions;
    - further data-derived properties, in addition to the sensitivity, relationship, and proximity measures.
- **PAD system**—the feature set of the prototype PAD engine implementation is limited, as only functionality used for the demonstration of the tool was developed. It is recommended that effort be spent on improvement and extension:
    - development of many more component assemblers (both geometrical, structural, and composite building parts);
    - further development of the PAD library models feature to serve as a warehouse where users can share and rate models;
    - development of script components using a high-level language like *JavaScript* or *Python*, which would allow easier customisation.

REGARDING THE DEVELOPED FINITE ELEMENT METHOD ANALYSIS ENGINE

- **Functionality**—Many improvements could be made to the prototype FEA engine implementation. Assuming further development of the custom FEA engine (instead of adopting a third-party engine; see point below), it is recommended that effort be spent on:
    - nodal releases, which would allow the user to determine which forces are transferred between adjoining elements, and allow more building types (or safe time on approximation of releases using other features);
    - higher-dimensional or higher-node elements, in addition to the engine's currently only supported extension bar and Euler-Bernoulli/torsion beam wireframe elements (the core computation routines were written in such a way that higher-dimensional or higher-node elements are supported, but none were implemented);
    - concurrency programing or hardware acceleration of floating point computations, which could dramatically improve analysis speed.
- **Third-party solution**—of all technical subsystems, the FEA engine exists among the most mature commercially (or otherwise) available counterparts. While StructuralComponents doesn't currently place

extreme demands on the engine, it seems unlikely that a custom solution would remain a valid option for very long. It is recommended that effort be spent on finding a suitable third-party engine.

REGARDING THE REIMPLEMENTATION AS A CLIENT–SERVER SOFTWARE ARCHITECTURE

- **Client application**—This thesis proposes a client-server application architecture, but only discusses the prototype implementation of a server application in depth. The reason for this focus is that most of the essential components and subsystems are on the server side.

  A small prototype client application was written to demonstrate the essence of the tool. However, it is recommended that for the purpose of further demonstrations and research, more effort be spent on a fully featured client application:

  - core application systems such as document management (i.e. loading and saving);
  - versioning (c.f. *Git* or *Subversion*) to track changes through the iterations of a design;
  - fully three-dimensional model visualisation;
  - further visualisation and interactivity techniques;
  - integration with or exchange methods for common CAD software packages;
  - investigation into connectivity, caching, or client-side execution.

- **Cross-platform**—In order to develop the prototype implementation discussed in this thesis more quickly, a degree of platform dependency was introduced. The core parts of the prototype were written as *C* functions and subroutines. However, for application structure much of it is wrapped in an object-oriented fashion. It is recommended that the platform-dependent parts be rewritten to be platform-agnostic.

  It is important to note that much of the concurrency programming makes use of the open-source *libdispatch* library, which was released near the beginning of this thesis. One major platform implementation is available (for the *Apple OS X* operating system). A number of emerging *Linux* implementations are available, but they remain only partially complete and their robustness unproven. It is recommended that either an adequate cross-platform implementation be found, or another concurrency programming framework be used.

- **Scaling**—The client-server separation enables scaling up of the computationally intensive subsystems independent of client devices. The proposed system is envisioned to be used by multiple users (for example public or company-wide clusters), but the prototype implementation was not developed with such scale in mind. The application runs on a single machine, and includes all separate subsystems. It is recommended to:

  - rewrite the server application such that it can communicate with instances running it on other machines, and share computation jobs as a cluster;
  - refactor parts into separate dedicated applications for some of the subsystem, each running on different machines (suitable subsystem would be those for FEM analysis, PAD solving, and data analysis, as well as HTTP handling and database storage);
  - replace the HTTP protocol with a more complex, but faster streaming protocol (one option would be the UDP protocol; another would be *WebSockets* over TCP).

- **Testing and error handling**—For the prototype implementation, only the FEA engine is covered by unit and integration tests (and only to a certain extent). Proper error handling was almost entirely omitted. For continued development, it is recommended that effort be spent both on further test coverage and proper (avoiding crashes, helpful to users) error handling.

This page is intentionally left blank.

References

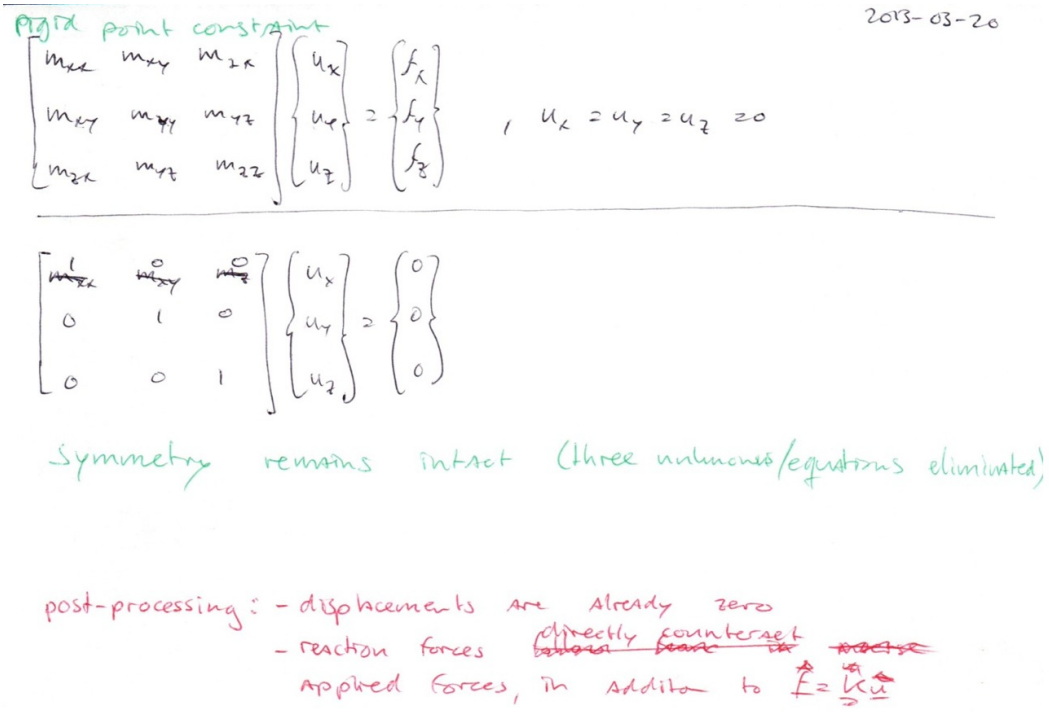
- Aish R., 2005, *"Introduction into GenerativeComponents, a parametric and associative design system"*, Bentley
- Apache Software Foundation, (©) 2013, *"Apache HTTP Server"* (https://httpd.apache.org, retrieved on 6-5-2013)
- Aspettati S. et al., 2001, *"Parametric and feature-based assembly in motorcycle design: from preliminary development to detail definition"*, XIIth ADM-IC
- Bateman, S. et al., 2010, *"Useful Junk? The Effects of Visual Embellishment on Comprehension and Memorability of Charts"*, CHI 2010
- Bentley Systems, Inc., (©) 2013, *"GenerativeComponents"* (http://www.bentley.com/getgc, retrieved on 6-5-2013)
- Bentley, J.L., 1975, *"Multidimensional binary search trees used for associative searching"*, ACM, vol. 18(9), p.p. 509–517
- Berg, M. de et al., 2008, *"Computational Geometry: Algorithms and Applications"*, Springer-Verlag, p. 198
- Berners-Lee, T. et al., 1991, *"HyperText Transfer Protocol"*, version 0.9 (W3C: http://www.w3.org/Protocols/HTTP/AsImplemented.html, retrieved on 4-29-2013)
- Blasio, A. and Bisantz, A., 2002, *"A comparison of the effects of data–ink ratio on performance with dynamic displays in a monitoring task"*, IJIE/Ergon 30
- Breider, J., 2008, "StructuralComponents – development of parametric associative design tools for the structural design of high-rise buildings", *MSc thesis*, Delft University of Technology
- Chapman, W.L. et al., 2001, *"System Design Is an NP-Complete Problem"*, Systems Engineering, 222–228
- Coenders, J.L. and Wagemans, L., 2005, "The next step in modelling for structural design: structural design tools", *Proceedings of the International Association for Shell and Spatial Structures (IASS)*, p.p. 85–92
- Coenders, J.L., 2008, "Parametric and associative strategies for engineering", *IABSE report: Information and Communication Technology (ICT) for Bridges, Buildings and Construction Practice*, vol. 94
- Coenders, J.L., 2011, "NetworkedDesign – next generation infrastructure for computational design", *PhD thesis*, Delft University of Technology
- Cuthill, E., and McKee, J., 1969, *"Reducing the bandwidth of sparse symmetric matrices"*, Proc. 24th Nat. Conf. ACM, p.p. 157—172
- DARPA, 1981, *"Transmission Control Protocol"*, (IETF: https://tools.ietf.org/rfc/rfc793.txt", retrieved on 4-29-2013)
- Davis, D., 2011, *"Datamining Grasshopper"* (http://www.danieldavis.com/datamining-grasshopper/, retrieved on 6-6-2013)
- Davis TA, 1996, *"Direct Methods for Sparse Linear Systems"*, SIAM
- Dini, D., 2009, *"Design, Constraints, and Integrity"*, GDC Europe
- Dorst, K., 1995, *"Comparing paradigms for describing design activity"*, Design Studies, vol. 16 (2)
- Few, S., 2006, *"Information Dashboard Design: The Effective Visual Display of Data"*, O'Reilly Media
- Fielding, R.T., 2000, *"Architectural Styles and the Design of Network-based Software Architectures"*, University of California, Irvine
- Francia, S., 2010, *"The difference between SOAP and REST"*, (SPF 13: http://spf13.com/post/soap-vs-

