

ABSTRACT

LIU, SHU. On Convergence of the Hardy Cross Method of Moment Distribution. (Under the direction of John Baugh.)

The Hardy Cross method of moment distribution is an iterative method that calculates the moments at member ends of statically indeterminate beams and frames. It was published by Professor Hard Cross in the early 1930s and soon became widely adopted by engineers around the world since then. The calculations of the method can be easily performed by hand. The rapid convergence of the method in practice made it possible for engineers to estimate end moments in just a couple of iterations. Interestingly, while engineers have relied on the correctness and convergence of the method, formal proofs were not published until recent years.

Although the method has largely been superseded by the convenience and availability of more general computational approaches, like the direct stiffness method, for decades it was the primary tool to efficiently and safely design many reinforced concrete buildings.

There are several features of the method that make it interesting from a computational point of view. These features include but are not limited to multiprocess oriented domain decomposition, natural parallelism, and scheduling. These features are shared with more challenging problem domains, so the otherwise simplicity of the Hardy Cross method makes it a useful vehicle for studies of those features. Looking at the family of potential distribution sequences—that is, the order in which the calculations are performed—convergence issues arise. For instance, what are the most relaxed conditions on distribution sequences necessary for convergence? From the literature, it appears this question remains unanswered.

Two common types of distribution sequences are used interchangeably by engineers when applying the Hardy Cross method. One is a simultaneous joint balancing distribution sequence, a Jacobi-like iteration in which calculations in the current iteration are based on results from the prior iteration. The other is a consecutive joint balancing distribution sequence, a Gauss-Seidel-like iteration in which calculations use the most recently updated values in the current iteration. While the convergence properties of these two types of distribution sequences have been characterized, individually, a mathematical treatment of their relationship to each other and to other distribution sequences has not been undertaken.

From physical intuition, a valid distribution sequence for the Hardy Cross method is arbitrary as long as every joint gets updated infinitely often. From a computational point of view, the Hardy Cross method can be relaxed even further. There are benefits if these arbitrary sequences and more relaxed forms of the method always converge to the same result for any problem statement. First, the nature of the Hardy Cross method can be more broadly understood as a family of algorithms. Second, the sufficient conditions needed to guarantee convergence of the

method in its more relaxed form can be established.

This thesis proves the equivalence of arbitrary distribution sequences as long as moments at each joint are balanced infinitely often. A more relaxed version of the Hardy Cross method is also introduced. Similarly, this thesis proves that this more relaxed form always converges to the same result for any particular problem statement. The proofs are performed by focusing on the mathematical properties of the corresponding iterate matrices without resorting to physical analogies. In addition, several possible computational implementations of the method, as well as their relationships to matrix forms, are provided. These include traditional ones based on Jacobi-like and Gauss-Seidel-like iterations, as well as a more relaxed form implemented as a multiprocess algorithm.

© Copyright 2015 by Shu Liu

All Rights Reserved

On Convergence of the Hardy Cross Method of Moment Distribution

by
Shu Liu

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Civil Engineering

Raleigh, North Carolina

2015

APPROVED BY:

Murthy Guddati

Pierre Gremaud

John Baugh
Chair of Advisory Committee

DEDICATION

To my family.

BIOGRAPHY

Shu Liu was born in 1990 in Fujian, China. He graduated from University of Sydney in Australia in 2012 with a bachelor degree in civil engineering before beginning the Master of Science program at NC State in 2013.

ACKNOWLEDGEMENTS

I would like to thank Dr. John Baugh for his generous support and patient guidance throughout my study at NC State. I would also like to thank Dr. Murthy Guddati and Dr. Pierre Gremaud for their support and for taking the time to serve on my committee.

TABLE OF CONTENTS

| | |
|---|-------------|
| LIST OF TABLES | vii |
| LIST OF FIGURES | viii |
| Chapter 1 Introduction | 1 |
| Chapter 2 Background | 4 |
| 2.1 Hardy Cross Method of Moment Distribution | 4 |
| 2.2 Jacobi Iteration | 6 |
| 2.3 Gauss-Seidel Iteration | 6 |
| 2.4 Convergence of Hardy Cross Method with Simultaneous Joint Balancing | 7 |
| Chapter 3 Distribution Sequence and Iteration | 10 |
| 3.1 Simultaneous Joint Balancing Distribution Sequence and Jacobi Iteration | 11 |
| 3.2 Consecutive Joint Balancing Distribution Sequence and Gauss-Seidel Iteration | 12 |
| 3.3 Arbitrary Distribution Sequence and Arbitrary Sequence Iteration | 13 |
| Chapter 4 Iterate Matrix Representation | 14 |
| 4.1 Jacobi Iterate Matrix Representation | 14 |
| 4.1.1 General Parameters and Notations | 14 |
| 4.1.2 Distribution Matrix | 15 |
| 4.1.3 Carryover Matrix | 16 |
| 4.1.4 E Matrix | 16 |
| 4.1.5 Iterate Matrix | 17 |
| 4.2 Gauss-Seidel Iterate Matrix Representation | 18 |
| 4.2.1 Iterate Matrix | 18 |
| 4.2.2 Properties of the Iterate Matrix | 18 |
| Chapter 5 Relaxed Hardy Cross Method Convergence Conditions | 20 |
| 5.1 Relaxed Sufficient Condition | 21 |
| 5.1.1 Arbitrary Sequence Iteration Matrix Representation | 21 |
| 5.1.2 Relaxed Sufficient Condition | 22 |
| 5.2 More Relaxed Sufficient Condition | 24 |
| 5.2.1 Further Decomposition of E Matrix | 25 |
| 5.2.2 Elementary Operation Iteration | 26 |
| 5.2.3 More Relaxed Sufficient Condition | 28 |
| Chapter 6 Python Implementation of Hardy Cross Method | 31 |
| 6.1 Graph Representaion of Structure | 31 |
| 6.2 Sequential Implementation | 32 |
| 6.2.1 Flow Chart | 33 |
| 6.2.2 Python Code Snippets | 33 |
| 6.2.3 Example | 36 |

