MANAGING COMPLEXITY IN SCIENTIFIC SOFTWARE

SHARMIN ISLAM
Bachelor of Science, Military Institute of Science and Technology, 2013
Master of Business Administration, Bangladesh University of Professionals, 2015

A thesis submitted
in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

MANAGING COMPLEXITY IN SCIENTIFIC SOFTWARE

SHARMIN ISLAM

Date of Defence: April 27, 2020

Dr. Shahadat Hossain Supervisor	Professor	Ph.D.
Dr. Saurya Das Committee Member	Professor	Ph.D.
Dr. Robert Benkoczi Committee Member	Associate Professor	Ph.D.
Dr. Howard Cheng Chair, Thesis Examination Committee	Associate Professor	Ph.D.

Dedication

To my beloved parents and

my son, Rayan.

Abstract

One of the expected benefits of a modular design is flexibility. By the word "flexibility" we mean possibility of drastic changes to a module without changing or without knowing other modules. Based on the evolutionary data available on version control systems, it should be possible to analyze the quality of a modular software architecture and decide whether it is worth to restructure its design. In this thesis we investigate this issue using a novel approach based on a general theory of modularity that uses design structure matrices (DSM) for reasoning about quality attributes. Using our approach, we can categorize the functions in different tiers. This finding suggests that the analysis of different tiers of functions of a software system might serve as guidance to developers in the challenging task of redesigning a software by detecting and retrieving components that could be reused in other software projects.

Acknowledgments

First and foremost, I would like to thank the Almighty for giving me the opportunity, strength, and patience to undertake this research. This work would not have been possible without His blessing.

I am lucky that I have worked under the supervision of Dr. Shahadat Hossain. The way he treated me, it felt like he is not my supervisor instead my guardian. It may be difficult for me to work under any supervisor in future after working with such a great person. Thank you, Sir, for everything.

I want to express my sincere gratitude to my supervisory committee members, Dr. Saurya Das and Dr. Robert Benkoczi. Their guidance, encouragement, and suggestions helped me a lot. Their immense efforts and the way of directing the students of optimization research group can be a model to others.

Without the encouragement I got from my families it would not be possible for me to come to this far and go forward. I am very grateful to my parents as well as to my parents-in-law and all the members of my two families. I want to thank my husband Wali. Without him, my life would be a lot more difficult.

I also want to thank all my friends and well-wishers as well as all the members from the Optimization group.

I am thankful to the Alberta Innovates for Technology Futures Graduate Student Scholarship, and the School of Graduate Studies (SGS) of the University of Lethbridge for their financial support.

Last but not least, I am also grateful to the researchers for their ideas and contributions in this field.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software systems can be viewed as a network of components joined together by dependence relationships. In software and other technological systems such as process, product, or organizational architectures some of its functionalities are realized by the interaction pattern of components or subsystems [7]. For example, modular software systems allow tracking bugs to a small number of well-defined subsystems or modules.

Many scientific software are usually written by domain experts and address some specific scientific computing problem. For instance, CSparse software implemented in `C` is concerned with solving system of linear equation $Ax = b$ where the coefficient matrix A is sparse. ADOL-C is a software system to compute mathematical derivatives (gradient, Jacobian, Hessian, Taylor coefficients) of a mathematical function available as a computer program in a programming language (`C`). The software applies algorithmic differentiation techniques to compute accurate (upto machine precision) numerical derivatives of the function program at a specified point.

A convenient tool to represent and analyze architectural complexity of these software that is popular among systems engineers and architects is the so called Design Structure Matrix (DSM), and extensions Domain Mapping Matrix (DMM), and Multi Domain Matrix (MDM) [7]. Since a DSM can be represented by a matrix in $\mathbb{R}^{m\times n}$, it is amenable to analysis by sophisticated linear algebraic methods such as singular value decomposition (SVD). Moreover, utilizing the duality of a sparse matrix and its graph, a sparse matrix A can be rearranged (through permutation P) into a computationally beneficial form called "block

triangular form (BTF)" $P^{\top}AP$ also known as "partitioning" in DSM community.

Professor Hossain and his group have proposed using DSM to model and analyze dependence complexity of scientific software systems [14, 15, 1, 13]. One of the main motivations was to study legacy code to determine the dependency information through static call graphs. The information gathered can be utilized to restructure or reuse the components by analyzing the dependence information using techniques from the emerging filed of complex networks [9].

In a recent work, Professor Hossain and his group have studied architectural properties of a small suite of representative scientific software [15]. The studied software tools display shorter characteristic path lengths, small nodal degrees, and small propagation costs, similar to general-purpose software such as operating systems [4, 19].

For variety of reasons legacy software may not contain adequate technical documentation so that from a usability point of view it may be difficult to detect and retrieve components that could be reused in other software projects. In this thesis we study software systems specifically designed for problems arising in scientific and engineering applications [16].

Analytics tools such as "Understand" [21] allows us to view dependency structure of the software at varying level of details: file, class, function, statement etc. In our work we analyze the dependency structures of programs where caller-callee relation between functions captured by static call graph depicts fundamental control flow in the program.

Static call graphs are limited to portraying direct dependencies among design elements (in our case functions). In this paper we are interested in uncovering "similarity" among design elements. We can then use a suitable similarity metric to partition the design elements among groups or clusters where the elements in the same group are "similar" in a certain way. An immediate application of such a decomposition is the ability to retrieve group of "similar" functions from a software repository. Combined with the notion of "importance" of design elements [13], our goal in this work is to group or cluster the design elements into

"tiers" ranked by their "importance".

1.1 Preliminaries

1.1.1 Scientific Computing Software

Thousands to millions of lines of source code make software systems as complex products. The software system depends on design decisions, internal and external constraints, different technical and non-technical concerns [11]. Development of scientific computing software applications is considered as a proof-of-concept tool. But powerful hardware resources facilitated scientific software to solve and simulate large problem. The recent development of more powerful hardware resources encourages to increase number of scientific applications which simulate more effectively and efficiently than the applications built before [15]. These simulation software applications are highly complex and contain millions of lines of computer code. These applications have significant investment in time and other computational and manpower resources. Re-usability, efficiency, portability, correctness, robustness and ease of use are various attributes of scientific software.

One or more independently developed modules compose a software system. We can consider each module as a segment of the software. From these modules we can find their dependencies. For example, we say, Module A depends on Module B when Module A uses (calls) Module B. Here we describe two types of software dependencies: static and dynamic. Static dependencies are extracted from code that is not in execution state and use source code as input. Dynamic dependencies are extracted from the code in an execution state and use executable code and the program state as input. The problem of dynamic dependency is the presence of some subroutines which are executed only at run time. The benefit of considering static dependency is using source code as input and does not relying on program state. This is why we have considered static dependency in our work.

1.1.2 Software Architecture

Architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution [20]. The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [2].

The complexity of large software systems can be identified easily using software architecture. For a software system, its architecture is considered as its high level structure. In other words, software architecture is an abstraction of a complex system. There are some benefits of this abstraction, such as:

1. The software architecture can give a basis for analysis of the behavior of software systems.

2. It can save design costs by providing a basis for re-use of elements (A complete software architecture or parts of it) whose stakeholders require similar quality attributes or functionality.

3. Early design decisions can be made which helps in software development life cycle (development, deployment, and maintenance).

1.1.3 Dependency Relationship

A dependency relationship can be applied between elements of a system to indicate that a change in one element may result in a change in other elements if there exists dependency.

Dependency relationship can be viewed as a complex network. This complex network view has been successfully applied in numerous areas. For example, the human cerebral cortex (a complex system) has been used in the paper [5]. That work reported local and global differences between diseased patients and controls by evaluating communicability measure of weighted networks.

1.2 Our Contribution

A software may not contain adequate technical documentation for variety of reasons. So it may be difficult to detect and retrieve components that could be reused in other software projects. Our goal is to retrieve group of "similar" design elements from a software repository into "tiers" ranked by their "importance".

Our contributions to achieve this goal are pointed below.

1. Analyzing of dependency structure of the software by capturing caller-callee relationship between functions using tool "Understand" [21].

2. Ranking the design elements (functions) using the notion of "importance" of those design elements [13].

3. Using a suitable similarity metric (cosine similarity) to partition the design elements (functions) among groups or clusters.

4. Grouping the design elements into "tiers" ranked by their "importance".

5. Numerical experiments to show the results of our implementation.

1.3 Thesis Organization

There are a total of 6 chapters in this thesis. Chapter 1 is the introductory chapter where we introduce the problem and significance of solving the problem in general. Then we present definitions and description of scientific computing software and their architectures as well as dependency relationship concepts. We also discuss our contribution and thesis organization in this chapter.

In Chapter 2, we discuss the description of the dependency extraction and modeling. We describe some tools that are used to extract and visualize call graphs. At the end of this chapter, we mention the tool that was used in our thesis.

Detailed description about component centrality using a spectral method has given in Chapter 3. Before explaining these methods, with some small examples, we describe eigenvalues, eigenvectors, hub and authority.

Chapter 4 includes a brief discussion on the methodology of our novel approach to analyze the scientific software followed by a detailed description of the target systems. These systems include CSparse and ADOL-C. Then we discuss the implementation of the algorithm. Finally, we report and discuss our results from the experiments.

We give the concluding remarks and future work directions in Chapter 5.

Chapter 2

Dependency Extraction and Modeling

2.1 Design Structure Matrix

The Design Structure Matrix (DSM) is a simple, compact and visual representation of a system or project in the form of a square matrix [7]. The DSM is a network modeling tool used to represent the elements comprising a system and their interactions. It highlights the system's architecture or the relationships between elements in a system by examining the dependencies that exist between its elements in a square matrix.

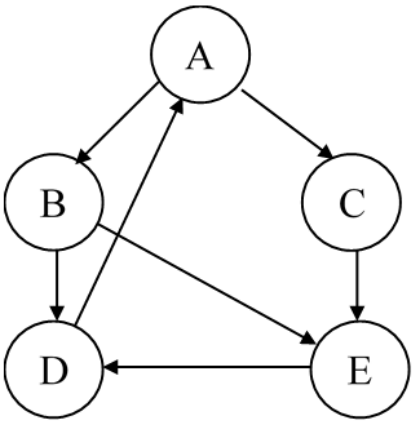
To analyse a system, DSM models can be rearranged or partitioned using various analytical methods, such as, clustering and sequencing.

DSM offers following advantages:

- The DSM provides a compact representation format for large, complex system.
- The DSM highlights a system-level view to a system designer which supports more globally optimal decision making.
- The basic structure of a complex system becomes understandable because of the DSM.
- The DSM is represented using a square matrix. Hence a number of powerful analysis in graph theory and matrix mathematics as well as specialized DSM analysis methods are applicable to DSM.

We use a simple example to show the element relationships (see Figure 2.1). We note that the system that is composed of five elements (or sub-systems): "A", "B", "C", "D" and

"E". We assume that the five elements completely describe the system and characterize its behavior while we use DSM for the modeling purpose. To represent this system pictorially we use graphical form. The system graph is constructed by allowing a vertex/node on the graph to represent a system element and an edge joining two nodes to represent the relationship between two elements. The directionality of influence from one element to another is captured by an arrow instead of a simple link. For example, we can see there is an arrow from Element *A* to Element *B*. If these elements are considered as function then we can say Function *A* calls Function *B*. Therefore Function *A* is caller and Function *B* is callee. The resultant graph is called a directed graph or simply a digraph (shown in Figure 2.1 a).



(a)

Element	A	B	C	D	E
A		X	X		
B				X	X
C					X
D	X				
E				X	

(b)

Figure 2.1: An example of DSM: a) Dependency Graph, b) Dependency Matrix

We can represent DSM using matrix form. The matrix representation of a directed graph has some properties, such as, it is binary (unweighted); or it can also be weighted, it is square (it has n rows and columns where n is the number of nodes of the digraph), it has k non-zero elements (k is the number of edges in the digraph).

The elements' names are placed down the side of the matrix as row headings and across the top as column headings in the same order (see Figure 2.1 b). If there exists an edge (relation) from node x to node y, then the value of $Element[x][y]$ is marked with an X. Otherwise,

the value is left empty. The diagonal elements of the matrix do not have any interpretation but in some cases they are considered as representative of the nodes themselves. For binary matrices X is 1 and for other matrices X means numeric value.

2.2 Dependency Extraction

Investigating program dependencies such as function calls is challenging for very large systems [22]. There are two main classes of the call graph extractors: Lightweight and Heavyweight. A fraction of the entire static information is provided by lightweight extraction, on the other hand, heavyweight extractors provide a complete call graph. Heavyweight extractors again can be categorized in two types: strict and tolerant. Like compilers, a strict heavyweight extractor stops when there is a lexical or syntax error. Tolerant extractor provides complete static call graphs.