rest, retrieved on 4-29-2013)

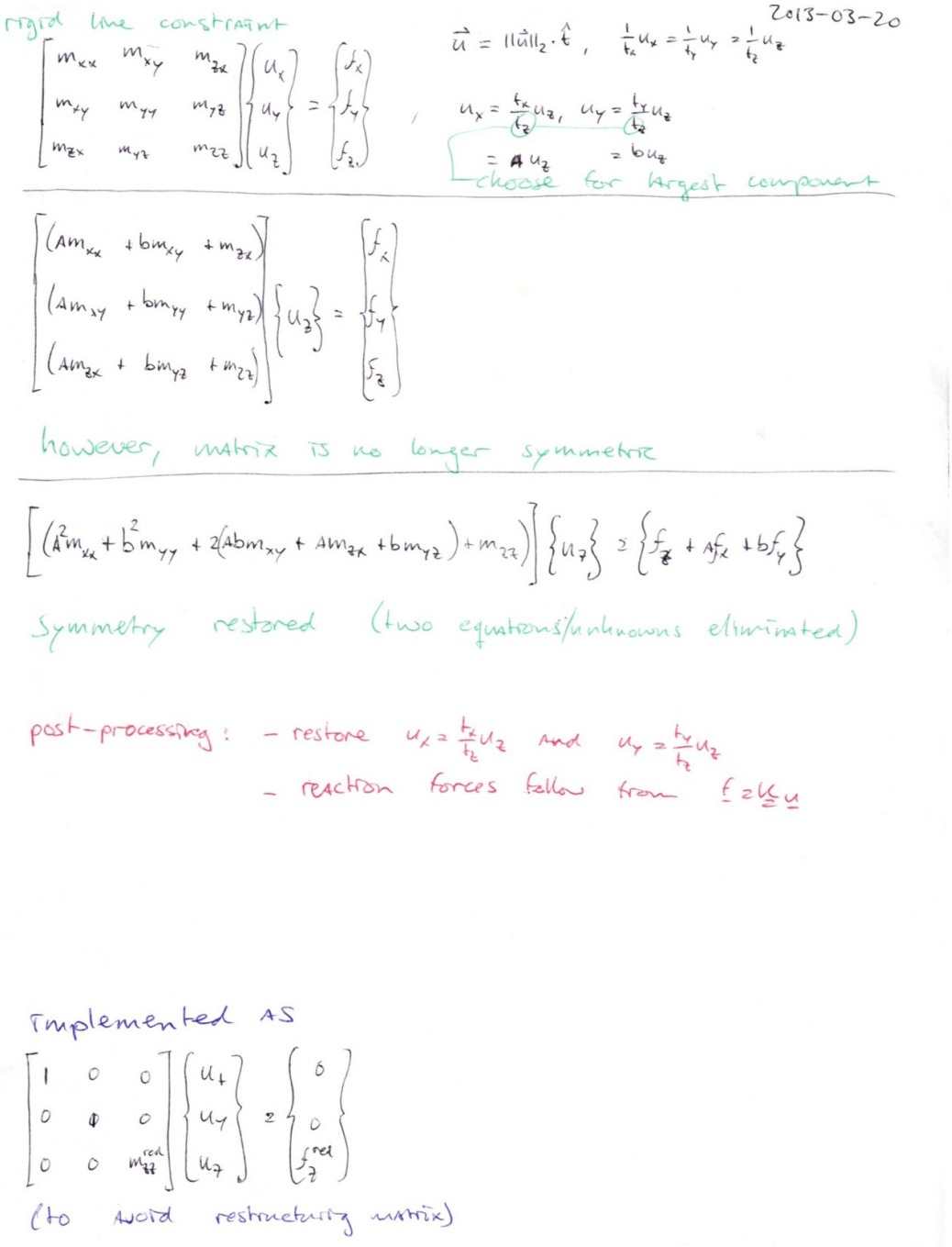
- Fricke, G., 1996, *"Successful Individual Approaches in Engineering Design"*, Research in Engineering Design, vol. 8, pp. 151–165
- George, J.A, and Liu, J.W., 1981, *"Computer Solution of Large Sparse Positive Definite Systems"*, Prentice-Hall
- Hanson, R., 2008, *"CocoaHTTPServer"*, (GitHub: https://github.com/robbiehanson/CocoaHTTPServer, retrieved on 4-29-2013)
- Inselberg, A., 1986, *"The Plane with Parallel Coordinates"*, Visual Computer, vol. 1(4), p.p. 69–91
- JSON, 2013, (http://json.org, retrieved on 6-5-2013)
- Klein, M. et al., 2006, *"The Dynamics of Collaborative Design: Insights From Complex Systems and Negotiation Research"*, Massachussets Institute of Technology
- Ledermann C. et al., 2005, *"Associative parametric CAE methods in the aircraft pre-design"*, AESCTE. vol. 9
- MacLeamy, P., 2004, *"The future of the building industry: the effort curve"* (HOK Network, YouTube: http://www.youtube.com/watch?v=9bUlBYc_Gl4, retrieved on 6-6-2013)
- McNeel, R., and Associates, (©) 2013, *"Rhinoceros"* (http://www.rhino3d.com, retrieved on 6-5-2013) and *"Grasshopper"* (http://www.grasshopper3d.com, retrieved on 6-5-2013)
- Morrison J.P., 2010, *"Flow-Based Programming: A New Approach to Application Development"*, CreateSpace (2; 1994)
- NGINX, Inc., (©) 2013, *"nginx"* (http://nginx.org, retrieved on 6-5-2013)
- Oasys, Ltd., 2013, *"GSA Analysis"* (http://www.oasys-software.com/products/engineering/gsa-analysis.html, retrieved on 6-5-2013)
- Richardson, L., and Ruby, S., 2007, *"RESTful Web Services"*, O'Reilly
- Rolvink, A., 2010, "StructuralComponents – a parametric and associative toolbox for conceptual design of tall building structures", *MSc thesis*, Delft University of Technology
- Sawilowsky, S.S. et al., 2003. *"Statistics via Monte Carlo Simulation with Fortran"*, Rochester Hills, MI: JMASM
- Sev et al., 2009, *"Space efficiency in high-rise office buildings"*, METU, Journal of the Faculty of Architecture
- Simon, H., 1955, *"A Behavioral Model of Rational Choice"*, Quarterly Journal of Economics, vol. 69, pp. 99–188
- Sims-Knight, J.E. et al., 2005, *"A Simulation Task to Assess Students1 Design Process Skill"*, 35th ASEE/IEEE Frontiers in Education Conference
- Smith, R.P. et al., 1998, *"Experimental Observation of Iteration in Engineering Design"*, Research in Engineering Design, vol. 10, pp. 107–117
- Steenbergen, R., 2007, "Super Element in High-Rise Buildings Under Stochastic Wind Load", *PhD thesis*, Delft University of Technology
- Tufte, E., 1983, *"The Visual Display of Quantitative Information"*, Graphical Press
- Victor, B., 2011, _"Up and Down the Ladder of Abstraction", (Worrydream: http://worrydream.com/LadderOfAbstraction/, retrieved on 4-29-2013)
- W3C, 2013, (http://www.w3.org/XML, retrieved on 6-5-2013)
- W3C, 2000, *"Simple Object Access Protocol"*, version 1.1 (W3C: http://www.w3.org/TR/soap/, retrieved on 4-29-2013)
- Wigderson, A., 2009, *"Knowledge, Creativity and P versus NP"*, IAS Princeton