| | | |
|-------------------|---|-----------|
| 6.3 | Asynchronous Shared Memory Parallel Implementation with Simple Termination | 39 |
| 6.3.1 | Flow Chart | 39 |
| 6.3.2 | Python Code Snippets | 40 |
| 6.3.3 | Example | 42 |
| Chapter 7 | Termination Detection | 46 |
| 7.1 | Process Priorities | 46 |
| 7.2 | Quiescence Detection | 47 |
| 7.3 | Snapshot of Global State | 47 |
| 7.4 | Generic Techniques for Distributed Computation | 48 |
| 7.4.1 | Dijkstra’s Ring-Based Termination Detection Algorithm | 48 |
| 7.4.2 | Asynchronous Shared Memory Parallel Implementation with Dijkstra’s Termination Detection Algorithm | 50 |
| Chapter 8 | Conclusions | 53 |
| REFERENCES | | 55 |
| APPENDICES | | 57 |
| Appendix A | Equivalence of Jacobi and Gauss-Seidel Iterations Example | 58 |
| Appendix B | Python Implementation Source Code | 61 |
| B.1 | Sequential Implementation File | 61 |
| B.2 | Asynchronous Shared Memory Parallel Implementation with Simple Termination File | 64 |
| B.3 | Asynchronous Shared Memory Parallel Implementation with Dijkstra’s Termination Detection Algorithm File | 67 |

LIST OF TABLES

| | | |
|-----------|---|----|
| Table 3.1 | Step-by-Step Results of Jacobi Iterations | 11 |
| Table 3.2 | Step by Step Results of Gauss-Seidel Iterations | 12 |
| Table 3.3 | Step by Step Results of Arbitrary Sequence Iterations | 13 |

LIST OF FIGURES

| | | |
|------------|--|----|
| Figure 3.1 | Two Members and Three Joints Structure with Initial Fixed-End Moments . | 10 |
| Figure 6.1 | Two Members and Three Joints Structure with Initial Fixed-End Moments . | 32 |
| Figure 6.2 | Graph Representation of Two Members and Three Joints Structure | 32 |
| Figure A.1 | Two Members and Three Joints Structure | 58 |

Chapter 1

Introduction

The Hardy Cross method of moment distribution [Cross, 1930, Cross and Morgan, 1932] is an iterative method that calculates the moments at member ends of statically indeterminate beams and frames. It is based on a principle that the sum of all the moments at member ends at a non-fixed joint is zero when a structure is in static equilibrium. Professor Hardy Cross published the method in the early 1930s, several years after he began teaching it to his students at the University of Illinois [McCormac, 1975]. Afterward, the method was widely adopted by engineers around the world. It is still being taught in many undergraduate structural analysis courses. Even today, while computers and structural engineering software can efficiently solve many thousands of simultaneous equations, practicing engineers sometimes still use the Hardy Cross method to check those results by hand.

There are several reasons why the method became so popular and pervasive in such a short amount of time. Before the invention of the method, engineers usually needed to set up and solve equations based on physical principles, which involve equilibrium, compatibility, and force-displacement relations. For structures of realistic size, such computations were burdensome since, at that time, they had to be performed by hand. In contrast, the steps of the moment distribution method are easy to remember and to carry out with pencil and paper. The rapid convergence of the method in practice made it possible for engineers to estimate the member end moments in just a few iterations.

Although the method has largely been superseded by the convenience and availability of more general computational approaches, like the direct stiffness method, for decades it was the primary tool to efficiently and safely design reinforced concrete structures [Eaton, 2001].