Understand [21] is a tool which can be used for dependency extraction. In this thesis, we have used this tool. *Understand* is used to analyze the dependencies between software artifacts in a project. The application supports a wide range of programming languages, including Java, C#, C++ classes & Ada packages for dependency information and can access dependency information from the C++ and PERL APIs. Using this software, we can navigate codes using a detailed cross-referencing and visualize them using graphical representations. Analyzing any source codes means analyzing its different units because these units have some distinct features which reflects on source codes. *Understand* has architecture features that help us to create hierarchical aggregations of source code units. Dependencies between these units can be observed from the Dependency Browser and visualized by Dependency Graphs. However, we can consider different parameters, such as, nodes, files, classes, packages, and interfaces to observe dependencies between different units of the source codes using the Dependency Browser. This tool has following features to observe dependencies[21]:

- Rapid browsing of dependencies for files and *Understand* architectures

- List “dependent” , and “dependent on” entities, for files and architectures
- Spreadsheet export of dependency relationships
- A Dependency Browsing dock that shows all dependency information

For a brief explanation of the dependencies that can be extracted using *Understand*, we can consider a demo source code shown in Figure 2.5. Now we can demonstrate this example to show different dependencies:

- Include dependency: The Include Dependency graph shows the files that are needed to be included for implementation. For example, a file “xyz.c” is dependent on another file “xyz.h” means that “xyz.c” has included the file “xyz.h”. This dependency can be found using *Understand*.
- Call dependency: Using *Understand*, Figure 2.2 shows calls relationship from *A* to *D*. We can check whether there is a relationship between two elements of the system or not. Then using the tool *Understand*, we can also see Figure 2.3 where the files/ functions with the outgoing edges are dependent on the files/ functions with the incoming edges. For example, by looking at Figure 2.3 we can tell that the function *A* is dependent on the functions *B* and *C*, as there is an edge from *A* to *B* and *C*. The details of these calls or dependencies can be found in the Information Browser.

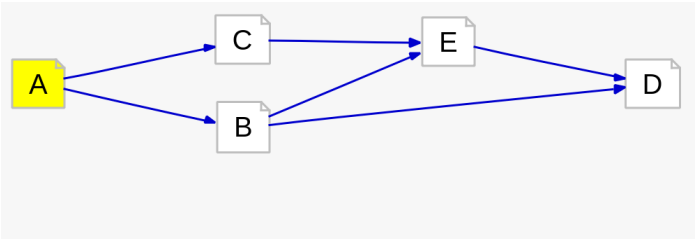


Figure 2.2: Calls relationship

- Init dependency: The init dependency focuses on the initialization of an object.

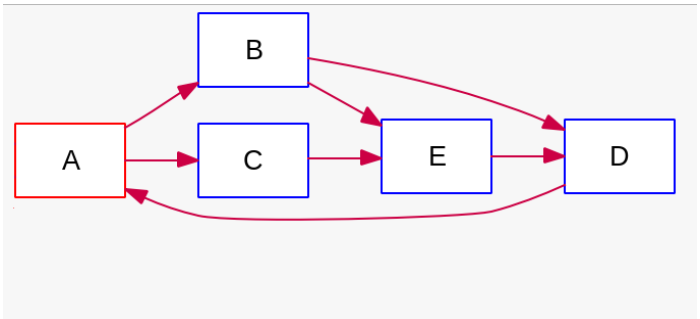


Figure 2.3: Function dependency

- Set dependency: Using *Understand* we can also find the set dependencies. Suppose a function from one file sets value of an object from different file. Then we say there exists a set dependency between these two files.

- Uses dependency: The Uses Dependency Graph shows the various uses between two files. For example by looking at the use dependency graph we can tell that how many times a file/ function uses another file/ function. Figure 2.4 shows some uses information of Function *B*. Here we can see *B* calls functions *D* and *E* and called by Function *A*. From graphical view we can see how many times it calls other functions.

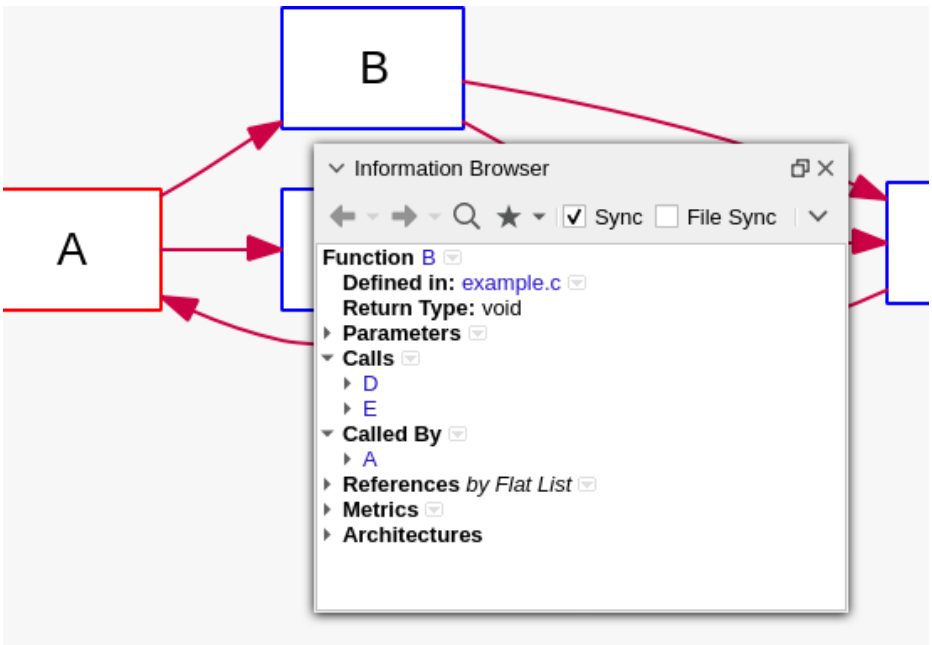


Figure 2.4: Function description

File Edit View Search Tools Documents

```
#include<stdio.h>
int x=0;

void E()
{
  printf("E");
  D();
}
void D()
{
  printf("D");
  A();
}
void C()
{
  printf("C");
  E();
}
void B()
{
  printf("B");
  D();
  E();
}
void A()
{
  if (x!=0) retun;
  x++;
  printf("A");
  B();
  C();
}
```

Figure 2.5: A pseudocode example for function dependency

2.3 Relations, Matrices and Graphs

Analysis of large complex network is common in analysis of social networks and hence their representations have become prime concern of the researchers. Besides a scientific software also has pairwise information between its units (modules) which require pairwise representation.

The field of pairwise information analysis uses three, highly related, mathematical constructs to represent them: relations, graphs and matrices.

2.3.1 Relations

A binary relation R can be defined as a set of ordered pairs (x,y). For most useful relations, the elements of the ordered pairs are naturally associated or related in some way. This relation (ordered pairs) relates the two sets together and comprises a mapping.

For example, a relation can be found in a function too. Here $y = f(x) = 2x$ is a function of all even numbers. The equation notation is just short hand for enumerating all the possible pairs in the relation, such as, $\{(1,2),(2,4),(3,6),\dots\}$.

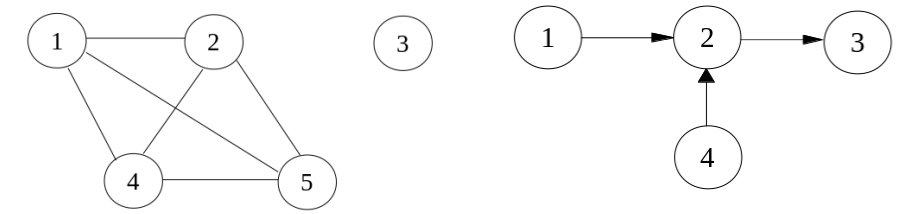
Example shown in Figure 2.5 can be represented as $R = \{(A,B),(A,C),(B,D),(B,E),(C,E),(D,A),(E,D)\}$.

2.3.2 Graphs

Let $G(V,E)$ be a graph where V is the finite set of vertices and E is the set of edges representing pairwise relationship between vertices in V. There are some categories of graphs and we use them according to our needs for analysis.

- **Directed and Undirected graphs:** Directed graphs (also called digraphs) is a graph that is made up of a set of vertices connected by edges, where the edges have a direction associated with them. This graphs consist of ordered pairs. We can use these to represent non-symmetric relations like call graphs. On the other hand, an undirected graph consists of unordered pairs where all the edges are bidirectional. They are used for the relations which are necessarily symmetric. Figure 2.6 (a) is an example of an undirected graph and Figure 2.6 (b) is an example of a directed graph.

- **Valued and Non-Valued graphs:** In valued graphs, the edges have values attached to represent characteristics of the relationships, such as strength, duration, capacity, flow, etc. For our example, we can label an edge by the number of times a function is called. Non-valued graphs do not express any value for the edges. Figure 2.7 is an example of valued graph, where each edge has a value on it. On the other hand, Figure 2.6 can be considered as examples of two non-valued graphs.



(a) (b)

Figure 2.6: An example of graph: a) Undirected Graph, b) Directed Graph

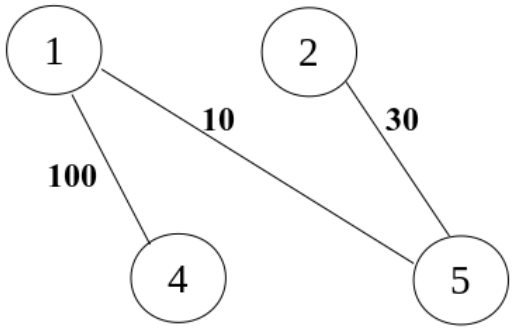


Figure 2.7: An example of a valued graph

- **Reflexive and Non-Reflexive graphs:** Reflexive graphs allow self-loops. That is, a vertex can have an edge to itself. For example, if a function calls itself (recursive function) then there will be an edge from that vertex to itself (self-loop).

- **Multi-graphs:** If there exists more than one edge between two vertices, then the graph is called a multigraph. However, instead of using multigraphs, we prefer to use valued graphs. For example, if Function *A* calls Function *B* for two times, then we place a label 2 on the edge between *A* and *B*.

Now we require some preliminary definitions.

The **degree** of a vertex is the number of vertices which are adjacent to that vertex. However, it is the number of edges that are incident upon that vertex. For example, in Figure 2.6 (a), Vertex 4 has degree 3. A zero-degree vertex is called **isolate**; Vertex 3 in

Figure 2.6 (a) is an isolate vertex. A vertex with degree 1 is called a **pendant**. Vertex 1 in Figure 2.6 (b) is such a vertex.

In a digraph (see Figure 2.6 (b)), the **indegree** of a vertex is the number of arcs (or edges) coming in to that vertex from others, while the **outdegree** is the number of arcs from that vertex to all others. In Figure 2.6 (b), Vertex 2 has indegree 2 and outdegree 1.

A graph is **connected** if there exists a path from every vertex to every other vertices. A maximal connected subgraph is called a **component**. The graph shown in Figure 2.6 (a) has two components: $\{1,2,4,5\}$ and $\{3\}$. A **maximal subgraph** is a subgraph that satisfies some specified property (such as being connected) and to which no vertex can be added without violating the property.

2.3.3 Matrices

The dependencies depicted by a graph can also be represented by a matrix of appropriate dimension. The reason of using two different ways, graph and matrix, to represent the same information is because there is a trade-off. Graphs are more intuitive than matrices but they can be difficult to understand when the number of nodes and edges grow. A few dozens nodes can be enough to produce a graph too complex. On the other hand, large and complex graphs can be very efficiently represented by a matrix.

$$X = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.1}$$

In Figure 2.8, we show a DSM considering the example shown in Figure 2.5. Here, rows correspond to the caller functions and columns correspond to the callee functions.

	A	B	C	D	E
A		1	1	0	0
B	0		0	1	1
C	0	0		0	1
D	1	0	0		0
E	0	0	0	1	

Figure 2.8: An example of a DSM

The equivalent matrix representation is presented in Equation 2.1. An entry $X_{ij} = 1$ means the function at Row_i calls the function at $Column_j$.

If we want the relation in other direction then we can simply transpose matrix X and get $X^\top$ (see Equation 2.2). When X is symmetric we have $X^\top = X$.

$$X^\top = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \tag{2.2}$$

In the beginning of this chapter, we have discussed about DSM. A DSM is a square matrix, which is used to represent the exactly same information that is in the graph or in the adjacency matrix. In this thesis, we use the duality of a graph and matrix to effectively represent and compute quantitative information about the architecture of a software system using Graph Theory and Linear Algebra.

Chapter 3

Dependency Analysis

In this thesis our objective is to find out groups of important functions (the relative importance of components in scientific software) where a group contains the most similar functions. Instead of using clustering algorithms we are using hub and authority ranking to consider functions in order and then make groups by evaluating their cosine similarity. Hub and authority ranking (using spectral methods) rely on the eigenvalues of matrix representations of networks, and capture global information on structure. In this chapter we will discuss about HITS method that we have used for our analysis.

3.1 Eigenvalues and Eigenvectors

If A is an $n \times n$ matrix, then a nonzero vector x in $\mathbb{R}^n$ is called an **eignevector** of A if Ax is a scalar multiple of x; that is,

$$Ax = \lambda x \tag{3.1}$$

for some scalar λ. The scalar λ is called **eigenvalue** of A, and x is said to be an eigenvector of A corresponding to λ.

For example, vector $x = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ is an eigenvector of $A = \begin{bmatrix} 4 & 0 \\ 12 & -2 \end{bmatrix}$ corresponding to the eigenvalue $\lambda = 4$, since

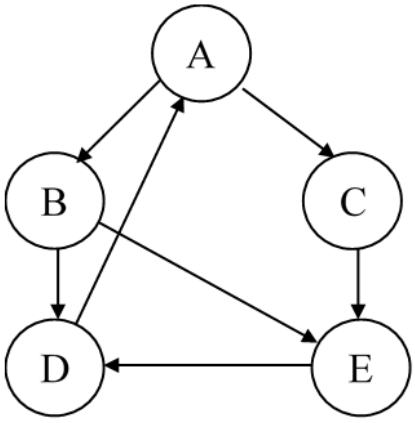
$$Ax = \begin{bmatrix} 4 & 0 \\ 12 & -2 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 4 \\ 8 \end{bmatrix} = 4x$$

To find the eigenvalues of an $n \times n$ matrix A, we rewrite Equation 3.1 as $Ax = \lambda Ix$, or as follows:

$$(\lambda I - A)x = 0 \tag{3.2}$$

Therefore, Equation 3.2 has a nonzero solution if and only if, $det(\lambda I - A) = 0$.