A Derivation of constraint elementary matrix operations

A.1 Point constraint



A.2 Line constraint



A.3 Plane constraint

REGULAR

2013-03-19

rigid plane constraint (A)

$$\begin{bmatrix} m_{xx} & m_{xy} & m_{zx} \\ m_{xy} & m_{yy} & m_{yz} \\ m_{zx} & m_{yz} & m_{zz} \end{bmatrix} \begin{Bmatrix} u_x \\ u_y \\ u_z \end{Bmatrix} = \begin{Bmatrix} f_x \\ f_y \\ f_z \end{Bmatrix}$$

$\vec{u} \cdot \hat{n} = 0$

$u_x n_x + u_y n_y + u_z n_z = 0$

$u_x = -\frac{n_y}{n_x} u_y - \frac{n_z}{n_x} u_z$

$u_{B_{dx}} = A u_y + b u_z$, $A = -\frac{n_y}{n_x}$, $b = -\frac{n_z}{n_x}$

choose for largest component

$$\begin{bmatrix} (A m_{xx} + m_{xy}) & (b m_{xx} + m_{zx}) \\ (A m_{xy} + m_{yy}) & (b m_{xy} + m_{yz}) \\ (A m_{zx} + m_{yz}) & (b m_{zx} + m_{zz}) \end{bmatrix} \begin{Bmatrix} u_y \\ u_z \end{Bmatrix} = \begin{Bmatrix} f_x \\ f_y \\ f_z \end{Bmatrix}$$

however, matrix is no longer symmetric

$$\begin{bmatrix} (A^2 m_{xx} + 2A m_{xy} + m_{zz}) & (Ab m_{xx} + A m_{zx} + b m_{xy} + m_{yz}) \\ (Ab m_{xx} + A m_{zx} + b m_{xy} + m_{yz}) & (b^2 m_{xx} + 2b m_{zx} + m_{zz}) \end{bmatrix} \begin{Bmatrix} u_y \\ u_z \end{Bmatrix} = \begin{Bmatrix} f_y + A f_x \\ f_z + b f_x \end{Bmatrix}$$

symmetry restored (one equation/unknown eliminated) qed

post-processing: - restore $u_x = -\frac{n_y}{n_x} u_y - \frac{n_z}{n_x} u_z$

- reaction forces follow from $\underline{f} = \underline{\underline{K}}\,\underline{u}$

Implemented as

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & m_{yy}^{red} & m_{yz}^{red} \\ 0 & m_{yz}^{red} & m_{zz}^{red} \end{bmatrix} \begin{Bmatrix} u_x \\ u_y \\ u_z \end{Bmatrix} = \begin{Bmatrix} 0 \\ f_y^{red} \\ f_z^{red} \end{Bmatrix}$$

(to avoid restructuring matrix)

ALTERNATIVE

Rigid plane constraint (B) 2013-03-19

$$\begin{bmatrix} m_{xx} & m_{xy} & m_{zx} \\ m_{xy} & m_{yy} & m_{yz} \\ m_{zx} & m_{yz} & m_{zz} \end{bmatrix} \begin{Bmatrix} u_x \\ u_y \\ u_z \end{Bmatrix} = \begin{Bmatrix} f_x \\ f_y \\ f_z \end{Bmatrix}, \quad u_x = a u_y + b u_z \text{ (plane)}, \quad \vec{u} \cdot \vec{f} = 0 \text{ (force orthogonal to plane)}$$

$$\begin{bmatrix} (a m_{xx} + m_{xy}) & (b m_{xx} + m_{zx}) \\ (a m_{xy} + m_{yy}) & (b m_{xy} + m_{yz}) \\ (a m_{zx} + m_{yz}) & (b m_{zx} + m_{zz}) \end{bmatrix} \begin{Bmatrix} u_y \\ u_z \end{Bmatrix} = \begin{Bmatrix} f_x \\ f_y \\ f_z \end{Bmatrix}$$

however, matrix is no longer symmetric

$$u_x f_x + u_y f_y + u_z f_z = 0$$

$$u_x f_x = -u_y f_y - u_z f_z$$

$$(a u_y + b u_z)\left[(a m_{xx} + m_{xy}) u_y + (b m_{xx} + m_{zx}) u_z\right] + u_y\left[(a m_{xy} + m_{yy}) u_y + (b m_{xy} + m_{yz}) u_z\right] + u_z\left[(a m_{zx} + m_{yz}) u_y + (b m_{zx} + m_{zz}) u_z\right] = 0$$

$$\left[a^2 m_{xx} + 2a m_{xy} + m_{yy}\right] u_y + \left[a b m_{xx} + a m_{zx} + b m_{xy} + m_{yz}\right] u_z \quad \text{(equal} \rightarrow \text{symmetry?)}$$

$$+ \left[a b m_{xx} + a m_{zx} + b m_{xy} + m_{yz}\right] u_y + \left[b^2 m_{xx} + 2 b m_{zx} + m_{zz}\right] u_z = 0$$

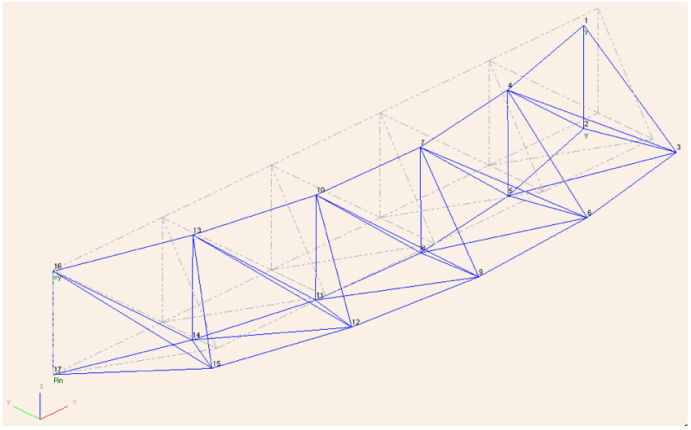
$$(f_y + a f_x) u_y + (f_z + b f_x) u_z = 0$$

$$\begin{bmatrix} (a^2 m_{xx} + 2a m_{xy} + m_{yy}) & (a b m_{xx} + a m_{zx} + b m_{xy} + m_{yz}) \\ (a b m_{xx} + a m_{zx} + b m_{xy} + m_{yz}) & (b^2 m_{xx} + 2 b m_{zx} + m_{zz}) \end{bmatrix} \begin{Bmatrix} u_y \\ u_z \end{Bmatrix} = \begin{Bmatrix} f_y + a f_x \\ f_z + b f_x \end{Bmatrix}$$

