A Framework for Analysis and Dynamic Visualisation of Mechanisms

Arjun Nagarajan, Sandipan Bandyopadhyay

Abstract

Position analysis and the evaluation of various performance indices are integral parts of iterative design of mechanisms using computers. To perform these tasks, one relies upon either a commercially available software or self-written codes. While the first option suffers from the lack of flexibility in terms of integration with other external modules such as an optimiser, the second one requires significant amounts of time and effort in terms of planning, programming, debugging, and code maintenance. In this paper, a unique solution to this problem is proposed, via the introduction of a meta-programming language called MML, developed for the specific purpose of modelling and analysing mechanisms. A few lines of codes in MML is enough to describe a mechanism, solve its position kinematic problem, and also to generate automatically programmes in C language that can be either compiled externally to create stand-alone analysis modules, or be integrated with any other system accepting a C module. Further, to complete the framework as a stand-alone analysis package, a Qt-based visualisation interface is added. It allows dynamic manipulation of the design parameters via the GUI elements, and updates the screen with the corresponding effects on motion and/or some pre-defined output function or performance index in the real time. The features and the usage of the framework is illustrated with the example of a Stephenson-III six-bar mechanism. The framework, however, is capable of handling more generic mechanisms and is designed to be easily extendable. It is hoped that the mechanism design community would find this framework of some interest and utility.

Keywords: Planar mechanisms, Graph theory, Dynamic visualisation, *flex*, *Bison*, Qt, Meta-programming

1 Introduction

Computerised analysis has become a routine part of the mechanism design process. With the availability of faster computers and better algorithms, there is a perceptible drift from the traditional close-form geometric synthesis methods to more general, numerical optimisation-based design procedures. Some of the key steps in such a design process is the position analysis, as well as the visualisation of the intermediate/final results. While there are plenty of commercial packages available for that purpose, (e.g., `ADAMS`, `RecurDyn` etc.), there are hardly any tools which provide the user a

Arjun Nagarajan
Department of Engineering Design, IIT Madras, Chennai - 36, E-mail:arjun.2048@gmail.com.

Sandipan Bandyopadhyay
Department of Engineering Design, IIT Madras, Chennai - 36, E-mail:sandipan@iitm.ac.in.

mathematical model of the mechanism that can be exported from the system, and integrated in another arbitrary tool (e.g., an optimiser), or analysed separately by the designer without restricting herself within the APIs (Application Programming Interface) exposed by a certain given software. On the other hand, writing one's own code for the design and analysis gives complete flexibility in terms of choosing appropriate algorithms at each stage, i.e., kinematic analysis, design objectives and constraints, optimisation algorithm etc. However, it is typically cumbersome to programme all of these from the scratch.

The present work tries to bridge this gap in a novel manner. The objective is to create an analysis and visualisation framework backed by a new computer *language* designed for the exclusive purpose of modelling mechanisms. The name of the language reflects this fact: "Mechanism Modelling Language" or "MML" in short. Using this language, the user would be very easily be able to construct a mechanism from simple elements such as points, lines and joints etc. Once the mechanism is created, the algorithms built into the system try to find suitable (pre-programmed) solvers for the position kinematics of the mechanism. If successful, the system is capable of solving the position kinematics problem, including identifying its different branches. With this solution, it can animate the mechanism through a visualisation interface. More importantly, it can construct the *loop-closure* equations for the user, and even generate functions (in `C` programming language) for solving the position kinematics problem. These functions can then be readily integrated in any optimiser/analysis tool, which provides a `C`-based API, or the user's own code for further analysis.Moreover, as the code is generated by the system from pre-defined and tested modules, it is guaranteed to be free of bugs and errors. Thus, the proposed framework delivers a unique combination of flexibility, productivity, as well as reliability.

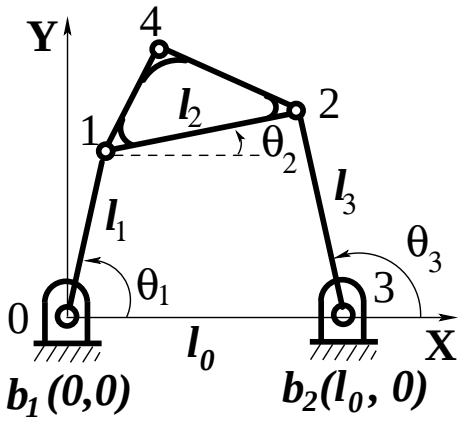
Position analysis of planar mechanisms on a computer has been studied for several decades. Shu and Radcliffe [1] were among the first to suggest that position analysis of a complex mechanism can be done by notionally decomposing it into simpler modules, and solving these modules in the right sequence. Many others (e.g., [2]) have extended this idea and presented procedures to derive equations for the component module. Understanding the connectivity between the different links is a prerequisite for the modular decomposition. It turns out that using various connectivity matrices, it is possible to identify mechanisms of a given *topology* (or kinematic structure) [3, 4]. The framework under discussion employs similar concepts. To create the first *proof-of-concept* (PoC) prototype of the system, only planar mechanisms with revolute joints are considered. Further, this paper is confined to the discussion of six-bar mechanisms only.