Here we can discuss these mathematical terms with an example. For simplicity, we recall the example stated in previous chapter.



(a)

	A	B	C	D	E
A		1	1	0	0
B	0		0	1	1
C	0	0		0	1
D	1	0	0		0
E	0	0	0	1	

(b)

Figure 3.1: An example: a) A Graph, b) An Equivalent Adjacency Matrix or DSM

From example shown in Figure 3.1, we get DSM as Matrix A as following equation:

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{3.3}$$

In Section 3.3, we shall discuss HITS algorithm from [17]. To describe that algorithm we require two special matrices $B_1 = AA^\top$ and $B_2 = A^\top A$, where $A^\top$ is transpose of Matrix

A (Matrix A is a Boolean matrix). In HITS algorithm we are computing the largest eigenvalue and the associated eigenvector of matrices B_1 and B_2. These B_1 and B_2 matrices are symmetric matrices. Symmetric matrices have real (real number) eigenvalues.

The command $[V,\lambda] = eig(B_1)$ in Octave returns diagonal matrix λ of eigenvalues and matrix V whose columns are the corresponding eigenvectors, so that $B_1 * V = V * \lambda$.

$$V = \begin{bmatrix} 0.00000 & -0.00000 & 0.00000 & 1.00000 & 0.00000 \\ 0.57735 & 0.00000 & -0.00000 & 0.00000 & 0.81650 \\ -0.57735 & 0.00000 & -0.70711 & 0.00000 & 0.40825 \\ 0.00000 & -1.00000 & 0.00000 & 0.00000 & 0.00000 \\ -0.57735 & 0.00000 & 0.70711 & 0.00000 & 0.40825 \end{bmatrix} \tag{3.4}$$

$$\lambda = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 3 \end{bmatrix} \tag{3.5}$$

Similarly, we compute the matrix of eigenvectors V associated with the eigenvalues λ of matrix B_2.

$$V = \begin{bmatrix} 0.00000 & 1.00000 & 0.00000 & -0.00000 & 0.00000 \\ -0.70711 & 0.00000 & 0.00000 & 0.70711 & 0.00000 \\ 0.70711 & 0.00000 & 0.00000 & 0.70711 & 0.00000 \\ 0.00000 & 0.00000 & -0.70711 & 0.00000 & 0.70711 \\ 0.00000 & 0.00000 & 0.70711 & 0.00000 & 0.70711 \end{bmatrix} \tag{3.6}$$

The spectrum of the DSM of call graph and the associated eigenvectors can reveal a wealth of structural information about the underlying network as we demonstrate in this thesis. The spectral ranking method that is used in this thesis (HITS) is obtained from the eigenvectors associated with selected eigenvalues of the associated DSM.

3.2 Hub and Authority

In a network, Hubs and Authorities are the two types of important nodes. A dependency graph also can be considered as a network, where functions are considered as node and their dependencies are considered as arcs. Hubs are nodes which point to many nodes of the type important, where authorities are these important nodes. For example, Figure 3.1 shows a network between five elements where a function is hub if it calls other functions and a function is authority when it is called by other functions.

From this comes a circular definition: good hubs are those which point to many good authorities and good authorities are those pointed to by many good hubs [3].

3.3 Hypertext Induced Topic Search (HITS)

Hypertext-Induced Topic Search (HITS) is an algorithm developed by Kleinberg [17], a professor in the Department of Computer Science at Cornell. This algorithm made use of the link structure of the web in order to discover and rank pages relevant for a particular

topic. The HITS ranking relies on an iterative method converging to a stationary solution. According to Kleinberg, each node in the network i is assigned two non negative weights: *authority weight* (x_i) and *hub weight* (y_i). Initially, each x_i and y_i is given an arbitrary nonnegetive value. Then the weights are updated using Equation 3.7 and Equation 3.8 for $k = 1, 2, 3, \ldots$.

$$x_i^{(k)} = \sum_{j:(j,i)\in E} y_j^{(k-1)} \tag{3.7}$$

$$y_i^{(k)} = \sum_{j:(i,j)\in E} x_j^{(k)} \tag{3.8}$$

- **Update authority weight:** Here we use Equation 3.7. In the k^{th} iteration, node i is assigned a new authority weight, $x_i^{(k)}$ which is equal to the sum of $y_j^{(k-1)}$ where the sum runs over each node j which points to node i. For all nodes in the graph we use this step, i.e. for $i = 1, 2, \ldots, n$ (n is the number of nodes in the network).

- **Update hub weight:** Equation 3.8 will be used here. The new hub weight $y_i^{(k)}$ is the sum of $x_i^{(k)}$, where the sum runs over the nodes j to which node i points. This is repeated for all nodes in the graph.

Note that the hub weights are computed from the current authority weights, where those authority weights were computed from the previous hub weights.

From the method described above, we observe the natural dependency relationship between hubs and authorities. y-value (hub) of a node is large, if the node points to many nodes with large x-values (authorities) and vice versa [18].

We need to normalize all the hub and authority values for all nodes after each iteration so that $\sqrt{\sum_{i=1}^{n} x_i^2} = \sqrt{\sum_{i=1}^{n} y_i^2} = 1$.

Now, in iteration k for n nodes we can represent hub and authority values in terms of vectors. If $\vec{x_k}$ represents the vector of authority values and $\vec{y_k}$ represents the vector of hub values in iteration k, then for n nodes we have

$$\vec{x_k} = \begin{bmatrix} x_k(1) \\ x_k(2) \\ \vdots \\ x_k(n) \end{bmatrix} \tag{3.9}$$

and

$$\vec{y_k} = \begin{bmatrix} y_k(1) \\ y_k(2) \\ \vdots \\ y_k(n) \end{bmatrix} \tag{3.10}$$

If $k = 0$, then using Equations 3.9 and 3.10 we can initialize $\vec{x_0}$ and $\vec{y_0}$, as $x_0(1) = x_0(2) = \cdots = x_0(n) = (1/\sqrt{n})$ and $y_0(1) = y_0(2) = \cdots = y_0(n) = (1/\sqrt{n})$.

Let A be an adjacency matrix of the directed graph G. Then using Equation 3.11 and 3.12, we can represent the algorithm stated above.

$$\vec{x_k} = c_k A^\top \vec{y}_{(k-1)} \tag{3.11}$$

$$\vec{y_k} = c'_k A \vec{x_k} \tag{3.12}$$

c_k and c'_k from Equation 3.11 and Equation 3.12 respectively are the normalization constants. In iteration k, these are chosen in a way that the sum of the squares of the authority weights, as well as that of the hub weights are equal to 1. Considering these equations we can now represent HITS method using following equations [17]:

$$\vec{x}_k = c_k c'_{(k-1)} A^\top A \vec{x}_{(k-1)} \quad \text{for } k > 1 \tag{3.13}$$

$$\vec{y}_k = c'_k c_k A A^\top \vec{y}_{(k-1)} \quad \text{for } k > 0 \tag{3.14}$$

Therefore we can say that HITS is an iterative power method to compute the dominant eigenvector for $AA^\top$ and $A^\top A$ [10]. Dominant eigenvector is the column of matrix V (corresponding eigenvector) which corresponds to the eigenvalue.

The hub scores and the authority scores are determined by the entries of the dominant eigenvector of $AA^\top$ and $A^\top A$ respectively [3].

Again, we can consider Figure 3.1 as an example to explain the HITS algorithm, where the adjacency matrix of that graph is given in Equation 3.3.

Table 3.1: Hub and Authority Ranking (Scores correspond to the dominant eigenvector)

Node	Hub Score	Hub Rank	Authority Score	Authority Rank
A	0.00000	3	0.00000	2
B	0.81650	1	0.00000	2
C	0.40825	2	0.00000	2
D	0.00000	3	0.70711	1
E	0.40825	2	0.70711	1

The eigenvectors of $AA^\top$ and $A^\top A$ corresponding to the largest eigenvalue $\lambda_{max} = 3$ (see Equation 3.5), yield the ranking for hubs and authorities (using HITS algorithm) as shown in Table 3.1. We observe in Equation 3.5 that the dominant eigenvalue is in column 5 and hence the 5^{th} column of V in Equations 3.4 and 3.6 are the corresponding dominant eigenvectors. We consider these as scores for calculating hub and authority ranking respectively. Here the ranking of nodes A to E for hubs is $\{3,1,2,3,2\}$ and the ranking for authorities is $\{2,2,2,1,1\}$.

Chapter 4

Methodology and Results

Many software are usually written by domain experts and address some specific problem. Most of the time these software do not contain adequate technical documentation for different reasons. So from a usability point of view it may be difficult to detect and retrieve components that could be reused in other software projects. Analyzing such software are more challenging for researchers.

In this thesis we analyze the dependencies between functions depicted by static call graphs to categorize functions into groups of "similar functions" (by using hidden dependencies as described below) according to their "importance" (by spectral ranking the functions using HITS algorithm) in the software system. The hub functions that are categorized "most important" are the functions that provide core services to its end users. The associated authority functions represent most important "service providers" to the hub functions.

A Hub function directly partakes in the implementation of core functionality of the software. Examples of Hub functions in CSparse are *cs_lusol* (solves a linear system with unsymmetric coefficient matrix) and *cs_lsolve* (solves a lower triangular linear system). On the other hand, functions that are tasked with providing support services to the software system are termed Authority functions. Examples of Authority function in CSparse are *cs_realloc* (changes size of a block of memory) and *cs_done* (frees workspace and returns a sparse matrix).

In a call graph if function i calls function j then we say that i depends on j. This type of dependency is explicit in that it can be extracted directly from the source code. Suppose,

functions i and j both call function k. Intuitively, it means that functions i and j are related in some way (depending on the context). This is an example of "hidden" dependency which is not discernible from the call graph. We compute this kind of hidden dependencies from the product matrix $B = A * A^{\top}$. A nonzero value of B(i,j) implies a "hidden dependency" between functions i and j.

In this work we analyze the dependency structures of software (caller-callee relation between functions) using tool "Understand" [21], and uncover "similarity" among design elements (functions) using a suitable similarity metric (cosine similarity). Then combined with the notion of "importance" of design elements [13], we group or cluster the design elements into "tiers" ranked by their "importance".

This chapter will discuss the impact of function dependencies on the software architecture as well as the impact of uncovering "similarity" among design elements. Methodology followed by experimental results of our novel work are also discussed in this chapter.

4.1 Methodology

This section describes the methodology we use in our work. We start this section by describing the method of dependency extraction followed by building DSM. Using the analytics tool "Understand" [21] we can view and extract dependency structure of the software. We extract caller-callee relation between functions to analyze the dependency structures of programs. The method of finding "importance" of design elements [13] computes hub and authority rankings which provide lists of important caller (hubs) and important callee (authorities). Then we discuss the method of uncovering "similarity" among design elements. Our goal is to group or cluster the design elements into "tiers" ranked by their "importance" which is presented using a pseudocode.

Computational Infrastructure for Operation Research [8] (COIN-OR) is one of the largest and most widely studied open source communities for scientific research software. We studied open source software projects from COIN-OR. For example, CSparse and ADOL-C

software implemented in C, will be described briefly in sections 4.2.1 and 4.2.1 respectively. Then results for these software that we got from our experiments will be discussed in Section 4.3. In the following sections we briefly describe our novel approach considering these software.

4.1.1 Extracting Dependencies

In Section 2.2, we have given some preliminaries regarding extracting dependencies.

In this section, we discuss how we extract the call dependencies between functions (for both CSparse and ADOL-C) using the "Understand" SciTool [21]. We have already discussed this tool in previous section. We can visualize these software (CSparse and ADOL-C) using Octave. Figure 4.1 and Figure 4.2 show the matrices of CSparse and ADOL-C respectively plotted using Octave.