There are several features of the method that make it interesting from a computational point of view. These features include but are not limited to multiprocess oriented domain decomposition, natural parallelism, and scheduling. These features are shared with more challenging problem domains, so the otherwise simplicity of the Hardy Cross method makes it a useful vehicle for

studies of those features. Looking at the family of potential distribution sequences—that is, the order in which the calculations are performed—convergence issues arise. For instance, what are the most relaxed conditions on distribution sequences necessary for convergence? From the literature, it appears this question remains unanswered.

Two common types of distribution sequences are used interchangeably by engineers when applying the Hardy Cross method. One is a simultaneous joint balancing distribution sequence, a Jacobi-like iteration in which calculations in the current iteration are based on results from the prior iteration. The other is a consecutive joint balancing distribution sequence, a Gauss-Seidel-like iteration in which calculations use the most recently updated values in the current iteration. While the convergence properties of these two types of distribution sequences have been characterized, individually, a mathematical treatment of their relationship to each other and to other distribution sequences has not been undertaken.

From physical intuition, a valid distribution sequence for the Hardy Cross method is arbitrary as long as every joint gets updated infinitely often. From a computational point of view, the Hardy Cross method can be relaxed even further. There are benefits if these arbitrary sequences and more relaxed forms of the method always converge to the same result for any problem statement. First, the nature of the Hardy Cross method can be more broadly understood as a family of algorithms. Second, the sufficient conditions needed to guarantee convergence of the method in its more relaxed form can be established.

This thesis proves the equivalence of arbitrary distribution sequences as long as moments at each joint are balanced infinitely often. A more relaxed version of the Hardy Cross method is also introduced. Similarly, this thesis proves that this more relaxed form always converges to the same result for any particular problem statement. The proofs are performed by focusing on the mathematical properties of the corresponding iterate matrices without resorting to physical analogies. In addition, several possible computational implementations of the method, as well as their relationships to matrix forms, are provided. These include traditional ones based on Jacobi-like and Gauss-Seidel-like iterations, as well as a more relaxed form implemented as a multiprocess algorithm.

The organization of this thesis is as follows. After this introductory chapter, the second includes background on the Hardy Cross method. The third chapter defines and includes examples of (1) the simultaneous joint balancing distribution sequence and Jacobi iteration, (2) the consecutive joint balancing distribution sequence and Gauss-Seidel iteration, as well as (3) a more general case, namely arbitrary distribution sequence and arbitrary sequence iteration. The fourth chapter describes matrix representations of different types of distribution sequences and iteration in detail. Afterward, it lists several important properties of these matrices, some of which will be used for subsequent proofs. The fifth chapter introduces and proves two relaxed sufficient conditions for convergence of the Hardy Cross method so that its characterization

can be more broadly understood. The sixth chapter provides several possible computational implementations of the method in the Python programming language, as well as their relationship to the matrix form. While the asynchronous parallel implementation in the sixth chapter uses simple termination criteria, the seventh chapter lists various more sophisticated termination detection algorithms and mechanisms that could be adopted. Finally, the last chapter concludes this thesis and outlines possible future work.

Chapter 2

Background

2.1 Hardy Cross Method of Moment Distribution

The moment distribution method published by Professor Hardy Cross in the early 1930s is useful for finding end moments in a structure, usually statically indeterminate beams and frames, in an iterative way [Cross, 1930]. In his 1930 paper, Professor Hardy Cross described the distribution sequence of this joint-by-joint method, referred to here as the *simultaneous joint balancing distribution sequence*, as follows:

1. Imagine all joints in the structure are clamped so that they cannot rotate. Under this condition, compute the moments, which are called the initial fixed-end moments, at the ends of each member based on loads on members.
2. Release each non-fixed joint simultaneously by calculating its unbalanced moments, and then redistribute these unbalanced moments among the connecting members in proportion to their stiffness. The unbalanced moment is the algebraic sum of the fixed-end moments at each joint. The relative stiffness of connecting members are called distribution factors.
3. Calculate the carryover moments by multiplying the redistributed moments by the carryover factors, then add this carryover moment to the far end of each member.
4. Repeat step 2 and 3 until the changes in moments are small enough to be neglected, and the results are the true moments at the ends.

The method with the simultaneous joint balancing distribution sequence can be written in a form of an infinite series of matrix productions. More than three decades later, based on the properties of the series, Mozingo found a closed form of the series if the method does converge [Mozingo, 1968]. The closed form can give the exact solution without explicitly performing the iterations but at the cost of inverting a matrix. More recently, the same closed form was

rediscovered by a group of researchers [Lopes et al., 2005] apparently unaware of Mozingo’s publication.

However, while the Hardy Cross method with the simultaneous joint balancing distribution sequence provides remarkable convergence in practice, a proof of its convergence was not published until much more recently [Volkh, 2002]. Volkh characterizes the method with the simultaneous joint balancing distribution sequence as a Jacobi iterative scheme. He starts with the classical displacement method of a structure and then shows an incremental form of the Jacobi iterative scheme that can be used to solve these simultaneous equations. By using a specific starting point, the incremental form of Jacobi iterative scheme can represent the process of applying the Hardy Cross method with the simultaneous joint balancing distribution sequence. Because of the diagonal dominance of the stiffness matrix from the displacement method’s simultaneous equations, the Jacobi iteration—and equivalently the Hardy Cross method—converges for any loading condition. Section 2.4 illustrates these mathematical transformations in detail.