qed

B FEA validation

B.1 Truss

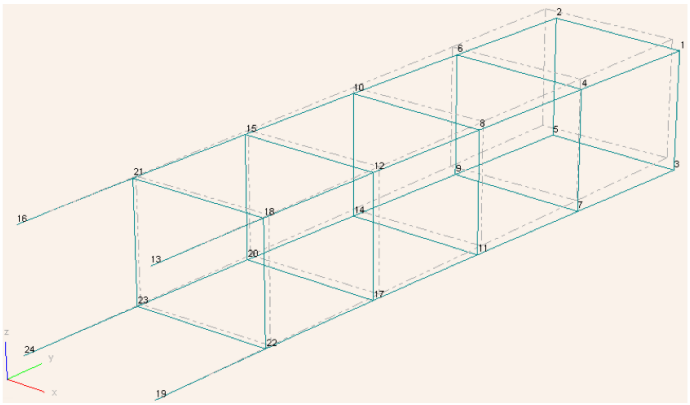


Node	Case	Ux [m]	Uy [m]	Uz [m]	|U| [m]	Rxx [rad]	Ryy [rad]	Rzz [rad]	|R| [rad]	Uxy [m]
1	A1	-52,41	0,0	-39,05	65,36	0,0	0,0	0,0	0,0	52,41
2	A1	-52,41	0,0	-33,92	62,43	0,0	0,0	0,0	0,0	52,41
3	A1	26,36	-61,20	-33,10	74,40	0,0	0,0	0,0	0,0	66,63
4	A1	-47,29	-115,0	-31,41	128,2	0,0	0,0	0,0	0,0	124,3
5	A1	-47,29	-118,0	-34,05	131,6	0,0	0,0	0,0	0,0	127,1
6	A1	5,483	-153,7	-26,31	156,1	0,0	0,0	0,0	0,0	153,8
7	A1	-34,18	-181,6	-23,55	186,3	0,0	0,0	0,0	0,0	184,8
8	A1	-34,18	-183,6	-26,19	188,5	0,0	0,0	0,0	0,0	186,7
9	A1	-26,21	-188,0	-19,53	190,8	0,0	0,0	0,0	0,0	189,8
10	A1	-18,24	-183,6	-12,86	184,9	0,0	0,0	0,0	0,0	184,5
11	A1	-18,24	-181,6	-15,50	183,2	0,0	0,0	0,0	0,0	182,5
12	A1	-57,90	-153,7	-12,74	164,8	0,0	0,0	0,0	0,0	164,3
13	A1	-5,126	-118,0	-5,001	118,2	0,0	0,0	0,0	0,0	118,1
14	A1	-5,126	-116,0	-7,641	116,4	0,0	0,0	0,0	0,0	116,1
15	A1	-78,77	-61,20	-5,956	99,93	0,0	0,0	0,0	0,0	99,75
16	A1	0,0	0,0	-5,126	5,126	0,0	0,0	0,0	0,0	0,0
17	A1	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0

i	x	y	z	dx	dy	dz
1	2.5	1	0.5	-52.4136	0	-39.0511
2	2.5	1	-0.5	-52.4136	0	-33.925
3	2	0	0	26.3592	-61.1987	-33.0956
4	1.5	1	0.5	-47.2875	-114.99	-31.4104
5	1.5	1	-0.5	-47.2875	-118.017	-34.0497
6	1	0	0	5.48278	-153.746	-26.3106
7	0.5	1	0.5	-34.1765	-181.607	-23.5502
8	0.5	1	-0.5	-34.1765	-183.559	-26.1894
9	0	0	0	-26.2068	-187.952	-19.5256
10	-0.5	1	0.5	-18.2371	-183.559	-12.8617
11	-0.5	1	-0.5	-18.2371	-181.607	-15.5009
12	-1	0	0	-57.8964	-153.746	-12.7406
13	-1.5	1	0.5	-5.12612	-118.017	-5.00145
14	-1.5	1	-0.5	-5.12612	-114.99	-7.6407
15	-2	0	0	-78.7728	-61.1987	-5.95556
16	-2.5	1	0.5	0	0	-5.12612
17	-2.5	1	-0.5	0	0	0

Table B.1: FEA engine nodal displacement results

B.2 Frame



Node	Case	Ux [m]	Uy [m]	Uz [m]	|U| [m]	Rxx [rad]	Ryy [rad]	Rzz [rad]	|R| [rad]	Uxy [m]
1	A1	33,42	-32,49	-0,02439	46,61	-0,02466	0,0	-9,979	9,979	46,61
2	A1	33,47	-22,51	-0,02439	40,33	-0,02466	0,0	-10,03	10,03	40,33
3	A1	33,42	-32,49	0,02439	46,61	0,02466	0,0	-9,979	9,979	46,61
4	A1	23,55	-30,79	-0,008617	38,76	-0,01503	0,0	-9,592	9,592	38,76
5	A1	33,47	-22,51	0,02439	40,33	0,02466	0,0	-10,03	10,03	40,33
6	A1	23,57	-21,21	-0,008617	31,71	-0,01503	0,0	-9,622	9,622	31,71
7	A1	23,55	-30,79	0,008617	38,76	0,01503	0,0	-9,592	9,592	38,76
8	A1	14,46	-26,69	-0,001848	30,35	-0,009517	0,0	-8,397	8,397	30,35
9	A1	23,57	-21,21	0,008617	31,71	0,01503	0,0	-9,622	9,622	31,71
10	A1	14,47	-18,31	-0,001848	23,34	-0,009517	0,0	-8,416	8,416	23,34
11	A1	14,46	-26,69	0,001848	30,35	0,009517	0,0	-8,397	8,397	30,35
12	A1	6,977	-20,19	890,1E-6	21,36	-0,007577	0,0	-6,399	6,399	21,36
14	A1	14,47	-18,31	0,001848	23,34	0,009517	0,0	-8,416	8,416	23,34
15	A1	6,976	-13,81	890,1E-6	15,47	-0,007577	0,0	-6,414	6,414	15,47
17	A1	6,977	-20,19	-890,1E-6	21,36	0,007577	0,0	-6,399	6,399	21,36
18	A1	1,889	-11,29	0,002159	11,44	-0,006284	0,0	-3,600	3,600	11,44
20	A1	6,976	-13,81	-890,1E-6	15,47	0,007577	0,0	-6,414	6,414	15,47
21	A1	1,884	-7,713	0,002159	7,940	-0,006284	0,0	-3,613	3,613	7,940
22	A1	1,889	-11,29	-0,002159	11,44	0,006284	0,0	-3,600	3,600	11,44
23	A1	1,884	-7,713	-0,002159	7,940	0,006284	0,0	-3,613	3,613	7,940

i	x	y	z	dx	dy	dz	dxx	dyy	dzz
1	0.5	5	0.5	33.4203	-32.4931	-0.0243926	-0.0246555	-1.33227e-14	-9.97878
2	-0.5	5	0.5	33.4691	-22.5069	-0.0243926	-0.0246555	-8.88178e-15	-10.0281
3	0.5	5	-0.5	33.4203	-32.4931	0.0243926	0.0246555	-2.62013e-14	-9.97878
4	0.5	4	0.5	23.5478	-30.7869	-0.0086174	-0.0150258	-3.19744e-14	-9.59154
5	-0.5	5	-0.5	33.4691	-22.5069	0.0243926	0.0246555	-1.11022e-15	-10.0281
6	-0.5	4	0.5	23.565	-21.2131	-0.0086174	-0.0150258	-7.10543e-15	-9.62159
7	0.5	4	-0.5	23.5478	-30.7869	0.0086174	0.0150258	-2.95319e-14	-9.59154
8	0.5	3	0.5	14.4646	-26.6867	-0.00184755	-0.00951722	-4.04121e-14	-8.39715
9	-0.5	4	-0.5	23.565	-21.2131	0.0086174	0.0150258	-1.22125e-14	-9.62159
10	-0.5	3	0.5	14.4683	-18.3133	-0.00184755	-0.00951722	-8.88178e-16	-8.41618
11	0.5	3	-0.5	14.4646	-26.6867	0.00184755	0.00951722	-3.04201e-14	-8.39715
12	0.5	2	0.5	6.97734	-20.1867	0.000890126	-0.00757732	-1.19904e-14	-6.39908
13	0.5	0	0.5	0	0	0	0	0	0
14	-0.5	3	-0.5	14.4683	-18.3133	0.00184755	0.00951722	-2.22045e-15	-8.41618
15	-0.5	2	0.5	6.97556	-13.8133	0.000890126	-0.00757732	-3.55271e-15	-6.41424
16	-0.5	0	0.5	0	0	0	0	0	0
17	0.5	2	-0.5	6.97734	-20.1867	-0.000890126	0.00757732	-1.4766e-14	-6.39908
18	0.5	1	0.5	1.88872	-11.2871	0.00215944	-0.00628379	-3.77476e-15	-3.60017
19	0.5	0	-0.5	0	0	0	0	0	0
20	-0.5	2	-0.5	6.97556	-13.8133	-0.000890126	0.00757732	-4.66294e-15	-6.41424
21	-0.5	1	0.5	1.8844	-7.71292	0.00215944	-0.00628379	-1.33227e-15	-3.61274
22	0.5	1	-0.5	1.88872	-11.2871	-0.00215944	0.00628379	-6.71685e-15	-3.60017

23	-0.5	1	-0.5	1.8844	-7.71292	-0.00215944	0.00628379	-2.60902e-15	-3.61274
24	-0.5	0	-0.5	0	0	0	0	0	0

Table B.2: FEA engine nodal displacement results

C Software concepts and patterns

The software framework behind this thesis is primarily written using the object-oriented software paradigm, where program logic is encapsulated within distinct objects. This logic consists of program data and the algorithms that manipulate it. Objects retain references to their respective portions of the data, and implement the algorithms in associated methods.