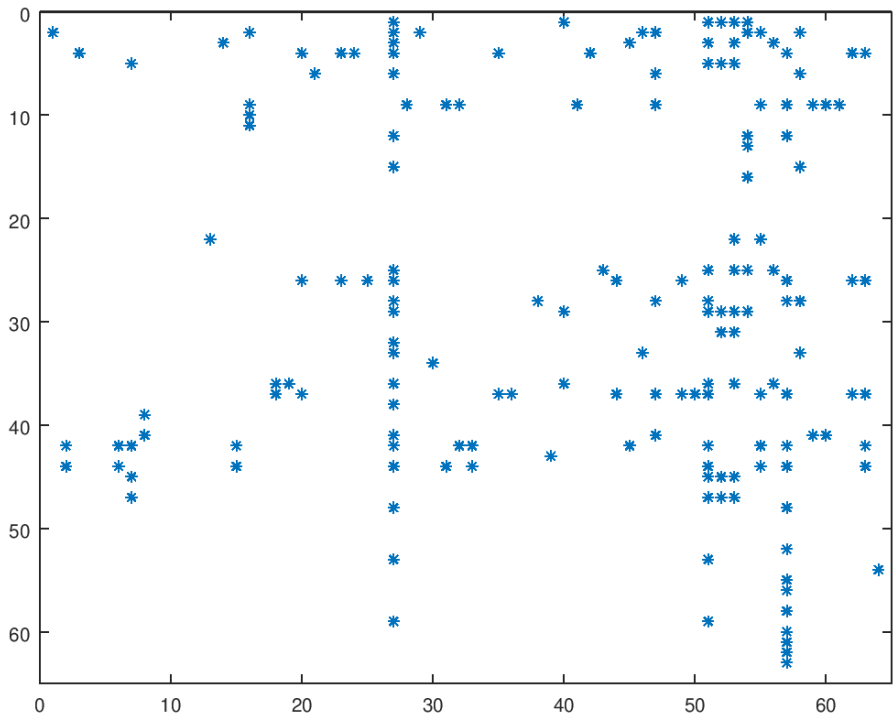


Figure 4.1: Sparse Matrix of CSparse

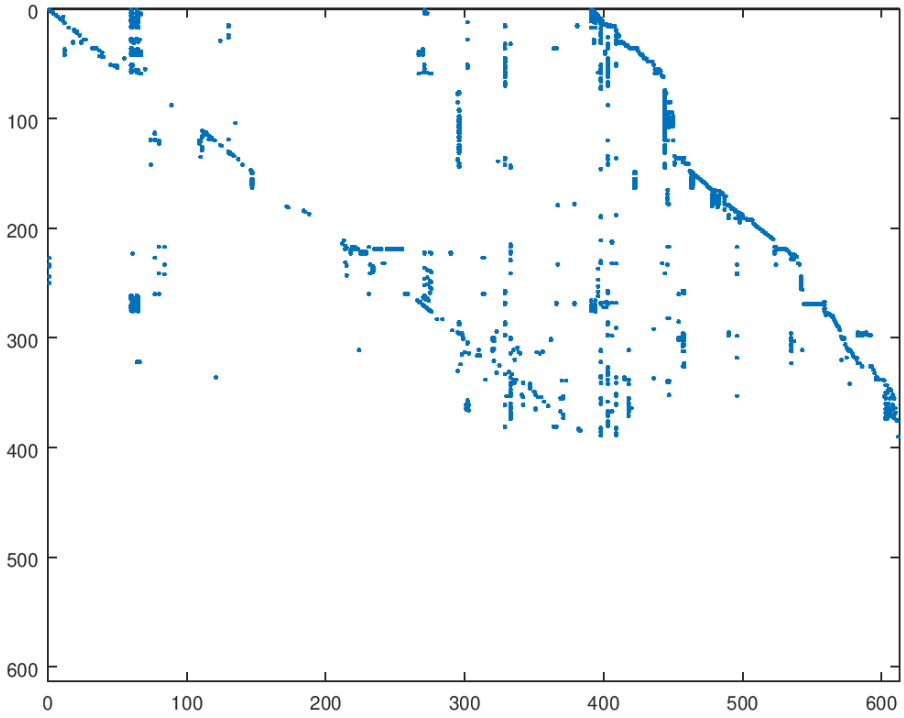


Figure 4.2: Sparse Matrix of ADOL-C

For extracting the dependencies, first we download the full package (latest release) of the system software (CSparse and ADOL-C). Then we import the package in "Understand" SciTool. Then we can visualize call graphs (see Figure 4.3 and Figure 4.4). Then we inspect each function and observe its detail from the description option. In this description, we can find the list of functions which are called by that particular function and also a lot of information. We can also export dependency matrices in different file formats.

4.1.2 Building DSMs

We create DSM with static dependencies. DSM is a visual representation of a system and in the form of a square matrix. We have investigated CSparse project with 64 functions and ADOL-C project with 612 functions. Therefore, CSparse has a DSM with $64X64$ matrix and $612X612$ DSM matrix for ADOL-C. These matrices are binary because we did not consider weighted matrices. It means, even if a function i calls another function j for

Figure 4.3: Dependency graph of CSparse

more than once, we assume $DSM[i, j] = 1$.

Figure 4.5 shows the DSM for CSparse. The order of the functions (in rows or columns) in this $64X64$ binary matrix can be found from Appendix A. We see that, $DSM[2, 1] = 1$, which means, Function *cs_amd* calls Function *cs_add*. The blank cells are considered as 0. We have similar DSM ($612X612$ binary matrix) for ADOL-C too.

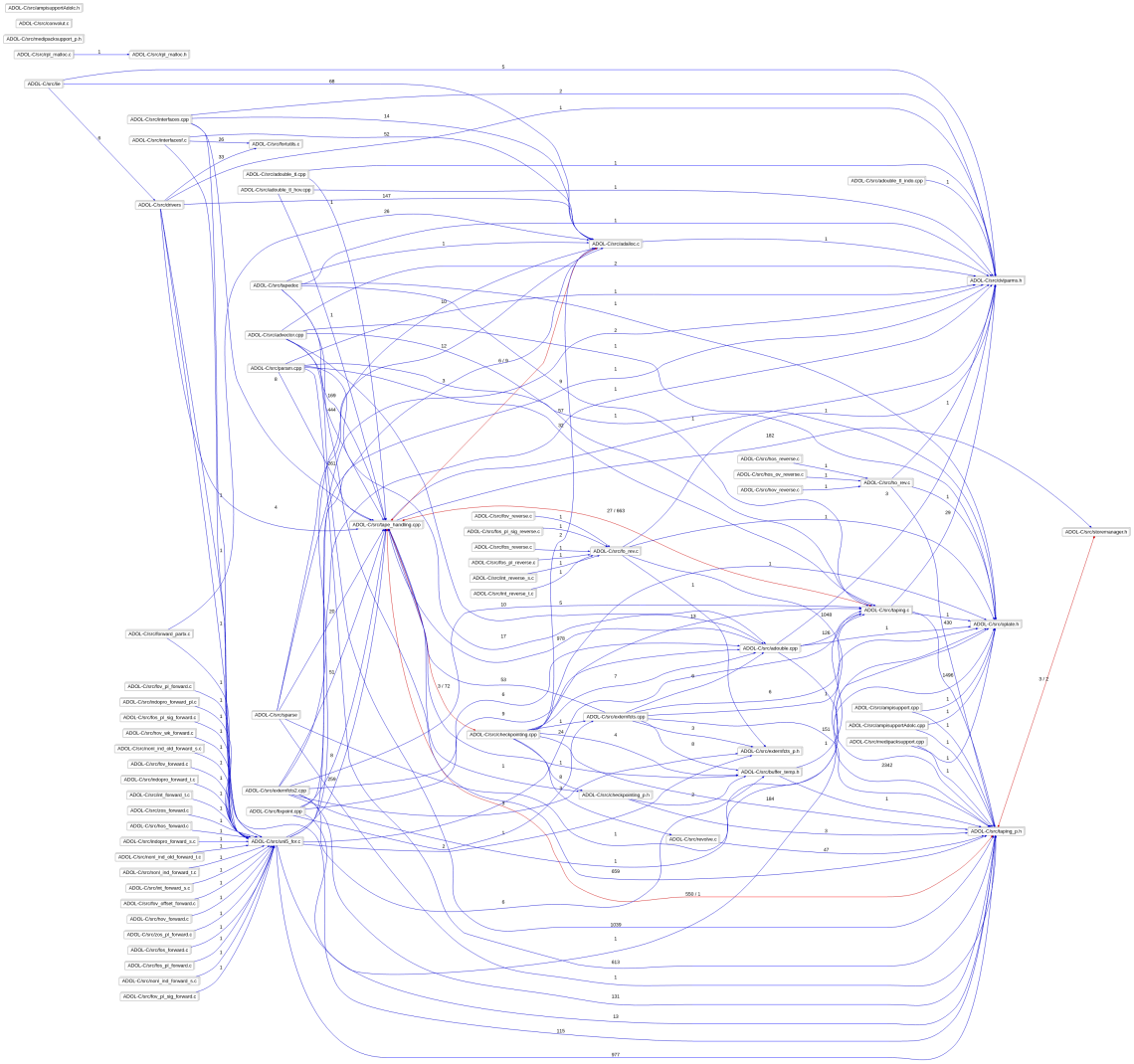


Figure 4.4: Dependency Graph of ADOL-C

4.1.3 Computing Hub and Authority Ranking

In Chapter 3, using a small example we have discussed some methods of analysing dependency between functions. For example, calculating eigenvalue and eigenvector from given matrix, identifying hubs and authorities and use of HITS method to rank functions. In this section, we describe the importance and the method of computing hub and authority ranking for our approach.

For a given Matrix A, HITS algorithm computes two matrices $B_1 = AA^\top$ and $B_2 = A^\top A$. Matrices B_1 and B_2 have important role to compute hub ranking and authority ranking

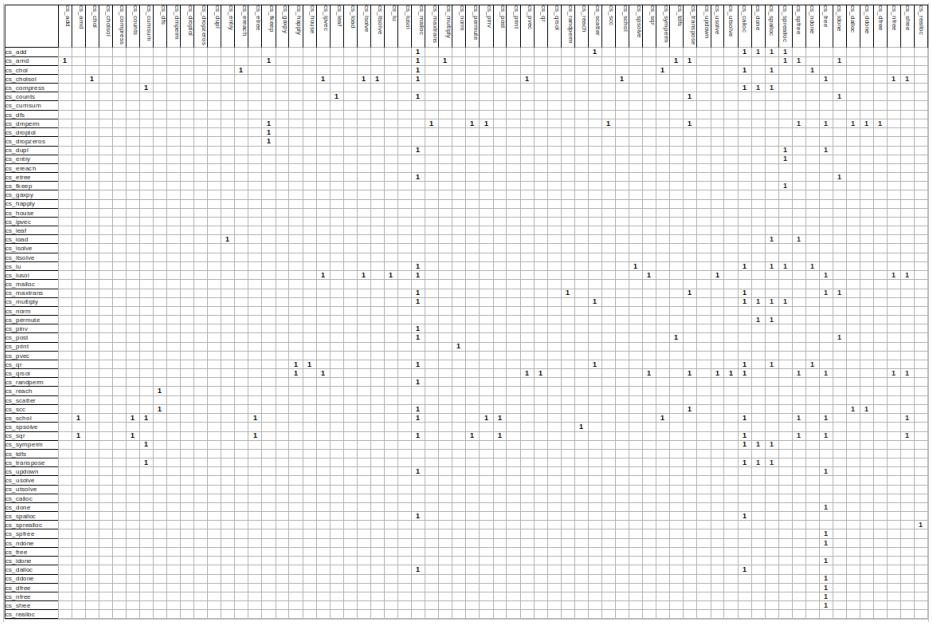


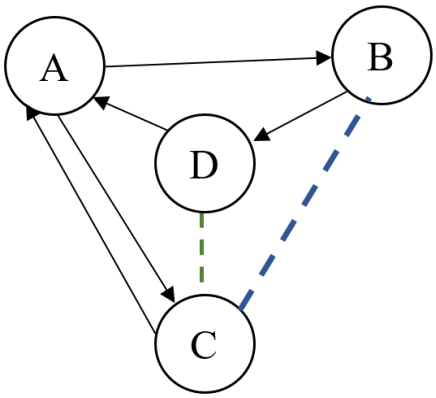
Figure 4.5: DSM of CSparse

respectively. Figure 4.6 shows an example of a call graph for which we get,

$$B_1 = AA^\top = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \mathbf{1} \\ 0 & 0 & \mathbf{1} & 1 \end{bmatrix} \tag{4.1}$$

$$B_2 = A^\top A = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & \mathbf{1} & 0 \\ 0 & \mathbf{1} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

Therefore, diagonal value of Matrix B_1, $diag(B_1) = [2,1,1,1]$ represents outdegree of the functions and diagonal value of Matrix B_2, $diag(B_2) = [2,1,1,1]$ represents indegree of



	A	B	C	D
A	0	1	1	0
B	0	0	0	1
C	1	0	0	0
D	1	0	0	0

Figure 4.6: An Example of Hub and Authority

the functions. Besides, in Equation 4.1 and 4.2, we see some values outside the diagonal (reported as bold text). Equation 4.1 tells that Function C and Function D have an indirect relationship though both of them call Function A (Green dot line in Figure 4.6). On the other hand, Equation 4.2 tells that Function B and Function C have an indirect relationship though both of them are called by Function A (Blue dot line in Figure 4.6).

So computing matrices B_1 and B_2 gives semantic dependency relationship where DSM gives only the syntactic dependency relationship between functions.

Suppose we have build our DSM from the given source code which is A. Now compute matrices $B_1 = AA^\top$ and $B_2 = A^\top A$, where $A^\top$ is transpose of Matrix A.

Then using Octave, we can find diagonal matrix λ of eigenvalues and matrix V whose columns are the corresponding eigenvectors, so that $B_i * V = V * \lambda$, where B_i is B_1 and B_2. We look for the dominant eigenvalue of λ which holds the highest magnitude of λ. The corresponding column of V is the dominant eigenvector which is considered as the hub value (for B_1) and authority value (for B_2) [13]. We sort the functions in descending order according to their values (see Section 3.1) and hence get the authority ranking and hub ranking of the functions.

In tables 4.1 and 4.2 we have reported 5 topmost authorities and hubs according to their values for both CSparse and ADOL-C software respectively.