The paper is structured as follows: in Section 2, different graph-based representation of mechanisms are described very briefly. In Section 3, some features and uses of MML are illustrated with examples. The dynamic visualisation interface is described in Section 4. The conclusions are presented in Section 5.

2 Representing a mechanism

One of the key requirements of achieving the objectives described above is to have a mathematical model representing mechanisms on a computer. It is essential that the

computers can *understand*, analyse, store and retrieve mechanisms. It turns out that mechanisms can be easily represented as *graphs*, which are very much amenable to computerised analysis. In this work, therefore, mechanisms are represented as graph objects, *undirected graphs* in particular. The revolute joints are represented as nodes, and the links as edges (excluding the grounded links). Thus, an *adjacency matrix* or an *adjacency list* coupled with additional attributes of the nodes (i.e., joints) viz. input, fixed, movable etc. represent the mechanism adequately. These representations are interchangeable, and both are used in the algorithms developed. The adjacency matrix is useful in establishing properties of the mechanism, like the characteristic polynomial, and the adjacency list is used in algorithms finding the shortest path etc.



(a) A planer revolute-jointed four-bar mechanism

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 1 & & \\ 0 & 2 & 4 \\ 1 & 3 & 4 \\ 2 & & \\ 1 & 2 & \end{pmatrix}$$

(b) Adjacency matrix $\mathbf{A}$ (c) Adjacency list A_L

Figure 1: (a) Schematic of the four-bar mechanism, (b) its adjacency matrix, and (c) the adjacency list representation

3 Elements of the MML

A small and intuitive language has been developed for the specific objective of modelling and analysing mechanisms. This language provides a simple and concise means to describe and analyse mechanisms. The user now has to write only a few lines of

code in MML to build and analyse a mechanism, instead of writing separate programs for each mechanism. To develop an interpreter for this new language, CASE tools such as *flex*, a fast lexical analyser [5], and *Bison*, a parser generator [6], has been used. Bison requires an *grammar* definition, which it converts into a parser that can be coupled with the lexer generated by the flex module.

3.1 Features of the language

Attempts have been made to keep the keywords of the language as intuitive and close to a natural language (i.e., English, in this case) as possible. For example, a set of keywords and their corresponding actions is given in table 1.

Table 1: A partial list of key-words recognised in MML

add	to add a link or point
base(x,y)	base point at (x,y)
point(x,y)	movable point at (x,y)
link(p0,p1)	link connecting points p0 and p1
setinput(n)	set n as input node
show	display list of variables used
shownodes	display list of nodes
showlinks	display list of links
amat	display adjacency matrix
alist	display adjacency list
solve	solve the mechanism
plot(X,Y)	plot X vs. Y

3.2 Building a mechanism using MML

One of the most important aspects of this framework is to represent the mechanism in a computer. For this, the simplest way is to mimic the way a mechanism is drawn on paper: the user builds it from smaller building blocks such as fixed base points, links and joints. Multi-loop mechanisms can be built up in stages. For example, one can construct a six-bar mechanism by first creating a four-bar, and then adding a RR-dyad between the coupler point and another point on the fixed base. The process is illustrated through the example of a *Stephenson-III* mechanism shown in figure 2. The following few lines suffice in creating the Stephenson-III (see Fig. 2), consisting of 7 joints and 7 links. The first set of commands corresponds to adding nodes, and the next set adds links connecting these nodes.

```
add base(0,0);        //Add node 0, a fixed point
add point(0.1,1);     //Add node 1, a moving point
add point(0.5,1.5);   //Add node 2, a moving point
add point(1,2.2);     //Add node 3, a moving point
add base(2.5,2);      //Add node 4, a fixed point
```