Professor Hardy Cross introduced another type of distribution sequence in a subsequent publication [Cross and Morgan, 1932], referred to here as the *consecutive joint balancing distribution sequence*. While the simultaneous joint balancing distribution sequence does not have a direct physical interpretation, the consecutive one does and can be used in the Hardy Cross method to better visualize the nature of the operations being performed [West, 1989]. Instead of calculating and adding the carryover moments after releasing *all* joints simultaneously, these steps are performed incrementally, one joint at a time. Thus every step in this consecutive joint balancing distribution sequence represents a real physical operation of releasing and clamping of a joint. Based on this type of distribution sequence, the method can be characterized as a Gauss-Seidel iteration [Guo, 1987]. Like Volkh after him, Guo starts with the classical displacement method of a structure and then presents an incremental form of the Gauss-Seidel iterative scheme that can be used to solve its simultaneous equations. Similarly, the Hardy Cross method with the consecutive joint balancing distribution sequence matches this incremental form of Gauss-Seidel iteration with a specific starting point. Due to the fact that the stiffness matrix from the simultaneous equations is positive definite, the Gauss-Seidel iteration—and equivalently the Hardy Cross method—converges for any loading conditions. While Volkh’s overall approach is similar, he was apparently unaware of Guo’s work.

In practice, for the Hardy Cross method, Gauss-Seidel iteration generally converges faster than the Jacobi iteration, but has higher mathematical complexity. Nevertheless, it is possible to find a closed form of this type of iteration in some special cases. In 2009, Dowell found a closed-form moment solution for continuous beams and bridge structures based on this type of distribution sequence [Dowell, 2009]. His new equations are derived from calculus by taking the limit of infinite geometric series generated by the consecutive joint balancing distribution sequence.

Unlike the approach taken by Volokh and Guo, this thesis begins with the Hardy Cross method itself, and then constructs corresponding iterate matrices. It also addresses the equivalence between any arbitrary distribution sequences and iterations, rather than the convergence of a particular case. Technically speaking, the form of the iterations introduced in this thesis include standard Jacobi-like and Gauss-Seidel-like iterations, which directly calculate member end moments instead of their increments, a less direct form that requires further analysis.

2.2 Jacobi Iteration

The Jacobi method [Bronshtein et al., 2007] is an iterative technique for solving simultaneous equations in the form $Ax = b$, where matrix A has non-zero elements on its main diagonal. In each iteration, each equation—say the i^{th} equation—is solved independently as

$$\sum_{j=1}^n a_{ij}x_j = b_i , \quad (2.1)$$

which updates the value of x_i while leaving the other entries of x unchanged [Black et al., b]. As a result, the value of x_i at the k^{th} iteration can be derived as

$$x_i^{(k)} = \frac{b_i - \sum_{j \neq i} a_{ij}x_j^{(k-1)}}{a_{ii}} . \quad (2.2)$$

In each iteration the updates are independent, so the order in which they are performed is irrelevant. Hence, in terms of matrices, the definition of the Jacobi method can be expressed as

$$x^{(k)} = D^{-1}(L + U)x^{(k-1)} + D^{-1}b , \quad (2.3)$$

where the matrices D , $-L$, and $-U$ are the diagonal, strictly lower triangular, and strictly upper triangular parts of A , respectively.

As an alternative approach, the Jacobi iteration can be translated into an incremental form. The incremental Jacobi iteration deals with the residual, $r^{(k)} = b - Ax^{(k)}$, instead of x itself. When the iteration converges, r tends to be 0, which is actually the solution of $Ar = 0$.

2.3 Gauss-Seidel Iteration

The Gauss-Seidel method is also an iterative technique for solving simultaneous equations. It can be used when A is strictly diagonally dominant, or symmetric and positive definite [Black et al., a]. Unlike the Jacobi method, Gauss-Seidel iteration uses updated values as soon

as they are available, so the value of x_i at the k^{th} iteration can be derived as

$$x_i^{(k)} = \frac{b_i - \sum_{j < i} a_{ij} x_j^{(k)} - \sum_{j > i} a_{ij} x_j^{(k-1)}}{a_{ii}} . \quad (2.4)$$

Consequently, multiple updates cannot be performed simultaneously as in the Jacobi method. Moreover, the order in which they are performed affects the intermediate results. Thus, the definition of the Gauss-Seidel method can be expressed with matrices as

$$x^{(k)} = (D - L)^{-1}(Ux^{(k-1)} + b) , \quad (2.5)$$

where the matrices D , $-L$, and $-U$ again are the diagonal, strictly lower triangular, and strictly upper triangular parts of A , respectively. An incremental form of Gauss-Seidel iteration also exists, which minimizes the norm of the residual as well.