Table 4.1: Hub and Authority rank of first five functions of CSparse project

Rank	Hub	Hub Name	Authority	Authority Name
1	42	*cs_schol*	27	*cs_malloc*
2	44	*cs_sqr*	51	*cs_calloc*
3	37	*cs_qrsol*	57	*cs_free*
4	28	*cs_maxtrans*	53	*cs_spalloc*
5	29	*cs_multiply*	63	*cs_sfree*

Table 4.2: Hub and Authority rank of first five functions of ADOL-C project

Rank	Hub	Hub Name	Authority	Authority Name
1	36	*tape_doc*	60	*myalloc*1
2	4	*jacobian*	403	*fprintf*
3	58	*forward*	64	*myfree*1
4	29	*inverse_Taylor_prop*	329	*adolc_exit*
5	9	*hessian*	61	*myalloc*2

4.1.4 Computing Cosine Similarities

Cosine similarity is a metric used to determine how similar the functions are. Mathematically, it measures the cosine of the angle between two vectors projected in a multi-dimensional space. In this context, the two vectors (two rows of DSM) are matrices containing the call information of two functions. When plotted on a multi-dimensional space, where each dimension corresponds to a function in the system, the cosine similarity captures the orientation (the angle) of the functions and not the magnitude. The cosine similarity is advantageous because even if the two similar functions are far apart by rankings they could still have a smaller angle between them. The smaller the angle, the higher the similarity.

Equation 4.3 gives the $\cos(\theta)$ (cosine similarity) between vectors $\vec{a}$ and $\vec{b}$.

$$\cos(\theta) = \frac{\vec{a}\vec{b}}{\|\vec{a}\|\|\vec{b}\|} \tag{4.3}$$

where $\vec{a}$ and $\vec{b}$ are vectors of the same size, $\|\vec{a}\|$ and $\|\vec{b}\|$ are the Euclidean norm of these

vectors, and n is the size of these vectors and the number of elements in the system.

In our work, DSM contains the call information between functions, where rows are caller functions (hub) and columns are callee functions (authority). When we compute cosine similarity between two caller functions, we select two corresponding rows from the DSM as vectors $\vec{a}$ and $\vec{b}$. Again, when we compute cosine similarity between two callee functions, we select two corresponding columns from the DSM as vectors $\vec{a}$ and $\vec{b}$.

In our approach, we consider DSM as well as $B_1 = AA^\top$ (hub) and $B_2 = A^\top A$ (authority) matrices to choose the vectors for similarity check. The importance of matrices B_1 and B_2 are discussed in Section 4.1.3, where we have shown that matrices B_1 and B_2 contain more important information than DSM. Therefore, we tested our approach for both type of matrices.

From Equation 4.3, we observe that to calculate the dot product between two vectors ($\sum_{i=1}^{n} \vec{a}_i \vec{b}_i$), it calculate sum of the product of corresponding entries of two selected rows or columns.

We know, $\cos(0^\circ) = 1$ and $\cos(90^\circ) = 0$. Therefore, if two vectors are orthogonal (not similar) than the value of $\cos(\theta)$ will be 0 and if two vectors are parallel (similar) then the value of $\cos(\theta)$ will be 1. But if the value of $\cos(\theta)$ is in between 0 and 1 then we can set a *threshold* to identify the similarity. In this thesis, we have tested our algorithm for different *threshold* value between 0 and 1. For example, 0.0, 0.1, 0.2, ..., 0.9, 1.0.

Besides, we have calculated similarities between functions for hubs and authorities. For hub, we choose two row vectors from the matrix and for authorities we choose two column vectors from the matrix.

Now we compute the cosine similarities between first five functions (according to their rank, see Table 4.1 and Table 4.2) considering both hubs and authorities (for projects CSparse and ADOL-C).

Table 4.3 and Table 4.4 report cosine similarities between functions, where $\vec{a}$ and $\vec{b}$ represents functions of the projects (see Appendix A and B). Here we get the value of

Table 4.3: Cosine Similarities of First Five Functions (considering Hub and Authority Rank) of CSparse Project

Hub Functions			Authority Functions		
$\vec{a}$	$\vec{b}$	$\cos(\theta)$	$\vec{a}$	$\vec{b}$	$\cos(\theta)$
cs_schol	*cs_sqr*	0.982167	*cs_calloc*	*cs_spalloc*	0.873418
cs_sqr	*cs_maxtrans*	0.798325	*cs_free*	*cs_sfree*	0.861267
cs_schol	*cs_maxtrans*	0.776425	*cs_malloc*	*cs_calloc*	0.848427
cs_sqr	*cs_qrsol*	0.722491	*cs_malloc*	*cs_sfree*	0.678954
cs_schol	*cs_qrsol*	0.696213	*cs_malloc*	*cs_free*	0.674013
cs_maxtrans	*cs_multiply*	0.69317	*cs_malloc*	*cs_spalloc*	0.636215
cs_qrsol	*cs_maxtrans*	0.691092	*cs_calloc*	*cs_sfree*	0.619324
cs_schol	*cs_multiply*	0.590327	*cs_calloc*	*cs_free*	0.557076
cs_sqr	*cs_multiply*	0.577527	*cs_spalloc*	*cs_sfree*	0.262881
cs_qrsol	*cs_multiply*	0.400871	*cs_free*	*cs_spalloc*	0.220262

Table 4.4: Cosine Similarities of First Five Functions (considering Hub and Authority Rank) of ADOL-C Project

Hub Functions			Authority Functions		
$\vec{a}$	$\vec{b}$	$\cos(\theta)$	$\vec{a}$	$\vec{b}$	$\cos(\theta)$
jacobian	*hessian*	0.953637	*myalloc*1	*myfree*1	0.970463
inverse_Taylor_prop	*hessian*	0.89385	*fprintf*	*adolc_exit*	0.944289
jacobian	*inverse_Taylor_prop*	0.822776	*myfree*1	*myalloc*2	0.879903
tape_doc	*forward*	0.68348	*myalloc*1	*myalloc*2	0.865987
jacobian	*forward*	0.529026	*myalloc*1	*adolc_exit*	0.359106
forward	*hessian*	0.504676	*myalloc*1	*fprintf*	0.317976
forward	*inverse_Taylor_prop*	0.407995	*adolc_exit*	*myalloc*2	0.243542
tape_doc	*hessian*	0.315302	*myfree*1	*adolc_exit*	0.221187
tape_doc	*jacobian*	0.295122	*fprintf*	*myalloc*2	0.211266
tape_doc	*inverse_Taylor_prop*	0.234508	*fprintf*	*myfree*1	0.190357

$\cos(\theta)$ between 0 and 1. Therefore, by fixing a threshold we can say whether two functions are similar or not.

4.1.5 The Algorithm

The complete algorithm of our approach to find out similar functions in different tiers is presented below.

Algorithm 1: Group_Similar_Functions (DSM A)

$N \leftarrow$ Number of functions
$threshold \leftarrow$ a numeric value between 0 and 1
$k \leftarrow 0$ ▷ Number of tiers
while $N > 0$ **do**
  $h \leftarrow$ List of top 5 elements in hub ranking order
  $a \leftarrow$ List of top 5 elements in authority ranking order
  $U \leftarrow [\,]$ ▷ U is the list of elements to be removed from A after each iteration
  for $i \leftarrow 1$ to 5 **do**
    $hub\_Similarity[i] \leftarrow 0$ ▷ Store similarity between hub elements from h
    $aut\_Similarity[i] \leftarrow 0$ ▷ Store similarity between authority elements from a
  for $i \leftarrow 1$ to 4 **do**
    for $j \leftarrow i+1$ to 5 **do**
      $hub\_Similarity[i] \leftarrow hub\_Similarity[i] + cosineSimilarity(h[i], h[j])$
      ▷ $cosineSimilarity()$ is a function as Equation 4.3
      $hub\_Similarity[j] \leftarrow hub\_Similarity[j] + cosineSimilarity(h[i], h[j])$
  for $i \leftarrow 1$ to 5 **do**
    $hub\_Similarity[i] \leftarrow hub\_Similarity[i]/4$ ▷ Calculating average similarity
  $k \leftarrow k+1$
  for $i \leftarrow 1$ to 5 **do**
    if $hub\_Similarity[i] >= threshold$ **then**
      Include $h[i]$ in T_k
      $U \leftarrow U \cup h[i]$
  for $i \leftarrow 1$ to 4 **do**
    for $j \leftarrow i+1$ to 5 **do**
      $aut\_Similarity[i] \leftarrow aut\_Similarity[i] + cosineSimilarity(a[i], a[j])$
      $aut\_Similarity[j] \leftarrow aut\_Similarity[j] + cosineSimilarity(a[i], a[j])$
  for $i \leftarrow 1$ to 5 **do**
    $aut\_Similarity[i] \leftarrow aut\_Similarity[i]/4$ ▷ Calculating average similarity
  $k \leftarrow k+1$
  for $i \leftarrow 1$ to 5 **do**
    if $aut\_Similarity[i] >= threshold$ **then**
      Include $a[i]$ in T_k
      $U \leftarrow U \cup a[i]$
  Remove all $i \in U$ from A
  $N \leftarrow N - |U|$
return $T_1, T_2, ..., T_k$ ▷ T_i is a tier containing similar functions

In this algorithm, DSM A is the input and a list of tiers having similar functions are the output $(T_1, T_2, ..., T_k)$. $threshold$ is a predefined numeric value between 0 and 1. In Section 4.1.3, we discussed how do we compute hub and authority ranking of the functions.

Following the same method, in Step 5, we get a list (h) of top 5 functions according to hub ranking and in Step 6, we get a list (a) of top 5 functions according to authority ranking.

Now we describe how do we compute hub tiers. In steps 11 to 15 we calculate cosine similarity between these 5 hub functions as described in Section 4.1.4. Then we compute the average similarity of each hub functions in Step 17. In Step 20, the algorithm checks whether the average similarity of function $h[i]$ (i.e. $hub\_Similarity[i]$) is greater or equal to the predefined *threshold* or not. If it satisfies the condition, then the function is included in tier T_k, and also stored in U to be removed from Matrix A after the current iteration; otherwise, it does nothing to that function. Similarly, for authority tiers, we followed steps 23 to 33. Therefore, in one iteration of the *while* loop, we compute two tiers, one for the hub and one for the authority. The algorithm continues until there is no functions or no such tier can be computed with our set conditions.

4.2 Settings

This section brings details about the study settings we use in our work. Here we discuss our target systems and selection of thresholds for our experiments.

4.2.1 Target Systems

For our experiment, we select two software implemented in `C/C++`, *CSparse* version 5.6.0 and *ADOL-C* version 2.7.2. These scientific software are used to compute accurate (upto machine precision) numerical derivatives of the function program at a specified point.

CSparse software is concerned with solving system of linear equation $Ax = b$ where the coefficient matrix A is sparse [6].

On the other hand, *ADOL-C* is a software system to compute mathematical derivatives (gradient, Jacobian, Hessian, Taylor coefficients) of a mathematical function [12].

CSparse

CSparse is a project which contains direct methods for sparse linear systems. There are many problems in computational field which deals with solution of sparse systems of linear equations. To solve these problems efficiently, we require an in-depth knowledge of the underlying theory, algorithms, and data structures found in sparse matrix software libraries. *CSparse* presents the fundamentals of sparse matrix algorithms to provide the requisite background [6]. This project is downloadable sparse matrix package that illustrates the algorithms and theorems presented in [6]. To work with this project user must have some knowledge on larger and more complex software packages and also a strong idea on MATLAB and the C programming language. To understand more about this project (Sparse Linear Systems), we suggest to get idea from [6].

The functions are categorized by the author of the software as: primary, primary utility, secondary, secondary utility, tertiary and tertiary utility [6]. In this thesis, we have analyzed 64 (`C`) functions from *CSparse* package (see Appendix A).

ADOL-C

ADOL-C software is implemented in `C/C++` [12]. This package facilitates the evaluation of first and higher derivatives of vector functions written in `C/C++`. Using `C`, `C++`, `Fortran`, or any other language that can be linked with `C`, anyone can use all routines in this package.

Error free numerical values of derivative vectors can be calculated with an efficient running time and small space by the given function evaluation program. Derivative matrices are obtained by columns, by rows or in sparse format. For solution curves defined by ordinary differential equations, special routines are provided that evaluate the Taylor coefficient vectors and their Jacobians with respect to the current state vector. For explicitly or implicitly defined functions derivative tensors are obtained with a complexity that grows only quadratically in their degree. The derivative calculations involve a possibly substantial but always predictable amount of data. Sequentially this data is accessed and hence it can be

automatically paged out to external files.

In our thesis, we have analyzed 612 functions (implemented in `C`) from *ADOL-C* package (see Appendix B).

4.2.2 Selection of the Threshold

In Algorithm 1, we have talked about a *threshold* (Step 2 in Algorithm 1), which helps to decide whether a function should be included in the tier or not (steps 20 and 31). The value of *threshold* should be chosen in between 0 and 1, because we know that the value of $\cos(\theta)$ varies from 0 to 1. In this thesis, the value of cosine similarity ($\cos(\theta)$) cannot be negative because the vectors are positive. Since the vectors are selected from dependency matrix (dependency cannot have negative value), the vectors will always lie in the first quadrant. Two functions are more similar if their $\cos(\theta)$ value is close to 1. (Note: If two vectors are similar or parallel it means their angle is 0°, hence $\cos(0^\circ) = 1$). On the other hand, if $\cos(\theta)$ value between two vectors is close to 0, then they are more dissimilar.