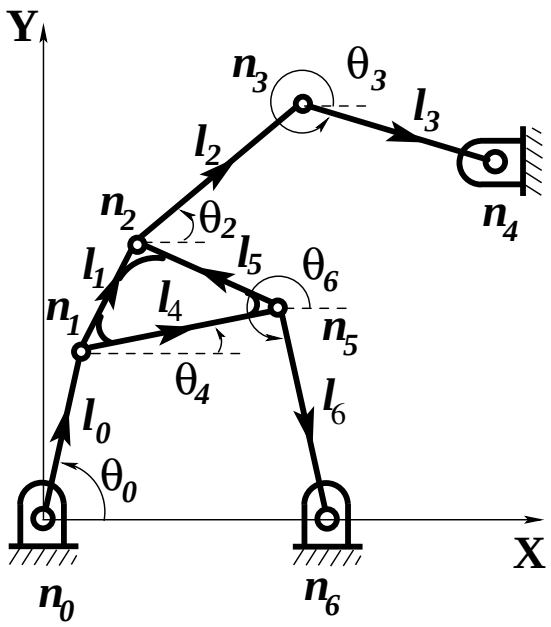


Figure 2: Schematic of the Stephenson-III mechanism with node and link designations

```
add point(1,1.2);     //Add node 5, a moving point
add base(1.4,0);      //Add node 6, a fixed point

add edge(0,1); //Add link 0, by connecting nodes 0 and 1
add edge(1,2); //Add link 1, by connecting nodes 1 and 2
add edge(2,3); //Add link 2, by connecting nodes 2 and 3
add edge(3,4); //Add link 3, by connecting nodes 3 and 4
add edge(1,5); //Add link 4, by connecting nodes 1 and 5
add edge(5,2); //Add link 5, by connecting nodes 5 and 2
add edge(5,6); //Add link 6, by connecting nodes 5 and 6

setinput(0); //set node 0 as the input (actuated) joint
```

The last line designates the node "0" as the actuated one. These input lines are parsed and processed to generate, internally, a list of nodes (*nodelist*), a list of edges (*edgelist*), and the adjacency matrix is also formed. The description of the mechanism is thus complete.

3.3 Framing loop-closure equations

By definition, all the links in a mechanism has pre-defined, predictable motion if the motion of the input link is given. This is made sure mathematically by framing the *loop-closure* equations and solving for the position of the *passive*, i.e., non-actuated

links from these equations. To do this from the above model of the mechanism, one needs to identify *all* the loops in the mechanism, and choose the correct number of *shortest* loops from them.

A loop in a mechanism is equivalent to a *path*[1] in a graph. Generally loop-closure equations are formed between the base nodes. Thus framing loops is equivalent to identifying shortest paths between the base nodes. For example, consider the Stephenson-III mechanism shown in Fig.2. Here the shortest path between the base nodes n_0 and n_6 is given by the sequence n_0 - n_1 - n_5 - n_6. Thus the loop-closure equations are framed based on this path:

$$n_{0x} + l_0 \cos\theta_0 + l_4 \cos\theta_4 + l_6 \cos\theta_6 - n_{6x} = 0$$
$$n_{0y} + l_0 \sin\theta_0 + l_4 \sin\theta_4 + l_6 \sin\theta_6 - n_{6y} = 0$$

The system is capable of finding the shortest loop and the corresponding equation without any manual intervention.

3.4 Solver identification

The solution of the loop-closure equations *automatically* (i.e., without any manual intervention) is one of the most important requirements for any mechanism modelling framework. In this case, it is achieved by decomposing the mechanism into *known* components, for each of which a dedicated solver is available. The solver is capable of finding *all* the solutions, as it employs pre-defined analytical solutions to these components. This, is a key difference between the present work and standard commercial software. The latter typically use numerical solvers for the non-linear problem of position kinematics, yielding only one of the multiple possible solutions[2].

In this paper, the solutions are restricted to mechanisms which can be decomposed into RR-dyads (or four-bars) and rigid coupler links. However, it may be noted that algorithm is designed to be modular, and as such, if other solvers are added, the algorithm can map them to the target mechanisms/components. The output of this algorithm is an ordered list of nodes, and the appropriate solver to be used for each node in the case of a successful termination, and a failure message otherwise. The algorithm takes care of the dependency of the position of nodes on the other nodes, the order of the output list of the nodes is the same order in which the nodes have to be solved. To illustrate the process, the process log generated while applying it to the Stephenson-III mechanism is shown below. Here, “FIXED” implies a base node, “INPUT” an actuated node, “RRDYAD” a node belonging to two R-jointed links with known end points, “COUPLER” a part of a rigid coupler link.

```
Mechanism can be solved
Node 0 dependants:      Solver: FIXED
Node 4 dependants:      Solver: FIXED
```

[1] Path is an *open walk* in which no vertices are revisited.

[2] For the same reason, the capability of the framework is limited. It can only solve the position kinematics problem if the mechanism can be decomposed into components, for each of which a solver is available. A large number of planar mechanisms, and indeed a majority of those used in machine components, are typically four-bar and six-bar mechanisms; hence from a practical standpoint, this limitation is not very significant.

```
Node 6 dependants:       Solver: FIXED
Node 1 dependants: 0     Solver: INPUT
Node 5 dependants: 1  6  Solver: RRDYAD
Node 2 dependants: 1  5  Solver: COUPLER
Node 3 dependants: 2  4  Solver: RRDYAD
```

Solvers for these components are readily available, the mechanism can be solved by calling the appropriate solvers in the above sequence. In addition to finding the solutions to the position kinematics problem in this manner, a key objective of this work is to generate `C` code embedding the solvers in it, so as to generate a stand alone `C` code, which when compiled, would create a simulator for the given mechanism. It is this *meta-programming* that makes this framework absolutely unique in the mechanism domain (to the best of the authors' knowledge)[3].