2.4 Convergence of Hardy Cross Method with Simultaneous Joint Balancing

This section gives a more detailed review of the approach taken by Volokh, which begins with the displacement method for solving the following canonical system of equilibrium equations

$$Ku + p = 0, \quad (2.6)$$

where K is a stiffness matrix, u is a nodal displacement vector, and p is a nodal loads vector. From physical principles, the matrix K is symmetric and diagonally dominant, and can be written as

$$K_{ii} = \sum_{j=1, i \neq j}^m \bar{M}_{ij} \quad (2.7)$$

and

$$K_{ij} = \frac{1}{2}\bar{M}_{ij} = \frac{1}{2}\bar{M}_{ji}, \quad (2.8)$$

where \bar{M}_{ij} is a reactive moment at the end of the element ij corresponding to a unit rotation of the nodal joint i .

Using the Jacobi iterative procedure to solve Eq. 2.6 gives

$$u^{(k+1)} = -B^{-1}(Cu^{(k)} + p) \quad (2.9)$$

where the superscript in parentheses is the iteration number, $B = \text{Diag}K$, and $C = K - B$.

Further, Eq. 2.9 has a component-wise form of

$$u_i^{(k+1)} = -K_{ii}^{-1} \sum_{j=1, i \neq j}^m (K_{ij}u_j^{(k)} + p_i). \quad (2.10)$$

The starting point of the iteration, $u^{(0)}$, does not affect the convergence of the Jacobi iteration. Let

$$u_i^{(0)} = 0, \quad (2.11)$$

then

$$u_i^{(1)} = -K_{ii}^{-1}p_i. \quad (2.12)$$

Define

$$\Delta u_i^{(k)} = u_i^{(k+1)} - u_i^{(k)}, \quad (2.13)$$

then we have

$$\Delta u_i^{(0)} = u_i^{(1)} - u_i^{(0)} = -K_{ii}^{-1}p_i \quad (2.14)$$

and

$$u_i = \sum_k \Delta u_i^{(k)}. \quad (2.15)$$

Thus we can transform Eq. 2.10 into incremental form as

$$\Delta u_i^{(k+1)} = -K_{ii}^{-1} \sum_{j=1, i \neq j}^m K_{ij} \Delta u_j^{(k)}. \quad (2.16)$$

Pre-multiplying Eq. 2.16 by \overline{M}_{ij} and rearranging the equation we get

$$\overline{M}_{ij} \Delta u_i^{(k+1)} = -K_{ii}^{-1} K_{ij} \sum_{j=1, i \neq j}^m \overline{M}_{ij} \Delta u_j^{(k)}. \quad (2.17)$$

Substituting Eq. 2.7 and Eq. 2.8 into Eq. 2.17 and letting

$$\Delta M_{i,j}^{(k)} = \overline{M}_{ij} \Delta u_j^{(k)}, \quad (2.18)$$

then Eq. 2.17 results in the form

$$\Delta M_{ij}^{(k+1)} = -\frac{\overline{M}_{ij}}{\sum_{j=1, i \neq j}^m \overline{M}_{ij}} \sum_{j=1, i \neq j}^m \frac{\Delta M_{ij}^{(k)}}{2}. \quad (2.19)$$

Eq. 2.19 is the general formal notation of the Hardy Cross method with simultaneous joint

balancing. Its convergence is guaranteed due to the diagonal dominance of matrix K .

Chapter 3

Distribution Sequence and Iteration

An entire family of distribution sequences may be used in the Hardy Cross method, though this more general form has not been demonstrated to be correct. A distribution sequence is actually a sequence of releases or a procedure of cross-carryover. This chapter defines and includes examples of (1) the simultaneous joint balancing distribution sequence and Jacobi iteration, (2) the consecutive joint balancing distribution sequence and Gauss-Seidel iteration, as well as (3) a more general case, namely arbitrary distribution sequence and arbitrary sequence iteration. Step-by-step results are demonstrated for these iterations by applying them to the structure shown in Figure 3.1 [West, 1989].

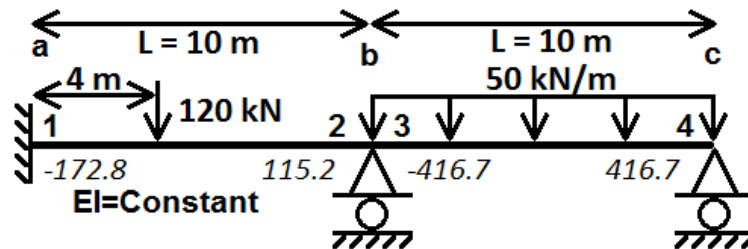


Figure 3.1: Two Members and Three Joints Structure with Initial Fixed-End Moments

It is a simple structure consisting of 3 joints (a, b, and c) and 4 ends (1, 2, 3, and 4). The initial fixed-end moments are -172.8 , 115.2 , -416.7 , and $416.7\text{ kN}\cdot\text{m}$, the distribution factors are 0 , 0.5 , 0.5 , and 1 , and the carryover factors are 0 , 0.5 , 0.5 , and 0.5 for end 1, 2, 3, and 4 respectively. These fixed-end moments, distribution factors, and carryover factors are calculated based on the loads and properties of the structure. However, since this thesis focuses primarily on convergence issues, details for obtaining those values are not presented here.