4.3 Results

In this section, we provide results from numerical experiments on selected projects. The software for the experiments is obtained from Computational Infrastructure for Operation Research (COIN-OR) [8]. The experiments were performed using a PC with 3.4 GHz Intel Xeon CPU, 8 GB RAM running Linux. The implementation language was GNU Octave and the code was compiled with version 4.2.2 compiler.

Test results for the selected test package *CSparse* are reported in figures 4.8, 4.9, 4.10 and 4.11. Here, figures 4.8 and 4.9 show results (functions of different tiers) where we considered DSM to choose functions for their similarity check. On the other hand, tables 4.10 and 4.11 show results where we considered $AA^\top$ and $A^\top A$ (for hub and authority respectively) to choose functions for their similarity check. In the following section we discuss these results to validate our approach.

Similarly, test results for the selected software *ADOL-C* are reported in figures 4.12, 4.13, 4.14 and 4.15. Here, in all cases we considered matrices $AA^\top$ and $A^\top A$ (for hub and authority respectively) to choose functions for their similarity check. We reported results separately for $threshold = 0.4$ in figures 4.14 and 4.15.

Now we discuss the results for software *ADOL-C*. Figures 4.12 and 4.14 show that for different *threshold* values we get different number of tiers for hub functions. Again, figures 4.13 and 4.15 show that for different *threshold* values we get different number of tiers for authority functions.

4.3.1 Discussion

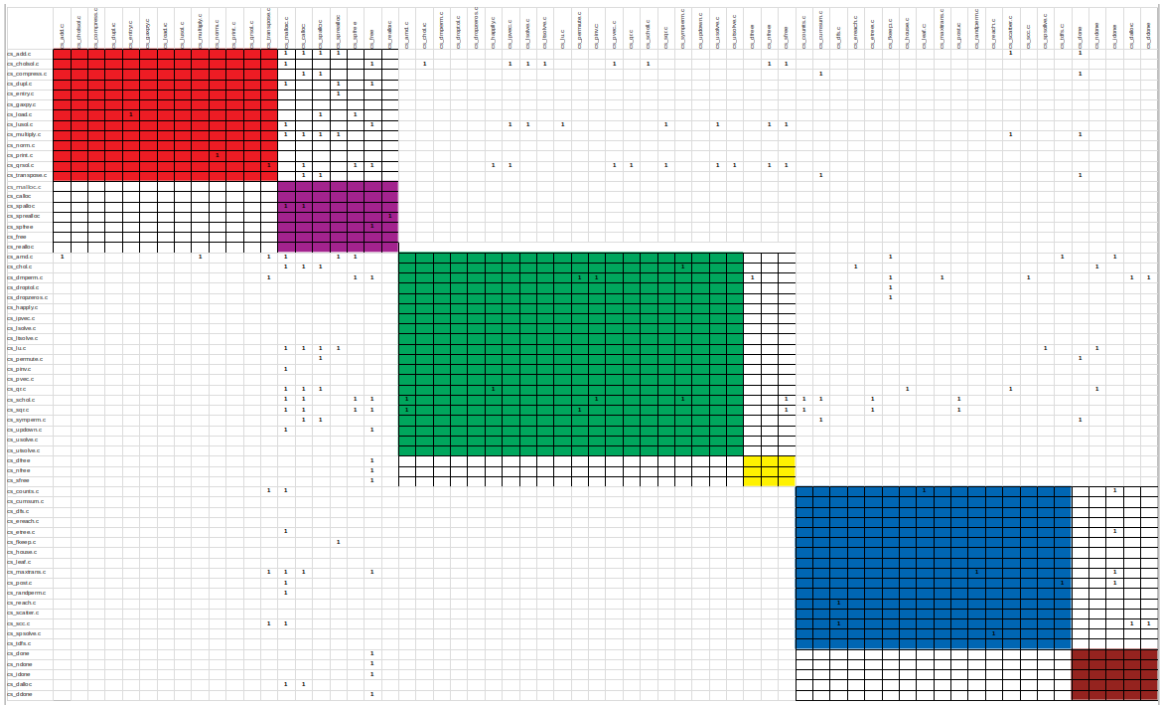


Figure 4.7: DSM of CSparse with provided partitions [6]. Primary, Primary Utility, Secondary, Secondary Utility, Tertiary and Tertiary utility are marked by red, purple, green, yellow, blue and brown colors respectively.

Figure 4.7 is the DSM which represents the partition provided in [6]. Here, the partitions are primary (red color), primary utility (purple color), secondary (green color), secondary utility (yellow color), tertiary (blue color) and tertiary utility (brown color). We see that

Hub tiers	Threshold			
	0.3		0.4	
	Name	Category	Name	Category
1	cs_schol cs_sqr cs_maxtrans	Secondary Secondary Tertiary	cs_schol cs_sqr	Secondary Secondary
2	cs_qrsol cs_cholsol cs_lusol	Primary Primary Primary	cs_qrsol	Primary
3	cs_multiply cs_add cs_lu cs_qr cs_symperm	Primary Primary Secondary Secondary Secondary	cs_multiply cs_add cs_transpose cs_symperm cs_compress	Primary Primary Primary Secondary Primary
4	cs_dmperm cs_amd cs_dropzeros cs_droptol	Secondary Secondary Secondary Secondary		
5	cs_post cs_counts cs_etree	Tertiary Tertiary Tertiary		

Figure 4.8: CSparse, Tiers of Hub Functions functions are selected from DSM

Authority tiers	Threshold			
	0.3		0.4	
	Name	Category	Name	Category
1	cs_malloc cs_calloc cs_sfree	Primary_Util Primary_Util Secondary_Util	cs_malloc cs_calloc	Primary_Util Primary_Util
2	cs_free cs_transpose cs_spfree cs_nfree cs_ipvec	Primary_Util Primary Primary_Util Secondary_Util Secondary	cs_free cs_nfree cs_ipvec cs_sfree	Primary_Util Secondary_Util Secondary Secondary_Util
3	cs_spalloc cs_done cs_sprealloc cs_scatter	Primary_Util Tertiary_Util Primary_Util Tertiary	cs_spalloc cs_done	Primary_Util Tertiary_Util
4	cs_fkeep cs_ddone cs_dalloc cs_scc cs_pinv	Tertiary Tertiary_Util Tertiary_Util Tertiary Secondary	cs_spfree cs_ddone cs_dalloc	Primary_Util Tertiary_Util Tertiary_Util

Figure 4.9: CSparse, Tiers of Authority Functions functions are selected from DSM

the primary, secondary and tertiary functions mostly call other functions. According to HITS method [17], these functions are important hubs. On the other hand, we observe that utilities are mostly called by other functions and according to HITS method these can be considered as authorities.

Figures 4.8 and 4.10 show that for different *threshold* values we get different number of tiers for hub functions. These tiers also match with the category given in *CSparse*. For example, in Figure 4.8, Hub Tier 5 contains three functions: *cs_post*, *cs_counts* and *cs_etree* which are categorized as tertiary functions in [6] (see Figure 4.7).

Similarly, figures 4.9 and 4.11 show that for different *threshold* values we get different number of tiers for authority functions. These tiers also match with the category given in *CSparse*. For example, in Figure 4.9, Authority Tier 1 contains three functions: *cs_malloc*, *cs_calloc* and *cs_sfree* which are categorized as utility functions in [6] (see Figure 4.7).

Hence our method produces a good approximation of author's partition.

Hub tiers	Threshold									
	0.3		0.4		0.5		0.6		0.7	
	Name	Category	Name	Category	Name	Category	Name	Category	Name	Category
1	cs_schol.c cs_sqr.c cs_qrsol.c cs_maxtrans.c cs_multiply.c	Secondary Secondary Primary Tertiary Primary	cs_schol cs_sqr cs_qrsol cs_maxtrans cs_multiply	Secondary Secondary Primary Tertiary Primary	cs_schol cs_sqr cs_qrsol cs_maxtrans cs_multiply	Secondary Secondary Primary Tertiary Primary	cs_schol cs_sqr cs_qrsol cs_maxtrans	Secondary Secondary Primary Tertiary	cs_schol cs_sqr cs_maxtrans	Secondary Secondary Tertiary
2	cs_dmperm.c cs_amd.c cs_scc.c cs_counts.c cs_post.c	Secondary Secondary Tertiary Tertiary Tertiary	cs_dmperm cs_amd cs_scc cs_counts cs_post	Secondary Secondary Tertiary Tertiary Tertiary	cs_add cs_transpose cs_compress cs_symperm cs_lu	Primary Primary Primary Secondary Secondary	cs_multiply cs_add cs_transpose cs_symperm cs_compress	Primary Primary Primary Secondary Primary		
3	cs_transpose.c cs_compress.c cs_symperm.c cs_add.c cs_permute.c	Primary Primary Secondary Primary Secondary	cs_transpose cs_compress cs_symperm cs_add cs_permute	Primary Primary Secondary Primary Secondary	cs_dmperm	Secondary	cs_cholsol cs_lusol cs_dupl cs_nfree	Primary Primary Primary Secondary_Util		
4					cs_qr cs_chol cs_permute cs_load	Secondary Secondary Secondary Primary	cs_dmperm	Secondary		
5					cs_amd cs_post cs_counts cs_etree	Secondary Tertiary Tertiary Tertiary	cs_amd cs_post cs_counts cs_etree	Secondary Tertiary Tertiary Tertiary		

Figure 4.10: CSparse, Tiers of Hub Functions functions are selected from $AA^\top$

Authority tiers	Threshold									
	0.3		0.4		0.5		0.6		0.7	
	Name	Category	Name	Category	Name	Category	Name	Category	Name	Category
1	cs_malloc cs_calloc cs_free cs_spalloc cs_sfree	Primary_Util Primary_Util Primary_Util Primary_Util Secondary_Util	cs_malloc cs_calloc cs_free cs_spalloc cs_sfree	Primary_Util Primary_Util Primary_Util Primary_Util Secondary_Util	cs_malloc cs_calloc cs_free cs_sfree	Primary_Util Primary_Util Primary_Util Secondary_Util	cs_malloc cs_calloc cs_sfree	Primary_Util Primary_Util Secondary_Util	cs_malloc cs_calloc	Primary_Util Primary_Util
2	cs_usolve cs_lu cs_nfree	Secondary Secondary Secondary_Util	cs_usolve cs_lu cs_nfree	Secondary Secondary Secondary_Util	cs_usolve cs_lu cs_nfree cs_pvec cs_ltsolve	Secondary Secondary Secondary_Util Secondary Secondary	cs_spalloc cs_done cs_scatter cs_cumsum	Primary_Util Tertiary_Util Tertiary Tertiary	cs_sfree cs_ipvec cs_nfree	Secondary_Util Secondary Secondary_Util
3	cs_done cs_cumsum cs_scatter	Tertiary_Util Tertiary Tertiary	cs_done cs_scatter	Tertiary_Util Tertiary	cs_fkeep cs_spfree cs_ddone cs_dalloc	Tertiary Primary_Util Tertiary_Util Tertiary_Util	cs_free cs_lsolve cs_ipvec cs_nfree	Primary_Util Secondary Secondary Secondary_Util		
4					cs_idone cs_tdfs	Tertiary_Util Tertiary	cs_fkeep cs_spfree cs_ddone cs_dalloc	Tertiary Primary_Util Tertiary_Util Tertiary_Util		
5							cs_sprealloc	Primary_Util		
6							cs_ndone cs_house cs_happly	Tertiary_Util Tertiary Secondary		

Figure 4.11: CSparse, Tiers of Authority Functions functions are selected from $A^\top A$

Hub tiers	Threshold	
	0.3	0.5
1	tape_doc jacobian forward inverse_Taylor_prop hessian	jacobian forward inverse_Taylor_prop hessian
2	operator + operator / operator * operator - operator <	tape_doc filewrite_start grow reverse hov_ti_reverse
3	inverse_tensor_eva jac_solv tensor_eval hov_ti_reverse read_params	operator + pow operator / operator * operator -
4	ADTOOL_AMPI_popGSVinfo ADTOOL_AMPI_popReduceInfo ADTOOL_AMPI_popGSinfo	filewrite_ampi filewrite filewrite_end put_op_reserve
5	openTape initNewTape filewrite_start init_rev_sweep getTapeInfos	jac_solv inverse_tensor_eva tensor_eval
6		sparse_jac bit_vector_propagation sparse_hess

Figure 4.12: ADOL-C, Tiers of Hub Functions functions are selected from $AA^{\top}$

Authority tiers	Threshold	
	0.3	0.5
1	myalloc1 fprintf myfree1 adolc_exit myalloc2	myalloc1 myfree1 myalloc2
2	loc next_loc ADOLC_PUT_LOCINT put_op ADOLC_PUT_VAL	adolc_exit malloc
3	free malloc MINDEC	loc next_loc ADOLC_PUT_LOCINT put_op ADOLC_PUT_VAL
4	TAPE_AMPI_read_MPI_Comm allocatePack TAPE_AMPI_read_MPI_Datatype TAPE_AMPI_read_int unpackDeallocate	
5	myfree2 myfree3 myalloc3 spread1 hos_forward	
6	fail begin end empty	

Figure 4.13: ADOL-C, Tiers of Authority Functions functions are selected from $A^\top A$