Importantly, objects declare the ways in which its encapsulated logic can be accessed externally, and no assumptions need to be made about how anything is handled internally. This yields modular code, where an underlying implementation can be completely replaced by another (presumably better-performing) one, without the necessity to modify any external code.

Objects can be passed around to carry their encapsulated data and logic over to anywhere in the program, without ever requiring that assumptions are made as to how its declared methods are implemented, but only promising that they are. As such, different objects interact with one another on one level of complexity and combine to form the internal components of more complex objects, and ultimately the complex behaviour of a program.

An object is generally one of multiple members of a class. Classes are the collection of program code that implements the logic and determines the behaviour of the instantiated objects. Classes themselves behave like objects to the extent that they are similarly passed around and implement class methods which manipulate class-level data. Classes are related in a tree structure where lower classes extend the common behaviour of higher classes to perform a specialised role.

The object-oriented paradigm is a collection of software patterns—recurring manners of implementing program logic. Described above, examples of patterns are data and method referencing, encapsulation, and class inheritance. Far more patterns have been employed to implement the framework, the most important of which are described below.

INTERFACES

Implicitly or explicitly, objects of different classes can implement common methods. This fact makes it possible to use different objects to perform the same role. Implementing a set of expected methods implicitly is known as *duck-typing*, named after the phrase *"if it walks like a duck, and it quacks like a duck, then it's a duck"*. In software, the concept is that if any given object behaviour is expected, and that expectation is met, the object fulfils the role.

In order to make the implementation explicit, an object can be described as adhering an interface. An interface is like a class in that it describes behaviour, but does not have instantiations of itself. The primary benefit of explicitly adhering to an interface is that classes can be automatic type-checked, reducing the chance of human error.

EXTENSION

Class extension allows additional class behaviour to be defined at any arbitrary location in the program code. This is useful primarily for extending the behaviour of framework objects, which cannot be modified themselves. But it is also useful for grouping common behaviour of different complex objects together in one place—such as when making various classes adhere to the same set of behaviours. This grouping can provide benefits (primarily for the programmer) like avoiding code repetition and quick look-up and comparison.

SINGLETONS

Singleton classes give wide access to a single instance object shared by different parts of program logic. Such

classes can be employed whenever the object-oriented concept of a class with multiple instances is not a natural fit. These cases occur where the logic revolves around some central resource, such as a central register that keeps track of multiple objects. Only a single such object is required, and access to it must be guaranteed to ultimately access the same shared resource.

MODEL–VIEW–CONTROLLER

The model–view–controller pattern prescribes class categories, their typically associated roles and behaviours, and the boundaries and relationships between them. These prescriptions can be employed in designing the classes that form a program component. Model classes contain the primary program data, and typically provide access to the methods that manipulate it. View classes form the infrastructure through which the data enters and leaves the program. Controller classes form the link between model and view classes, and determine how events in one translate to modifications in the other.

DELEGATION

Objects can delegate part of the actual implementation of some of their functionality to other objects. This pattern usually occurs in complex objects that consist of several component objects. The complex parent object employs a reusable component object to act as wrapper around some functionality, but implements the core of that functionality itself. For example, a controller can employ a button view to listen for the user event of pressing the button, but itself implement what happens when the event occurs.

PLUGIN

A plugin is a software module that runs within a another software package, which provides a plugin programming interface for the purpose of allowing extending its own base functionality.

D Use case methodology (#d)

NB. This text is taken from Rolvink (2010).

Use cases are often performed during software development in order to be sure that the software does what it is required to do. A use case describes the interaction between one or more (external) actors and the system under consideration, represented as a sequence of simple steps, in order to achieve a certain goal. An actor in this case is someone who exists outside the system under study, whereby the system is treated as black-box ("Use case", Wikipedia: The Free Encyclopedia).

The primary goals of the use cases given in this chapter are to present different basic examples of how StructuralComponents functions and to determine various scenarios which are likely to happen. The use cases capture who does what with the system, for what purpose, without dealing with the internal design of the system.

According to use case terminology, it has been avoided to implement user interfaces into the use case under consideration. The next sections therefore only show UML diagrams of possible workflows within StructuralComponents.

The use cases described in this chapter all follow the template given in Table D.1.

1. Use case name	Use case identifier, written in verb-noun format, and should be clear enough to understand what the use case is about.
2. Description	Short description of the use case and goals to be achieved by the use case.
3. Actors	Description of the actors who interact with the system under consideration and are involved in this use case.
4. Assumptions	List of all the assumptions necessary that must be true for the use case to terminate successfully.
5. Course of events	Steps between actor and goals that are necessary to achieve the goals given in the description.
6. Alternatives (optional)	Any variations in the course of events in the use case.
7. Issues (optional)	List of issues that remain to be solved.

Table D.1: Template for use case design.

E Parametric and associative API

Data formats

All objects can be parsed from either XML- or JSON-formatted data, encoded using the UTF-8 unicode character set. Response data from the REST API will always be JSON-formatted.

In the case of XML data, object types are inferred from element names. JSON data is expected to contain a root-level key-value pair with key `type` and string value analogous to the XML element name. As a general rule, untyped data will be invalid and remain unparsed.

All objects are referred to using a Unique Resource Identifier (URI), designated by the XML attribute name or JSON key-value pair key `uri`. Objects may additionally be given a human-readable alias, designated by the attribute or key `alias`. The alias only carries semantic machine meaning in special cases.

Throughout this document parentheses (`(...)`, either with the ellipsis or containing text) will act as placeholders. The expected format and type of the actual data can be deduced from the text between the parentheses (such as "any string" or a regular expression) or from the accompanying documentation text.

DEFINITION

The root-level definition object represents an iteration in the design process, encapsulating a parametric and associative model of logic. It contains one ore more models. Multiple models will be semantically related, for example as alternative solutions in an iteration in the design process. The definition further contains typical document metadata such as date stamps, author names, etc.

XML:

```
<definition
  alias="(any string)"
  meta-(key)="(any meta value)">
  (...)
</definition>
```

JSON:

{ "type": "definition" "alias": "(any string)" "meta-(key)": (any meta value) "models": [(…)] }

- Contents (XML) / **`models`** (JSON): (...)

  The system expects content elements (XML) or `models` -array objects (JSON) of one type: `model`

- **`meta-(key)`** : (any meta value)

  Definition can carry arbitrary values referred to by `meta-(key)` keys. These can be used to attach data such as date stamps or author names to the definition. The values can be of any representable type, such as a number, a string, an array, an object, or `null`. The meta values do not carry semantic machine meaning.

MODEL

The model object represents a design solution. In simple terms, it contains the individual pieces of logic that together form the solution. The objects themselves can recursively contain additional models, each with its own

pieces of logic. As such, the model object forms one level of logic in a bigger parametric and associative model.

XML:

```
<model
  alias="(any string)">
  (...)
</model>
```

JSON:

```
{
  "type": "model",
  "alias": "(any string)",
  "objects": [(...)]
}
```

- Contents (XML) / **`outlets`** (JSON): (...)

  The system expects content elements (XML) or `objects`-array objects (JSON) of three types: `component`, `parameter`, `modifier`

COMPONENT

The component, the namesake of this series of theses, is the atomic element of logic in the parametric and associative model. It maps input values to output values in the way a simple arithmetic addition operation maps input numbers to an output sum. Various components can be strung together to create more complex mappings in the way the addition operation can be combined with a multiplication. Multiple component types exist, each described below.

Components contain a combinator object which prepares the input values for mapping. All components take arrays as input, and perform the actual mapping on element pairs. Combinators are responsible for choosing the element pairs. The short-list and long-list combinators pair each respective element pair together, up to the end of the shortest array or the longest array, respectively. The cross-list combinator pairs creates pairs of each possible combination of elements (the resulting set of pairs is knowns as the Cartesian product).

Assembler component

The assembler component acts as the interface to a pre-compiled piece of logic, contained within an assembler object or function. These pieces of logic are encoded into the system through native programming code. *{APX X}* lists the full list of available assemblers, their required (and optional) input, and their available output.

Model component

The model component acts as the interface to a another model object contained within. Its inputs and output can be connected to both from the objects in the same model as the component, and from the objects in the model contained within the component. The model component encapsulates complex logic into a single component, for the purpose of readability and re-use.