3.1 Simultaneous Joint Balancing Distribution Sequence and Jacobi Iteration

The simultaneous joint balancing distribution sequence is defined such that in one cycle, all non-fixed joints are balanced simultaneously exactly once, then the carryover moments are recorded simultaneously as well. Its corresponding iteration is essentially a Jacobi iteration. In each iteration, all calculations are based on results from the prior iteration. In other words, moments calculated for joints in the current iteration will not be used until the next iteration. Table 3.1 lists the step-by-step results of applying this approach to the structure shown in Figure 3.1.

Table 3.1: Step-by-Step Results of Jacobi Iterations

| Joint | a | b | | c |
|----------------------------|--------------|-------|--------|--------|
| Member End | 1 | 2 | 3 | 4 |
| Distribution Factor | Not Released | 0.5 | 0.5 | 1.0 |
| Iteration 1: | | | | |
| Starting Fixed-End Moment | -172.8 | 115.2 | -416.7 | 416.7 |
| Redistributed Moment | 0 | 150.7 | 150.8 | -416.7 |
| Carryover Moment | 75.4 | 0 | -208.4 | 75.4 |
| Iteration 2: | | | | |
| Starting Fixed-End Moment | -97.4 | 265.9 | -474.3 | 75.4 |
| Redistributed Moment | 0 | 104.2 | 104.2 | -75.4 |
| Carryover Moment | 52.1 | 0 | -37.7 | 52.1 |
| Iteration 3: | | | | |
| Starting Fixed-End Moment | -45.3 | 370.1 | -407.8 | 52.1 |
| Redistributed Moment | 0 | 18.9 | 18.9 | -52.1 |
| Carryover Moment | 9.4 | 0 | -26.1 | 9.4 |
| Iteration 4: | | | | |
| Starting Fixed-End Moment | -35.9 | 389.1 | -415.0 | 9.4 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| Converged Fixed-End Moment | -27.1 | 406.5 | -406.5 | 0 |

3.2 Consecutive Joint Balancing Distribution Sequence and Gauss-Seidel Iteration

The consecutive joint balancing distribution sequence is defined such that in one cycle, each non-fixed joint is balanced one-by-one exactly once, and the corresponding carryover moments are recorded immediately after balancing each joint. Its corresponding iteration is essentially a Gauss-Seidel iteration. In each iteration, all calculations are based on the most recent available values from either the current or prior iteration. Table 3.2 lists several step-by-step results of applying Gauss-Seidel iterations on the same structure with a distribution sequence of *joint b* \rightarrow *joint c* while *joint a* is not released in each iteration.

Table 3.2: Step by Step Results of Gauss-Seidel Iterations

| Joint | a | b | | c |
|--|--------------|-------|--------|--------|
| Member End | 1 | 2 | 3 | 4 |
| Distribution Factor | Not Released | 0.5 | 0.5 | 1.0 |
| Iteration 1: | | | | |
| Starting Fixed-End Moment | -172.8 | 115.2 | -416.7 | 416.7 |
| Release Joint b : Redistributed Moment | | 150.7 | 150.8 | |
| Release Joint b : Carryover Moment | 75.4 | | | 75.4 |
| Release Joint c : Redistributed Moment | | | | -492.1 |
| Release Joint c : Carryover Moment | | | -246.1 | |
| Iteration 2: | | | | |
| Starting Fixed-End Moment | -97.4 | 265.9 | -512 | 0 |
| Release Joint b : Redistributed Moment | | 123.0 | 123.1 | |
| Release Joint b : Carryover Moment | 61.5 | | | 61.5 |
| Release Joint c : Redistributed Moment | | | | -61.5 |
| Release Joint c : Carryover Moment | | | -30.8 | |
| Iteration 3: | | | | |
| Starting Fixed-End Moment | -35.9 | 388.9 | -419.7 | 0 |
| Release Joint b : Redistributed Moment | | 15.4 | 15.4 | |
| Release Joint b : Carryover Moment | 7.7 | | | 7.7 |
| Release Joint c : Redistributed Moment | | | | -7.7 |
| Release Joint c : Carryover Moment | | | -3.8 | |
| Iteration 4: | | | | |
| Starting Fixed-End Moment | -28.2 | 404.3 | -408.1 | 0 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| Converged Fixed-End Moment | -27.1 | 406.5 | -406.5 | 0 |

3.3 Arbitrary Distribution Sequence and Arbitrary Sequence Iteration

An arbitrary distribution sequence is defined such that in one iteration, some subset of non-fixed joints is balanced simultaneously exactly once, after which the carryover moments are also recorded simultaneously. Assuming that all joints are be updated infinitely often, no joints are left unbalanced. Its corresponding iteration is referred to as *arbitrary sequence iteration*. In each iteration, all calculations are also based on results from the prior iteration. The joints can be chosen arbitrarily in each iteration, so some may be updated more frequently than others. Actually both Jacobi and Gauss-Seidel methods are special cases of arbitrary sequence iteration. Table 3.3 again lists a few step-by-step results of applying one possible set of arbitrary sequence iterations on the same structure.