For Threshold = 0.4			
Hub tiers	Functions	Hub tiers	Functions
1	jacobian forward inverse_Taylor_prop hessian	7	init_rev_sweep init_for_sweep put_op_reserve get_op_block_f
2	operator + operator / operator * operator - operator <	8	reg_ext_fct reg_timestep_fct
3	inverse_tensor_eva jac_solv tensor_eval hov_ti_reverse read_params	9	cp_fov_reverse cp_fos_reverse function vec_jac gradient
4	tape_doc ADOLC_TLM_AMPI_Bcast ADOLC_TLM_AMPI_Allgatherv ADOLC_TLM_AMPI_Scatterv ADOLC_TLM_AMPI_Gatherv	10	readConfigFile close_tape filewrite_end cleanUp taylor_close
5	ADTOOL_AMPI_popGSVinfo ADTOOL_AMPI_popReduceInfo ADTOOL_AMPI_popGSinfo	11	lie_covector lie_gradientcv lie_bracket reverse large_jacobian
6	openTape initNewTape filewrite_start getTapeInfos	12	grow free_loc ADTOOL_AMPI_copyActiveBuf ensure_block ADTOOL_AMPI_allocateTemp Buf

Figure 4.14: ADOL-C, Hub Functions are selected from $AA^{\top}$, $threshold = 0.4$

For Threshold = 0.4					
Authority tiers	Functions	Authority tiers	Functions	Authority tiers	Functions
1	myalloc1 fprintf myfree1 adolc_exit myalloc2	6	begin end empty	11	fclose strlen strcmp clearTapeBaseNames strncpy
2	loc next_loc ADOLC_PUT_LOCINT put_op ADOLC_PUT_VAL	7	fail fread fseek fopen	12	BW_AMPI_Barrier BW_AMPI_Wait BW_AMPI_Recv BW_AMPI_Send
3	free malloc MINDEC	8	cp_hov_forward; cp_fov_forward cp_hos_forward dummy edfoo_iarr_wrapper_fov_reverse	13	assert emplace_front erase_after before_begin
4	TAPE_AMPI_read_MPI_Datatype TAPE_AMPI_read_int TAPE_AMPI_read_MPI_Comm allocatePack unpackDeallocate	9	spread1 pack1 zos_forward pack2	14	edfoo_iarr_wrapper_zos_forward edfoo_iarr_wrapper_function edfoo_iarr_wrapper_fov_forward edfoo_wrapper_fov_reverse edfoo_wrapper_fos_reverse
5	myfree2 myfree3 myalloc3 hos_forward	10	cp_hov_reverse cp_hos_reverse cp_fos_forward edfoo_iarr_wrapper_fos_reverse edfoo_iarr_wrapper_fos_forward		

Figure 4.15: ADOL-C, Authority Functions are selected from $A^\top A$, $threshold = 0.4$

Chapter 5

Summary and Future Work

In this thesis we have presented an approach to group or cluster the design elements of scientific software into "tiers" ranked by their "importance". First we have analyzed the dependency structures of software using tool "Understand" and built DSM. The DSM of the call graph provides a convenient tool so that linear algebraic techniques can be applied to identify important callers and callees through the calculation of matrix exponential. Using the notion of "importance" of design elements [13], we ranked those design elements. Then we uncovered "similarity" among design elements. Finally, using the "similarity" among design elements, we group them into tiers. Besides using only DSM, we also used matrices $B_1 = AA^\top$ and $B_2 = A^\top A$ to choose functions which have semantic dependency relationship between functions. We applied our algorithm on CSparse and ADOL-C software implemented in `C`.

Using our algorithm, we can categorize all important functions of the system. This category helps the user of the system to identify their usable functions (hubs). A good system needs to be updated regularly. But for variety of reasons legacy software may not contain adequate technical documentation so that from a usability point of view it may be difficult to detect and retrieve components that could be reused in other software projects. Therefore using our approach a developer of a system can identify important authority functions which are called by important hubs. This finding suggests that the analysis of different tiers of functions of a software system might serve as guidance to developers in the challenging task of redesigning a software by detecting and retrieving components that

could be reused in other software projects.

In terms of further research, we would like to include more test problems of larger networks. It would be interesting to include problems from different scientific domain, for example, SNAP and DIMACS10 to identify groups of important nodes. In this research we have considered unweighted matrices because to identify similarity we needed only information about dependency. For further research, we can compare our algorithm for both weighted and unweighted matrices. Also, in future, we would like to apply our method to software library of legacy code, where very little or no documentation is available about the project. The method that we developed in this thesis is likely to be useful for these types of library.

Bibliography

[1] Lazima Ansari, Shahadat Hossain, and Ahamad Imtiaz Khan. DSMDE: A data exchange format for design structure models. *Sustainability in Modern Project Management: Proceedings of the 18th International DSM Conference*, pages 111–121, 2016.

[2] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, 2e*. Addison Wesley, 2003.

[3] Michele Benzi, Ernesto Estrada, and Christine Klymko. Ranking hubs and authorities using matrix functions. *Linear Algebra and its Applications*, 438(5):2447–2474, 2013.

[4] Dan Braha and Yaneer Bar-Yam. The statistical mechanics of complex product development: Empirical and analytical results. *Management Science*, 53(7):1127–1145, 2007.

[5] Jonathan J Crofts and Desmond J Higham. A weighted communicability measure applied to complex brain networks. *Journal of the Royal Society, Interface*, 6(33):411–414, 2009.

[6] Timothy A. Davis. Direct methods for sparse linear systems (fundamentals of algorithms 2). *SIAM*, 2006.

[7] Steven D Eppinger and Tyson R Browning. *Design structure matrix methods and applications*. MIT press, 2012.

[8] John Forrest, Ted Ralphs, Stefan Vigerske, Lou Hafer, Bjarni Kristjansson, jpfasano, Edwin Straver, Miles Lubin, Gambini Santos, rlougee, and Matthew Saltzman. coin-or/cbc: Version 2.9.9, July 2018.

[9] Linton C Freeman. conceptual clarification." social networks. *"Centrality in social networks*, 1(3):215–239, 1978.

[10] Gene H Golub and Charles F Van Loan. Matrix computations. *JHU Press*, 3, 2012.

[11] Marco A. Gonzalez. *A new change propagation metric to assess software evolvability*. PhD thesis, University of British Columbia, 2013.

[12] Andreas Griewank, David Juedes, and Jean Utke. ADOL-C: a package for the automatic differentiation of algorithms written in c/c++. *ACM Transactions on Mathematical Software*, 1996.

[13] S Hossain, SF Khan, and R Quashem. On ranking components in scientific software. In *DSM 2015: Modeling and managing complex systems-Proceedings of the 17th International DSM Conference Fort Worth (Texas, USA), 4-6 November 2015*, pages 245–254, 2015.

[14] Shahadat Hossain et al. Efficiently computing with design structure matrices. In *DSM 2010: Proceedings of the 12th International DSM Conference, Cambridge, UK, 22.-23.07. 2010*, pages 345–358, 2010.

[15] Shahadat Hossain and Ahmed Tahsin Zulkarnine. Design structure of scientific software–a case study. In *DSM 2011: Proceedings of the 13th International DSM Conference*, pages 129–141, 2011.

[16] D. Kelly and R. Sanders. Assessing the quality of scientific software. *in Proc of the First International Workshop on Software Engineering for Computational Science and Engineering*, 2008.

[17] Jon M Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5):604–632, 1999.

[18] Anany V Levitin. Introduction to design & analysis of algorithms: For anna university, 2e. *Pearson Education India*, 2009.

[19] Alan MacCormack, John Rusnak, and Carliss Y Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015–1030, 2006.

[20] M. W. Maier, D. Emery, and R. Hilliard. Software architecture: introducing ieee standard 1471. *Computer*, 34(4):107–109, 2001.

[21] Scientific Toolworks Inc. Scitools: Understand. `https://scitools.com/`.

[22] Alexandru Telea, Hessel Hoogendorp, Ozan Ersoy, and Dennie Reniers. Extraction and visualization of call dependencies for large C/C++ code bases: A comparative study. In *Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2009, Edmonton, Alberta, Canada, September 25, 2009*, pages 81–88. IEEE Computer Society, 2009.

Appendix A

List of Functions of CSparse

Node Number	Function Name	Node Number	Function Name	Node Number	Function Name
1	cs_add	23	cs_lsolve	45	cs_symperm
2	cs_amd	24	cs_ltsolve	46	cs_tdfs
3	cs_chol	25	cs_lu	47	cs_transpose
4	cs_cholsol	26	cs_lusol	48	cs_updown
5	cs_compress	27	cs_malloc	49	cs_usolve
6	cs_counts	28	cs_maxtrans	50	cs_utsolve
7	cs_cumsum	29	cs_multiply	51	cs_calloc
8	cs_dfs	30	cs_norm	52	cs_done
9	cs_dmperm	31	cs_permute	53	cs_spalloc
10	cs_droptol	32	cs_pinv	54	cs_sprealloc
11	cs_dropzeros	33	cs_post	55	cs_spfree
12	cs_dupl	34	cs_print	56	cs_ndone
13	cs_entry	35	cs_pvec	57	cs_free
14	cs_ereach	36	cs_qr	58	cs_idone
15	cs_etree	37	cs_qrsol	59	cs_dalloc
16	cs_fkeep	38	cs_randperm	60	cs_ddone
17	cs_gaxpy	39	cs_reach	61	cs_dfree
18	cs_happly	40	cs_scatter	62	cs_nfree
19	cs_house	41	cs_scc	63	cs_sfree
20	cs_ipvec	42	cs_schol	64	cs_realloc
21	cs_leaf	43	cs_spsolve		
22	cs_load	44	cs_sqr		

Figure A.1: List of Functions of CSparse

Appendix B

List of Functions of ADOL-C

No	Name	No	Name	No	Name	No	Name	No	Name
1	function	31	tensor_eval	61	myalloc2	91	operator >>	121	log10
2	gradient	32	filewrite_start	62	myalloc3	92	operator++	122	sinh
3	vec_jac	33	filewrite	63	myfree3	93	operator--	123	tanh
4	jacobian	34	filewrite_ampi	64	myfree1	94	operator +=	124	ceil
5	large_jacobian	35	filewrite_end	65	myfree2	95	operator -=	125	floor
6	jac_vec	36	tape_doc	66	myfreeI2	96	operator *=	126	asinh
7	hess_vec	37	lie_scalarcv	67	myallocI2	97	operator /=	127	acosh
8	hess_mat	38	lie_scalarc	68	myalloc1_uint	98	operator !=	128	atanh
9	hessian	39	lie_gradientcv	69	myalloc1_ulong	99	operator ==	129	erf
10	hessian2	40	lie_gradientc	70	myalloc2_ulong	100	operator <=	130	fabs
11	lagra_hess_vec	41	lie_covector	71	myfree1_uint	101	operator >=	131	fmin
12	forodec	42	lie_bracket	72	myfree1_ulong	102	operator >	132	fmax
13	forodec_	43	lie_scalar	73	myfree2_ulong	103	operator <	133	ldexp
14	accodec_	44	lie_gradient	74	condassign	104	operator +	134	frexp
15	abs_normal	45	jac_pat	75	condeqassign	105	operator -	135	myquad
16	directional_active_gradient	46	absnormal_jac_pat	76	initInternal	106	operator *	136	operator >>
17	abs_normal_	47	generate_seed_jac	77	adouble	107	operator /	137	adubref
18	freecoefflist	48	hess_pat	78	~adub	108	recipr	138	adub
19	tensorpoint	49	generate_seed_hess	79	adubp_from_adub	109	exp	139	blocker
20	tensorsetup	50	deepcopy_HP	80	getValue	110	log	140	nondecreasing
21	freetensorpoint	51	sparse_jac	81	operator double const&	111	sqrt	141	operator[]
22	freetensor	52	sparse_hess	82	operator double&&	112	cbrt	142	lookupindex
23	summand	53	set_HP	83	operator double	113	sin	143	adolc_vec_copy
24	coeff	54	get_HP	84	setValue	114	cos	144	adolc_vec_dot
25	tensor_address	55	bit_vector_propagation	85	operator =	115	tan	145	adolc_vec_axpy
26	LUFactorization	56	freeSparseHessInfos	86	declareIndependen	116	asin	146	deallocate
27	GauszSolve	57	ADOLC_get_sparse_jacobian	87	operator <<=	117	acos	147	unpackDeallocate
28	jac_solv	58	forward	88	operator >>=	118	atan	148	ADOLC_TLM_init
29	inverse_Taylor_prop	59	reverse	89	declareDependent	119	atan2	149	ADOLC_TLM_AMPI_Send
30	inverse_tensor_eva	60	myalloc1	90	operator <<	120	pow	150	ADOLC_TLM_AMPI_Recv