XML:

```
<component
  type="(assembler|model)"
  combinator="(short|long|cross)"
  assembler="(any assembler name string)"
  alias="(any string)">
  (...)
</model>
```

JSON:

```
{
  "type": "component",
  "component-type": "(assembler|model)",
  "combinator": "(short|long|cross)",
  "assembler": "(any assembler name string)",
  "alias": "(any string)",
  "model": (any model object),
  "outlets": [(...)]
}
```

- Contents (XML) / **`outlets`** (JSON): (...)

  The system expects content elements (XML) or `outlets`-array objects (JSON) of four types: `input`, `output`, `assembler` (for assembler-type component and XML only), `model` (for model-type component and XML only)

- **`type`** (XML) / **`component-type`** (JSON): "(assemblerlmodel)"

  Component objects have a subtype in addition to their `component` type, designated in XML by a `type` attribute and in JSON by a `component-type` pair. In both cases, the value must be either `assembler` or `model`. If omitted, the subtype will be inferred from the presence of an assembler or model pair or attribute (or content element).

- **`combinator`**: "(shortllonglcross)"

  Components have a combinator object, designated by a `combinator` attribute or pair with one of the string values `short`, `long`, or `cross` (for a long-list, short-list, or cross-list combinator, respectively). If omitted, the `long` value will be assumed as default.

- **`assembler`** (for assembler-type component only): (...)

  Assembler-type components can contain an assembler, represented as an `assembler` attribute or pair, or for XML as an `assembler` element with name `attribute`, with string value from the assembler names listed in *{APX X}*.

- **`model`** (for model-type component and JSON only): (...)

  Model-type components can contain a model, represented as described for the `model` type above, or by URI. For XML, the model object is expected as content of the component element.

PARAMETER

Parameters contain the user's input in the form of numbers, strings, or some complex types. Parameters have abstraction settings (although these are separated out into **Solution** objects). These settings determine how the system will generate variations of the contained values in order to abstractions (e.g. the settings might define a probability function from which variations are stochastically sampled).

XML:

```
<parameter
  type="value"
  alias="(any string)">
  (...)
</model>
```

JSON:

```
{
  "type": "parameter",
  "parameter-type": "value",
  "alias": "(any string)",
  "values": [(...)]
}
```

- Contents (XML) / `values` (JSON): (...)

  Comma-separated numbers (XML) or array of numbers (JSON), or a series element (XML) or object (JSON).

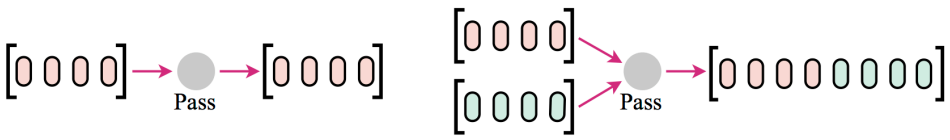
- `type` (XML) / `parameter-type` (JSON): "value"

  Currently only `value` -type parameters are supported. Can be omitted.

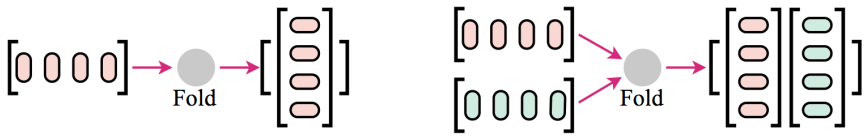
FLOW MODIFIER {#E-1E {#E-1F}}

Flow modifiers ('modifiers'), which can be of the type 'pass', 'fold', 'flatten', or 'merge'. Modifiers manipulate the dimensionality of the value flow depending on the type (note that the order of connections matters):

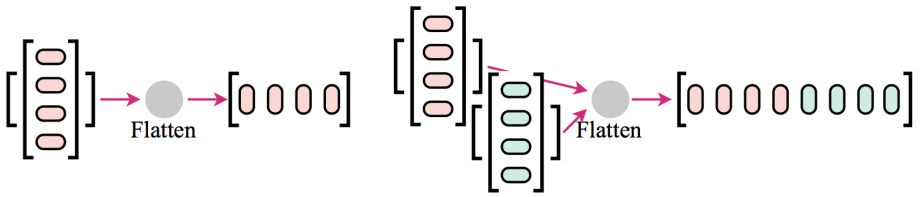
- **Pass** modifiers let the value flow pass through without manipulation, but combine multiple connections into one.



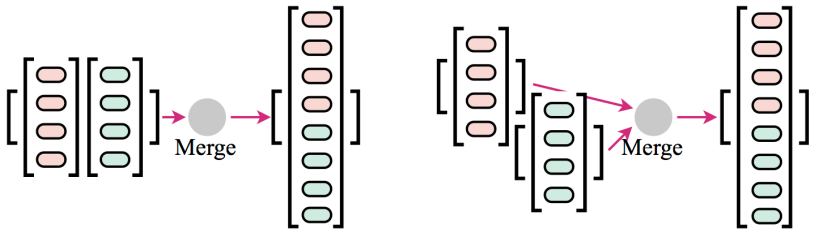
- **Fold** modifiers take individual values form the flow and return them in an array, and in the case of multiple connections will return multiple arrays.



- **Flatten** modifiers take arrays from the flow and return the combined individual array elements, combining the elements of multiple connections together.



- **Merge** modifiers take arrays from the flow and return the merged arrays, again combining the elements of multiple connections together.



- **Collect** modifiers conveniently combine the **pass** and **fold** modifiers in order to cover the common case of collecting separate flat flows into a single collection flow.

It should be noted that several combinations of value dimensionalities exist that do not logically fit a functional operation. For example, a flow of individual (no-dimensional) values cannot be further flattened. An flow of array (one-dimensional) values could theoretically be further folded (to become two-dimensional, i.e. arrays of arrays, or higher), though the result of that may not have any logical application. As implemented, some of these combinations may throw program exceptions, the handling of which is purposefully left outside of the scope of this thesis.

XML:

```
<modifier
  type="(pass|fold|flatten|merge|collect)"
  from="(URI string)"
  alias="(any string)">
  (...)
</modifier>
```

JSON:

```
{
  "type": "modifier",
  "modifier-type": "(pass|fold|flatten|merge|collect)",
  "from": "(URI string)",
  "alias": "(any string)",
  "connections": [(...)]
}
```

- Contents (XML) / **`connections`** (JSON): (...)

  The system expects content elements (XML) or `connections` -array objects (JSON) of one type: `connection` (incoming only)

- **`type`** (XML) / **`modifier-type`** (JSON): "(pass|fold|flatten|merge|collect)"

  Modifiers have a subtype designated by a `type` attribute (XML) or `modifier-type` pair (JSON) with one of the string values `pass`, `fold`, `flatten`, `merge`, or `collect` (for a pass, fold,

flatten, merge, or collect modifier, respectively). If omitted, the `pass` value will be assumed as default.

- **`from`** : (URI string)

  Modifiers can have a `from` attribute or pair to represent a single connection instead of having a list of incoming connections, with a URI value equal to the `from` attribute or pair value of the equivalent incoming connection. If given both a `from` attribute or pair and a list of connections, the list will be ignored.

INPUT

Input contain and modify downstream connections. Inputs can be connection-type inputs, which additionally accept upstream connections. Connection inputs additionally contain a modifier to manipulate the incoming value flow as described in the section on modifiers. Inputs can also be value-type inputs, in which case they contain the user directly (these are equivalent to a connection input connected to a parameter with the same values).

XML:

```
<input
  (name|alias)="(any string)"
  type="(connections|values)"
  modifier="(pass|fold|flatten|merge|collect)"
  from="(URI string)">
  (...)
</input>
```

JSON:

```
{
  "type": "input",
  "(name|alias)": "(any string)",
  "input-type": "(connections|values)",
  "modifier": "(pass|fold|flatten|merge|collect)",
  "from": "(URI string)",
  "connections": [(...)],
  "values": [(...)]
}
```

- Contents (XML) / **`(connections|values)`** (JSON): (...)

  The system expects content elements (XML) or `connections` -array objects (JSON) of one type: `connection` (incoming only), comma-separated numbers (XML) or array of numbers (JSON), or a series element (XML) or object (JSON).

- **`(name|alias)`** : "(any string)"

  Inputs have a name (which is equal to their alias) designated by a `name` (or `alias`) attribute of pair. If the input is contained within an assembler-type component, the name must correspond to one of the input names expected by the assembler. If the input is contained within a model-type component or a modifier, the name is optional and may be arbitrary.