4 Visualisation interface

A basic visualisation interface, capable of allowing dynamic manipulation of the mechanism's geometric attributes (i.e., link lengths, base points etc.), has been included in the framework for the sake of completeness. It has been developed using the non-commercial version of the `Qt` GUI libraries. A screen-shot of the same showing the Stephenson-III mechanism along with the coupler-curve of the corresponding coupler point of the lower four-bar is shown in Fig. (3). It is very well-known that while blind-

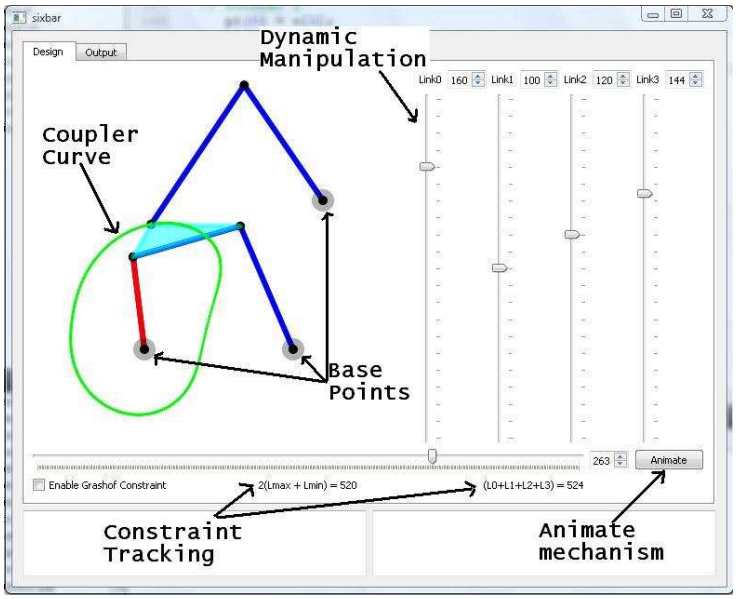


Figure 3: Screen-shot of the visualisation interface

[3]Implementation of this part of the framework is still under progress.

fold mathematical optimisation can produce excellent sets of design parameters, much is learnt by studying the effect of the variation of the different geometric parameters. In many cases, it is possible to arrive at an *acceptable* solution by manually changing the parameters, while keeping track of the objectives and constraints. Objectives and constraints can be programmed in C and integrated with the system. Some generic constraints are included in the system; e.g., while modelling four-bar mechanisms, one may choose to remain within the domain of Grashof mechanisms, and thus by selecting an on-screen option, can automatically define such limits on the range of the link lengths that Grashof's condition is satisfied.

5 Conclusions

In this work, an attempt was made to contribute to the broader goal of designing mechanisms by creating a new framework for analysis and dynamic visualisation. Mechanisms were represented as adjacency lists and adjacency matrices of the joints. Algorithms to frame loop-closure equations were developed. Algorithms to solve the position kinematics problem of a planar mechanism by decomposing it into known modules were demonstrated. These algorithms were tested on sample problems and results were reported. A language to describe and handle mechanisms was conceptualised and implemented. The impact of the meta-programming approach in terms of the significant reduction in the effort to develop and debug the position kinematics code have been demonstrated.

A visualisation interface, to aid the user in studying the performance of mechanism while dynamically varying the design parameters, was implemented.

The work reported here is still in its initial stage of development, and many extensions have already been planned. It is hoped that once completed, the framework may help the mechanism community in their design/analysis activities.

References

[1] C. H. Suh and C. W. Radcliffe, *Kinematics and Mechanism Design*. Wiley, 1978.

[2] G. L. Kinzel and C. Chang, "The analysis of planar linkages using a modular approach," *Mechanism and Machine Theory*, vol. 19, no. 1, pp. 165 – 172, 1984.

[3] T. S. Mruthyunjaya and M. R. Raghavan, "Structural analysis of kinematic chains and mechanisms based on matrix representation," *Journal of Mechanical Design*, vol. 101, no. 3, pp. 488–494, 1979.

[4] J. Uicker Jr. and A. Raicu, "A method for the identification and recognition of equivalence of kinematic chains," *Mechanism and Machine Theory*, vol. 10, no. 5, pp. 375 – 383, 1975.

[5] Flex manual, *Lexical Analysis With Flex*.

[6] Bison manual, *Bison - GNU parser generator*.