Figure B.1: List of Functions of ADOL-C

No	Name	No	Name	No	Name	No	Name	No	Name
151	ADOLC_TLM_AMPI_Isend	101	ADTOOL_AMPI_pop_AMPI_Request	211	Buffer	241	call_ext_fct_commonPrior	271	zos_forward
152	ADOLC_TLM_AMPI_Irecv	182	ADTOOL_AMPI_push_request	212	SubBuffer	242	all_ext_fct_commonPost	272	hov_forward
153	ADOLC_TLM_AMPI_Wait	183	ADTOOL_AMPI_push_comm	213	zeroAll	243	*get_ext_diff_fct	273	fov_forward
154	ADOLC_TLM_AMPI_Barrier	184	ADTOOL_AMPI_packDType	214	append	244	edfoo_wrapper_function	274	hos_ti_reverse
155	ADOLC_TLM_AMPI_Gather	185	ADTOOL_AMPI_unpackDType	215	getElement	245	edfoo_wrapper_fov_forward	275	fos_reverse
156	ADOLC_TLM_AMPI_Scatter	186	ADTOOL_AMPI_getAdjointCount	216	reg_timestep_fct	246	edfoo_wrapper_zos_forward	276	fov_reverse
157	ADOLC_TLM_AMPI_Allgather	187	ADTOOL_AMPI_setAdjointCountAndTempBuf	217	checkpointing	247	edfoo_wrapper_fos_forward	277	callHandleReverse
158	ADOLC_TLM_AMPI_Gatherv	188	ADTOOL_AMPI_allocateTempBuf	218	get_cp_fct	248	edfoo_wrapper_fos_reverse	278	callHandleForward
159	ADOLC_TLM_AMPI_Scatterv	189	ADTOOL_AMPI_releaseAdjointTempBuf	219	init_edf	249	edfoo_wrapper_fov_reverse	279	callHandlePrimal
160	ADOLC_TLM_AMPI_Allgatherv	190	ADTOOL_AMPI_allocateTempActiveBuf	220	cp_zos_forward	250	edfoo_iarr_wrapper_function	280	initTape
161	ADOLC_TLM_AMPI_Reduce	191	ADTOOL_AMPI_copyActiveBuf	221	revolve_for	251	edfoo_iarr_wrapper_fov_forward	281	freeTape
162	ADOLC_TLM_AMPI_Allreduce	192	ADTOOL_AMPI_setupTypes	222	cp_fos_reverse	252	edfoo_iarr_wrapper_zos_forward	282	mediAddHandle
163	ADOLC_TLM_AMPI_Bcast	193	ADTOOL_AMPI_cleanupTypes	223	cp_fov_reverse	253	edfoo_iarr_wrapper_fos_forward	283	mediInitTape
164	AMPI_Init_NT	194	MPI_Datatype ADTOOL_AMPI_FW_rawType	224	cp_clearStack	254	edfoo_iarr_wrapper_fos_reverse	284	mediInitStatic
165	ADTOOL_AMPI_pushBcastInfo	195	MPI_Datatype ADTOOL_AMPI_BW_rawType	225	cp_takeshot	255	edfoo_iarr_wrapper_fov_reverse	285	addHandle
166	ADTOOL_AMPI_popBcastInfo	196	AMPI_Send	226	cp_restore	256	iteration	286	pdouble
167	ADTOOL_AMPI_pushDoubleArray	197	AMPI_Recv	227	cp_taping	257	fp_zos_forward	287	mkparam
168	ADTOOL_AMPI_popDoubleArray	198	AMPI_Isend	228	cp_release	258	fp_fos_forward	288	mkparam_idx
169	ADTOOL_AMPI_pushReduceInfo	199	AMPI_Irecv	229	revolveError	259	fp_fos_reverse	289	adjust
170	ADTOOL_AMPI_popReduceCountAndType	200	AMPI_Wait	230	edf_zero	260	fp_iteration	290	revolve
171	ADTOOL_AMPI_popReduceInfo	201	AMPI_Barrier	231	reg_ext_fct	261	zos_forward_partx	291	rpl_malloc
172	ADTOOL_AMPI_pushSRinfo	202	AMPI_Gather	232	update_ext_fct_memory	262	fos_forward_partx	292	rpl_calloc
173	ADTOOL_AMPI_popSRinfo	203	AMPI_Scatter	233	call_ext_fct	263	hos_forward_partx	293	rpl_realloc
174	ADTOOL_AMPI_pushGSinfo	204	AMPI_Allgather	234	get_ext_diff_fct_v2	264	fov_forward_partx	294	GlobalTapeVarsCL
175	ADTOOL_AMPI_popGScommSizeForRootOrNull	205	AMPI_Gatherv	235	edfoo_v2_wrapper_function	265	hov_forward_partx	295	free_loc
176	ADTOOL_AMPI_popGSinfo	206	AMPI_Scatterv	236	edfoo_v2_wrapper_zos_forward	266	hos_reverse	296	next_loc
177	ADTOOL_AMPI_pushGSVinfo	207	AMPI_Allgatherv	237	edfoo_v2_wrapper_fos_forward	267	hov_reverse	297	ensure_block
178	ADTOOL_AMPI_popGSVinfo	208	AMPI_Bcast	238	edfoo_v2_wrapper_fov_forward	268	hov_ti_reverse	298	grow
179	ADTOOL_AMPI_push_CallCodeReserve	209	AMPI_Reduce	239	edfoo_v2_wrapper_fos_reverse	269	int_reverse_safe	299	initTapeInfos_keep
180	ADTOOL_AMPI_push_AMPI_Request	210	AMPI_Allreduce	240	edfoo_v2_wrapper_fov_reverse	270	hos_forward	300	initNewTape

Figure B.2: List of Functions of ADOL-C

No	Name	No	Name	No	Name	No	Name	No	Name
301	openTape	331	setStoreManagerType	361	set_param_vec	391	spread1	421	get_val_block
302	getTapeInfos	332	reallocStore	362	read_tape_stats	392	pack1	422	TAPE_AMPI_read_int
303	releaseTape	333	fail	363	skip_tracefile_cleanup	393	MINDEC	423	TAPE_AMPI_read_MPI_Datatype
304	set_nested_ctx	334	printError	364	init_for_sweep	394	pack2	424	TAPE_AMPI_read_MPI_Request
305	currently_nested	335	clearTapeBaseNames	365	init_rev_sweep	395	fov_offset_forward	425	accodeout
306	cachedTraceTags	336	createFileName	366	end_sweep	396	fos_forward	426	acccov
307	setTapeInfoJacSparse	337	duplicatestr	367	put_op_reserve	397	hov_wk_forward	427	accadj
308	setTapeInfoHessSparse	338	readConfigFile	368	get_op_block_f	398	free	428	accbrac
309	init_lib	339	take_stock	369	put_loc_block	399	myfree	429	indopro_forward_tight
310	clearCurrentTape	340	keep_stock	370	put_vals_writeBlock	400	spread3	430	indopro_forward_safe
311	cleanUp	341	taylor_begin	371	put_val_block	401	accodec	431	indopro_forward_absnormal
312	removeTape	342	taylor_close	372	get_val_block_f	402	pack3	432	nonl_ind_old_forward_tight
313	trace_on	343	taylor_back	373	get_val_space	403	fprintf	433	nonl_ind_old_forward_safe
314	trace_off	344	write_taylor	374	discard_params_r	404	zos_pl_forward	434	nonl_ind_forward_tight
315	checkInitialStoreSize	345	write_taylors	375	reset_val_r	405	fos_pl_reverse	435	nonl_ind_forward_safe
316	Keeper	346	write_scaylors	376	get_op_f	406	memset	436	calloc
317	initADOLC	347	put_tay_block	377	get_op_r	407	ov_pl_forward	437	int_reverse_tight
318	beginParallel	348	get_taylors	378	get_locint_f	408	fos_pl_sig_reverse	438	int_reverse_safe
319	endParallel	349	get_taylors_p	379	get_locint_r	409	malloc	439	int_forward_tight
320	TapeInfos	350	get_tay_block_r	380	get_val_f	410	dbinomi	440	int_forward_safe
321	copy	351	initTapeBuffers	381	get_num_switches	411	binomi	441	freeSparseJacInfos
322	PersistantTapeInfos	352	start_trace	382	copy_index_domain	412	convert	442	populate_dpp
323	StoreManagerLocintBlock	353	save_params	383	merge_2_index_domains	413	multma2vec1	443	populate_dppp
324	ensureContiguousLocations	354	stop_trace	384	combine_2_index_domains	414	multma2vec2	444	loc
325	setStoreManagerControl	355	close_tape	385	merge_3_index_domains	415	strncpy	445	upd_resloc_check
326	consolidateBlocks	356	freeTapeResources	386	free_tree	416	checkPageBreak	446	ADOLC_PUT_LOCINT
327	enableMinMaxUsingAbs	357	tapestats	387	traverse_crs	417	fflush	447	put_op
328	disableMinMaxUsingAbs	358	printTapeStats	388	traverse_unary	418	fclose	448	upd_resloc
329	adolc_exit	359	get_num_param	389	extend_nonlinearity_domain_binary_ste	419	get_op_block	449	upd_resloc_inc_prod
330	free_all_taping_params	360	read_params	390	extend_nonlinearity_domain_unary	420	get_loc_block_f	450	ADOLC_PUT_VAL

Figure B.3: List of Functions of ADOL-C

No	Name	No	Name	No	Name	No	Name	No	Name
451	value	484	TAPE_AMPI_push_double	517	FW_AMPI_Gatherv	550	BW_AMPI_Gather	583	emplace_front
452	ADOLC_Init_sparse_pattern	485	TAPE_AMPI_pop_double	518	FW_AMPI_Scatterv	551	BW_AMPI_Scatter	584	clear
453	delete_pattern	486	TAPE_AMPI_push_MPI_Op	519	FW_AMPI_Allgatherv	552	BW_AMPI_Allgather	585	pop_front
454	push_back	487	TAPE_AMPI_pop_MPI_Comm	520	FW_AMPI_Bcast	553	BW_AMPI_Gatherv	586	next
455	get_pattern_size	488	TAPE_AMPI_pop_MPI_Op	521	FWB_AMPI_Reduce	554	BW_AMPI_Scatterv	587	cend
456	.get_pattern	489	ADTOOL_AMPI_pop_CallCode	522	FW_AMPI_Allreduce	555	BW_AMPI_Allgatherv	588	cbegin
457	begin	490	assert	523	function_double	556	BW_AMPI_Bcast	589	front
458	end	491	TAPE_AMPI_push_MPI_Request	524	ADOLC_WRITE_SCAYLOR	557	BWB_AMPI_Reduce	590	gcTriggerRatio
459	trunc	492	TAPE_AMPI_pop_MPI_Request	525	dummy	558	BW_AMPI_Allreduce	591	before_begin
460	size	493	MPI_Request ADTOOL_AMPI_pop_request	526	cp_fos_forward	559	spread2	592	erase_after
461	dubref	494	MPI_Comm ADTOOL_AMPI_pop_comm	527	cp_fov_forward	560	getTapeVector	593	sort
462	MPI_Op_create	495	ADTOOL_AMPI_isActiveType	528	cp_hos_forward	561	funcReverse	594	FatalError
463	TAPE_AMPI_read_MPI_Comm	496	memcpy	529	cp_hov_forward;	562	funcForward	595	StoreManagerLocint
464	allocatePack	497	isDerivedType	530	cp_hos_reverse	563	funcPrimal	596	strlen
465	TLM_AMPI_Isend	498	derivedTypeIdx	531	cp_hov_reverse	564	HandleVector	597	sprintf
466	TLM_AMPI_Irecv	499	ADTOOL_AMPI_setAdjointCount	532	evolve_for	565	clearHandles	598	strchr
467	TLM_AMPI_Wait	500	MPI_Abort	533	evolve	566	AdolcMediStatic	599	strtoul
468	TLM_AMPI_Barrier	501	ADTOOL_AMPI_isActiveTyp	534	ADOLC_GET_TAYLOR	567	max	600	strcmp
469	TLM_AMPI_Gather	502	getDTypeData	535	empty	568	maxrange	601	stat
470	TLM_AMPI_Scatter	503	MPI_Type_contiguous	536	top	569	numforw	602	put_vals_notWriteBlock
471	TLM_AMPI_Allgather	504	MPI_Type_commit	537	pop	570	realloc	603	fseek
472	TLM_AMPI_Gatherv	505	MPI_Type_contiguous	538	saveNonAdoubles	571	initTapeInfos	604	fread
473	TLM_AMPI_Scatterv	506	MPI_Type_commit	539	restoreNonAdoubles	572	rewind	605	fopen
474	TLM_AMPI_Allgatherv	507	MPI_Type_free	540	populate_dppp_nodata	573	push	606	fwrite
475	TLM_AMPI_Allreduce	508	FW_AMPI_Send	541	function_iArr	574	resize	607	get_tay_block
476	TLM_AMPI_Bcast	509	FW_AMPI_Recv	542	get_ext_diff_fct	575	init	608	markNewTape
477	MPI_Init	510	FW_AMPI_Isend	543	back	576	pop_back	609	put_op_block
478	TAPE_AMPI_push_int	511	FW_AMPI_Irecv	544	BW_AMPI_Send	577	remove	610	MIN_ADOLC
479	TAPE_AMPI_push_MPI_Datatype	512	FW_AMPI_Wait	545	BW_AMPI_Recv	578	erase	611	get_val_block_r
480	TAPE_AMPI_push_MPI_Comm	513	FW_AMPI_Barrier	546	BW_AMPI_Isend	579	flush	612	extend_nonlinearity_domain_binary_step
481	TAPE_AMPI_pop_MPI_Comm	514	FW_AMPI_Gather	547	BW_AMPI_Irecv	580	touch		
482	TAPE_AMPI_pop_int	515	FW_AMPI_Scatter	548	BW_AMPI_Wait	581	omp_get_thread_num		
483	TAPE_AMPI_pop_MPI_Datatype	516	FW_AMPI_Allgather	549	BW_AMPI_Barrier	582	omp_get_num_threads		

Figure B.4: List of Functions of ADOL-C