- **`type`** (XML) / **`input-type`** (JSON): "(connectionslvalues)"

  Inputs have a subtype designated by a `type` attribute (XML) or `input-type` pair (JSON) with one of the string values `connections` or `values` (for a connections or values input, respectively). If

omitted, the subtype will be inferred from the presence of `values`, `connections`, or `from` pair (JSON only).

- **`modifier`**: "(pass|fold|flatten|merge|collect)"

  Inputs can have a built-in modifier designated by a `modifier` attribute or pair with one of the string values `pass`, `fold`, `flatten`, `merge`, or `collect` (for a pass, fold, flatten, merge, or collect modifier, respectively). If omitted, the `pass` value will be assumed as default.

- **`from`**: "(URI string)"

  Inputs can have a `from` attribute or pair to represent a single connection instead of having a list of incoming connections, with a URI value equal to the `from` attribute or pair value of the equivalent incoming connection. If given both a `from` attribute or pair and a list of connections, the list will be ignored.

OUTPUT

Outputs accept upstream connections.

XML:

```
<output
  (name|alias)="(any string)"
  from="(URI string)"
  modifier="(pass|fold|flatten|merge|collect)">
  (...)
</output>
```

JSON:

```
{
  "type": "output",
  "(name|alias)": "(any string)",
  "from": "(URI string)",
  "modifier": "(pass|fold|flatten|merge|collect)",
  "connections": [(...)]
}
```

- Contents (XML) / **`connections`** (JSON): (...)

  The system expects content elements (XML) or `connections`-array objects (JSON) of one type: `connection` (incoming from model of model-type component only)

- **`(name|alias)`**: "(any string)"

  Outputs have a name (which is equal to their alias) designated by a `name` (or `alias`) attribute of pair. If the output is contained within an assembler-type component, the name must correspond to one of the output names expected by the assembler (if omitted, the "default" output is assumed). If the input is contained within a model-type component or a modifier, the name is optional and may be arbitrary.

- **`from`** (contained within a model-type component only): "(URI string)"

  Outputs contained within a model-type component can have a `from` attribute or pair to represent a single connection instead of having a list of incoming connections, with a URI value equal to the `from` attribute or pair value of the equivalent incoming connection. If given both a `from` attribute or pair

and a list of connections, the list will be ignored.

- `modifier` (contained within a model-type component only): "(pass|fold|flatten|merge|collect)"

  Outputs contained within a model-type component can have a built-in modifier designated by a `modifier` attribute or pair with one of the string values `pass`, `fold`, `flatten`, `merge`, or `collect` (for a pass, fold, flatten, merge, or collect modifier, respectively). If omitted, the `pass` value will be assumed as default.

CONNECTION

Connections simply connect an output to a downstream input, and have no additional semantics.

XML:

```
<connection
  from="(URI string)"/>
```

JSON:

```
{
  "type": "connection",
  "from": "(URI string)"
}
```

- `from`: "(URI string)"

  Connections have a `from` attribute or pair referencing the connection origin by URI (an outlet object, or a component, parameter, or modifier object in which case the default output is implied).

Resource paths and methods

The `GET` methods of the various resources additionally support the following URL-encoded parameters:

- `(shallow|level|deep)`: `true` (or empty, as flag)

  Determines the depth of the returned resource representation (`level` is the default, and can be omitted; if multiple parameters are included `shallow` takes precedence over `level`, which takes precedence over `deep`):

  - `shallow`: includes only the resource's own properties, and uses URIs to refer to any resources contain within.
  - `level`: includes the resource's representation, and recursively the representations of contained resources, up to a 'sensible' extent (otherwise, uses URIs).
  - `deep`: includes full representations of the resource itself and recursively of any contained resources.

- `(lean|fat)`: `true` (or empty, as flag)

  Determines the scope of the returned resource representation (`lean` is the default, and can be omitted; if both parameters are included, `lean` takes precedence over `fat`):

  - `lean`: omits redundant properties that could be inferred from other properties.

    - `fat` : includes all redundant properties.
- **`pretty`** : `true` (or empty, as flag)

  Determines that returned resource representation is 'pretty-printed', using indentation (two spaces) for human readability.

In all cases where representation data is submitted to the API (i.e. using `PATCH` , `PUT` , and `POST` methods) it is recursively traversed to create or modify any content objects from the included representations as well.

The various methods will return the following return codes:

- **`200 OK`** : Successful `GET` ; the response will contain data.
- **`201 Created`** : Successful `POST` ; the response will contain the URI of the newly created resource as the value of the `Location` header (but no data).
- **`204 No Content`** : Successful `PUT` , `PATCH` , or `DELETE` ; the request was handled successfully as sent, though returns no data.
- **`400 Bad Request`** : The request was received and resolved to its destination, but was there found to be invalid.
- **`404 Not Found`** : The request was received but could not be resolved to its destination.
- **`500 Server Error`** : The server encountered an unexpected error; no data is returned.

It should be noted that URIs are always generated by the server. URIs sent by the user will be ignored, and the newly created or modified resource will not be resolvable from that URI.

COMMON

All resources share the following common path and methods:

- Individual: **`/api/(singular resource noun)/(URI)
    - **`GET`** : Returns a representation of the resource.
    - **`PATCH`** : Modifies the properties of the resource included in the representation.
    - **`PUT`** : Replaces the resource's properties with those of the representation.
    - **`DELETE`** : Deletes the resource *(**warning**: this cannot be undone).*

DEFINITION

- Collection: **`/api/definitions`**
    - **`GET`** : Returns the full list of definitions *(pagination not supported at this time).*
    - **`POST`** : Creates a new `definition` object from the representation.
- Individual: **`/api/definition/(URI)`**
    - **`POST`** : Expecting a **model** representation (or URI), adds the newly created (or referenced, in the case of a URI) model to the definition.

MODEL

- Collection: `/api/models`
  - `GET` : Returns the full list of library models.
  - `POST` : Creates a new re-usable library model from the representation.
- Individual: `/api/model/(URI)`
  - `POST` : Expecting a **component**, **parameter**, or **modifier** representation (or URI), adds the newly created (or referenced, in the case of a URI) resource to the model.

COMPONENT

- Individual: `/api/component/(URI)`
  - `POST` : Expecting a **input** or **output** representation (or URI), adds the newly created (or referenced, in the case of a URI) object to the component.

PARAMETER

- Individual: `/api/parameter/(URI)`
  - `POST` : Expecting a **connection** representation (or URI), adds the newly created (or referenced, in the case of a URI) connection to the parameter.

MODIFIER

- Individual: `/api/modifier/(URI)`
  - `POST` : Expecting a **connection** representation (or URI), adds the newly created (or referenced, in the case of a URI) connection to the modifier.

INPUT

- Individual: `/api/input/(URI)`
  - `POST` : Expecting a **connection** representation (or URI), adds the newly created (or referenced, in the case of a URI) connection to the input.

INPUT

- Individual: `/api/output/(URI)`
  - `POST` : Expecting a **connection** representation (or URI), adds the newly created (or referenced, in the case of a URI) connection to the output.

CONNECTION

- Individual: `/api/connection/(URI)`

  (As specified under the common paths and methods above.)

F Discussion on networking technologies and implications

F.1 Communications protocol: HTTP/TCP

Depending on the requirements of an application, any of a plethora of communications protocols may be most suitable for its networking layer. Aspects which must be considered include performance (in the sense of transfer speed), reliability (packet loss), and logistics (routes). Long-running, interactive communications will benefit from different aspects than discrete, unchanging communications. Applications that aim to be widely deployed must rely on aspects like general availability, contrasted with localised applications which are free to focus on specialised aspects.

Importantly, the choice of communications protocol is also resource-constrained, as their implementation costs likely differ. Consequently, there is always a trade-off between specialised demands and costs. In this light, a suboptimal choice may often be good enough.

The communications types—and consequently the sets of requirements—for the StructuralComponents client-server system primarily follow its interactions with the user. It most prominently features two separate types of interactions:

- **logic manipulation**—where the user retrieves or changes a piece of parametric-associative logic, such as a component or a connection;
- **parameter variation**—where the user interactively varies the value or values of a parameter, notably by dragging a slider.