Table 3.3: Step by Step Results of Arbitrary Sequence Iterations

| Joint | a | b | | c |
|--|--------------|-------|--------|--------|
| Member End | 1 | 2 | 3 | 4 |
| Distribution Factor | Not Released | 0.5 | 0.5 | 1.0 |
| Iteration 1: | | | | |
| Starting Fixed-End Moment | -172.8 | 115.2 | -416.7 | 416.7 |
| Release Joint b and c : Redistributed Moment | | 150.7 | 150.8 | -416.7 |
| Release Joint b and c : Carryover Moment | 75.4 | | -208.4 | 75.4 |
| Iteration 2: | | | | |
| Starting Fixed-End Moment | -97.4 | 265.9 | -474.3 | 75.4 |
| Release Joint b only: Redistributed Moment | | 104.2 | 104.2 | |
| Release Joint b only: Carryover Moment | 52.1 | | | 52.1 |
| Iteration 3: | | | | |
| Starting Fixed-End Moment | -45.3 | 370.1 | -370.1 | 127.5 |
| Release Joint c only: Redistributed Moment | | | | -127.5 |
| Release Joint c only: Carryover Moment | | | -63.8 | |
| Iteration 4: | | | | |
| Starting Fixed-End Moment | -45.3 | 370.1 | -433.9 | 0 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| Converged Fixed-End Moment | -27.1 | 406.5 | -406.5 | 0 |

Chapter 4

Iterate Matrix Representation

Jacobi and Gauss-Seidel iterations can be represented in matrix form. This chapter illustrates the constructions of these iterate matrices in detail, then shows several properties of them. Some of these properties will be used in later chapters to demonstrate convergence conditions.

4.1 Jacobi Iterate Matrix Representation

4.1.1 General Parameters and Notations

This section defines some general parameters and notations that are also used in subsequent sections. Let N be the number of joints and M the number of member ends. Obviously, M is an even number. Define

$$\begin{aligned} p(i) &= j \\ p(j) &= i \end{aligned} \tag{4.1}$$

as a mapping between the indexes i and j of the two ends of a member. Eq. 4.1 suggests

$$i = p(p(i)) . \tag{4.2}$$

Let K_i be the stiffness of the member containing end i . Member stiffnesses are not necessarily symmetric, so K_i may not be the same as $K_{p(i)}$. Stiffness is always positive, so

$$K_i > 0 . \tag{4.3}$$

4.1.2 Distribution Matrix

Let matrix D , whose size is $M \times M$, be a distribution matrix of the redistribution process. Let set $S_i, i = 1, 2, \dots, M$, be a set containing all ends, including i itself, that meet at a common joint. For instance, in the prior example ends ab, ba, bc , and cb are numbered 1 through 4, so at joint b we have $S_2 = \{2, 3\}$ and $S_3 = \{2, 3\}$. This thesis considers only structures with $N \geq 3$ and all joints connected together. Hence at least one of the joints has two or more ends connected, e.g., $\exists i : size(S_i) \geq 2$. Since one end can only be connected to one joint,

$$\begin{aligned} S_i \cap S_j &= \emptyset \text{ if and only if end } i \text{ and end } j \text{ do not frame into a common joint} \\ S_i &= S_j \text{ if and only if end } i \text{ and end } j \text{ frame into a common joint .} \end{aligned} \quad (4.4)$$

Let D_i be the i^{th} column of D . Let d_{ij} be the element at i^{th} row and j^{th} column in matrix D . d_{ij} is the ratio between the increment of the moment at end i due to the moment at end j during the redistribution process. Thus

$$d_{ij} = \begin{cases} 0 & \text{if } S_i \neq S_j \text{ or end } i \text{ is connected to a fixed joint} \\ -\frac{K_i}{\sum_{k \in S_i} K_k} & \text{otherwise} \end{cases} \quad (4.5)$$

that gives D some special properties:

1. D has multiple identical columns. Particularly, if $S_j = S_k$, then D_j is identical to D_k ,

$$d_{ij} = d_{ik} \text{ if } S_j = S_k . \quad (4.6)$$

Thus there are at least $size(S_j)$ columns identical to D_j including itself. Also because of Eq. 4.4, there is at most one distinct negative value in each row.