The first of the two interaction types is oriented around the parametric-associative logic objects, and is contained within separate user actions. The user might for example load a definition, change a component property, or make a new connection. These are discrete actions, with comparatively long spans of time between them, for which it is important that they are communicated reliably and in their entirety.

The second of the two interaction types is more dynamic. For example, as the user slides a parameter value up or down, the system must repeatedly solve the logic based on the new value and send back the result. These are continuous actions, for which it is more important that results are returned quickly but a certain degree of loss is acceptable—since a lost piece of information will quickly be replaced by another.

It will be clear that these two interactions pose different sets of network protocol requirements. For the software implementation of this thesis, both will make use of HTTP over TCP. This protocol is an optimal choice for the first type of interaction oriented around objects, although suboptimal for the second, streaming interaction.

> **HTTP** stands for *HyperText Transfer Protocol* (Berners-Lee, 1991). It was invented for the *WorldWideWeb* project and currently underpins much of the internet. It revolves around client requests identifying certain resources to which servers respond with a status code and the requested resource data. Nominally, the server only responds to requests, and does not itself initiate communication.
>
> **TCP** stands for *Transmission Control Protocol* (DARPA, 1981). Its specifics are not relevant within the scope of this thesis, so it suffices to state that the protocol emphasises reliability and data integrity over transmission speed and latency.

Box F.1: HTTP and TCP

The HTTP/TCP protocol suits the first type of interaction well due to its reliability and ease of implementation. Reliability is important since the data carrying interaction semantics—e.g. a number value, or a string identifying

some property—is not naturally redundant. That is, any data integrity error will likely change those semantics and yield unexpected results. In simple terms, the protocol ensures that data is either received fully or not received at all—using a combination of retrying, rerouting, and checking completeness.

The HTTP protocol was not designed for the second, streaming type of interaction. It is one-way, has comparatively large data overhead, does not provide any methods of dynamically throttling transmission rate, and many of its features are simply not required. However, due to the protocol's wide availability, many solutions are available that work around the issues and in a rudimentary way enable it to handle streaming communications in spite of its shortcomings. Consequently, the protocol was deemed good enough for the purposes of this thesis.

For the sake of completeness, further investigation into a specialised, streaming protocol such as *UDP* or *WebSocket* will be left as a recommendation (see Section 7.2).

F.2 Communications interface: REST

While data formats and transmission are generally handled by the system libraries implementing the underlying networking protocol, there always remains a translation step between these libraries and application logic. Different sets of requirements determine the suitability of a particular architectural style or framework for such a communications interface than for a protocol. Whereas important aspects for the latter include performance, reliability, and integrity, the main consideration for the communications interface is whether to transfer intention or state.

> As a side note, it should be stated that in the case of retrieving a resource from a server, there are few difference between interface styles. In simple terms, all styles will require the resource to be somehow identified and subsequently return a copy. It is primarily when data is being sent to the server that the differences will become apparent.

After retrieving a representation of a record out of a database, a client may wish to somehow alter it. When transferring intention, the client would proceed to identify the record, indicate what is being altered, and send the new value. The server would interpret the intention and make the actual change. In many cases, identification of the resource is unnecessary as the server has kept track of what was first retrieved. This interface style resembles the object-oriented programming style of calling an object's method.

As an alternative to transferring intention, the client might itself change the retrieved representation and subsequently send it back to the server. The act of sending it back would itself signify that there is an alteration, and the server would proceed to replace the database record with the newly represented version.

The former, intention-transfer style has a range of readily available implementations, such as SOAP (Simple Object Access Protocol; W3C 2000). The latter, state-transfer interface style is pioneered by ReST (Representational State Transfer; Fielding, 2000). Both of these can use HTTP as their transfer protocol, and generally have extensive—often system-provided—libraries to generate an interface to suit application logic.

Both interface styles mentioned have their merits, and both have demonstrated wide applicability. Certainly for the requirements of this thesis, both styles would suffice. While it might be investigated which of the two would ultimately be more suitable, this falls outside the scope of this thesis. Instead, following an IT industry adage that says "unless you have a definitive reason to use SOAP use REST" (Francia, 2010), the choice was made to use implement this thesis using a RESTful style.

REST AND ITS INTERFACE DESIGN CONSTRAINTS

REST (*Representational State Transfer*) is generally described as a style of interface architecture. Many software libraries exist that facilitate implementation, but any interface that adheres to the style philosophies, encoded as

the *REST Constraints*, is considered RESTful. REST was first introduced by Fielding (2000), who described it as a generalisation of the pre-existing HTTP protocol and its underlying philosophies. As such, the style was modelled on, but also enveloped the way the way the *World Wide Web* ("web") already worked.

Of the six *REST Constraints*, three are relevant to the scope of this thesis. For the sake of completeness, the full list of constraints is *client-server*, *stateless*, *cacheable*, *layered system*, *code on demand*, and *uniform interface*. Cacheability refers to the possibility of clients storing previously retrieved data to increase performance. The layered system constraint ensures that REST interactions can be routed over the heterogeneous internet infrastructure. Code on demand refers to the retrieval of executable code from the server. These constraints are important, but not directly relevant tot his thesis—in contrast to the following list:

- The **client-server** architecture separates the various concerns of a distributed system. The client is not concerned with such things as data storage, or (specific to thesis) computationally expensive parametric-associative logic resolution. The server is not concerned with user interface and data presentation. This strict separation prohibits assumptions to be made, increasing modularity and portability.
- **Stateless** communication requires that each request to the server is self-contained, and can be interpreted without the need to make assumptions. Similarly, server responses contain the full extent of information required to interpret the new state. This again increases modularity and portability, and ensures that the system is scalable—as any second machine can at any time pick up from previous interaction.
- The **uniform interface** constraint limits the scope of interactions to whatever is naturally provided by the underlying transfer protocol. In the (most common) case of HTTP, this restricts REST interaction to revolve around URIs and the following (sub)set of *verbs*:
    - **GET**: Retrieve the resource at the specified URI.
    - **PUT/PATCH**: Update the resource at the specified URI (respectively in full or in part).
    - **POST**: Create a new resource under the specified URI (which can be a collection of, or an existing resource that has a to-many relationship with the posted resource type).
    - **DELETE**: Delete the resource at the specified URI.

G Example JSON listings

The following JSON listings represent the parametric and associative definitions, using the API specification of Appendix E, of the examples found in Chapter 6.

```
{ "type": "definition",
  "models": [{
    "objects": [
      { "type": "parameter", "alias": "n"},
      { "type": "parameter", "alias": "r"},
      { "type": "parameter", "alias": "pi", "values": [3.14159]},
      { "type": "parameter", "alias": "phi", "values": [0.7]},
      { "type": "parameter", "alias": "dr", "values": [11]},
      { "type": "component",
        "alias": "r^2",
        "assembler": "exponent",
        "inputs": [
          { "name": "base", "from": "@r"}
        ]},
      { "type": "component",
        "alias": "n * pi",
        "assembler": "product",
        "inputs": [
          { "name": "multiplications",
            "modifier": "collect",
            "connections": [
              { "from": "@n"},
              { "from": "@r"}
            ]}
        ]},
      { "type": "component",
        "alias": "d - dr",
        "assembler": "sum",
        "inputs": [
          { "name": "additions", "modifier": "fold", "from": "@r"},
          { "name": "subtractions", "modifier": "fold", "from": "@dr"}
        ]},
      { "type": "component",
        "alias": "(r - dr)^2",
        "assembler": "exponent",
        "inputs": [
          { "name": "base", "from": "@{d - dr}"},
        ]},
      { "type": "component",
        "alias": "r^2 - (r - dr)^2",
        "assembler": "sum",
        "inputs": [
          { "name": "additions", "modifier": "fold", "from": "@{r^2}"},
          { "name": "additions", "modifier": "fold", "from": "@{(r - dr)^2}"},
        ]},
      { "type": "component",
        "alias": "total by phi",
        "assembler": "product",
        "inputs": [
          { "name": "multiplications",
            "modifier": "collect",
            "connections": [
              { "from": "@phi"},
              { "from": "@{n * pi}"},
```

```
            { "from": "@{r^2}"}
          ]}
      ]},
    { "type": "component",
      "alias": "total by dr",
      "assembler": "product",
      "inputs": [
        { "name": "multiplications",
          "modifier": "collect",
          "connections": [
            { "from": "@{n * pi}"},
            { "from": "@{r^2 - (r - dr)^2}"}
          ]}
      ]}
  ]
 }]
}
```