2. D is singular.

- 3.

$$0 \geq d_{ij} \geq -1 . \quad (4.7)$$

Moreover, $0 > d_{ij} \geq -1$ if and only if $S_i = S_j$ and the corresponding joint is not fixed. This implies

$$d_{ii} \begin{cases} = 0 & \text{if end } i \text{ is fixed} \\ < 0 & \text{otherwise} \end{cases} . \quad (4.8)$$

4. Column sum of D_i is either 0 if end i is fixed, or -1 otherwise:

$$\sum_{j=1}^M d_{ij} = \sum_{\forall j: S_i=S_j} d_{ij} = \begin{cases} 0 & \text{if end } i \text{ is fixed} \\ -1 & \text{otherwise} \end{cases} \quad (4.9)$$

5. Due to Eq. 4.4 and 4.5,

$$d_{ij} < 0 \Rightarrow d_{p(i),j} = 0, \quad (4.10)$$

which means an end will not be affected by the other end in the same member during the redistribution process.

4.1.3 Carryover Matrix

Let matrix C be the matrix of carryover factors. Because the carryover moments are only transferred between the two ends of a member,

$$\begin{cases} 0 < c_{ij} < 1 & \text{if } j = p(i) \\ c_{ij} = 0 & \text{otherwise} \end{cases} . \quad (4.11)$$

So matrix CD represents the ratio between the carryover moment on one end and the calculated unbalanced moment of the other end in the same member. Matrix CD essentially swaps any two rows in D that correspond to the two ends of a member, and then multiplies each row by the corresponding carryover factor. Also due to Eq. 4.10, D and CD cannot both have negative elements at the same position, e.g. $d_{ij} \times [CD]_{ij} = 0$.

4.1.4 E Matrix

Define

$$E = D + CD \quad (4.12)$$

and E has some special properties similar to D :

1. E has multiple identical columns. Particularly, if $S_j = S_k$, then E_j is identical to E_k ,

$$e_{ij} = e_{ik} \text{ if } S_j = S_k . \quad (4.13)$$

Thus there are at least $size(S_j)$ columns identical to E_j including itself. Also because of Eq. 4.4, there are at most two distinct negative values in each row.

2. E is singular .

- 3.

$$0 \geq e_{ij} \geq -1 . \quad (4.14)$$

Moreover, $0 > e_{ij} \geq -1$ if and only if $S_i = S_j$ and the corresponding joint is not fixed, or end j is connected to the same joint where the other end of the member that has end i is connected to, e.g. $j = p(i)$. This implies

$$e_{ii} = d_{ii} \begin{cases} = 0 & \text{if end } i \text{ is fixed} \\ < 0 & \text{otherwise} \end{cases} \quad (4.15)$$

and if $S_i = S_j$, then

$$e_{p(i),j} = c_{p(i),j} e_{ij} . \quad (4.16)$$

4. Column sum of E_i is either 0 if end i is fixed or less than 0 and greater than -2 otherwise.

$$0 \geq \sum_{j=1}^M e_{ij} = \sum_{\forall j: S_i=S_j \vee j=p(i)} e_{ij} > -2 \quad (4.17)$$

5.

$$\sum_{\forall j: S_i=S_j} e_{ij} = \sum_{\forall j: S_i=S_j} d_{ij} = \begin{cases} 0 & \text{if end } i \text{ is fixed} \\ -1 & \text{otherwise} \end{cases} \quad (4.18)$$

4.1.5 Iterate Matrix

The Jacobi iterate matrix of the Hardy Cross method, B_{Jacobi} , can be written as

$$B_{Jacobi} = I + D + CD = I + E \quad (4.19)$$

and the iteration can be written as

$$\mathcal{M}^{(k+1)} = B_{Jacobi} \mathcal{M}^{(k)} \quad (4.20)$$

where $\mathcal{M}^{(k)}$ is a vector representing moments at each member end at step k . The limit of Eq. 4.20 and the Jacobi iteration is ¹

$$\mathcal{M}^* = B_{Jacobi}^{\infty} \mathcal{M}^{(0)} , \quad (4.21)$$

where \mathcal{M}^* is the converged result.

¹An alternative expression could be $\mathcal{M}^* = \mathcal{M}^{(0)} + D\mathcal{M}^{(0)} + CD\mathcal{M}^{(0)} + DCD\mathcal{M}^{(0)} + CDCD\mathcal{M}^{(0)} + \dots$, which gives an exact closed form solution of $\mathcal{M}^* = (I + D)(I - CD)^{-1}\mathcal{M}^{(0)}$ [Mozingo, 1968].

4.2 Gauss-Seidel Iterate Matrix Representation

4.2.1 Iterate Matrix

For the Gauss-Seidel iteration shown in Section 3.2, the Hardy Cross method can also be represented as a simple iteration similar to Eq. 4.20 but with a different iterate matrix, B_{GS} .

The matrix E can be represented as

$$E = \sum_{l=1}^N \widehat{E}_l \quad (4.22)$$

where N is the number of joints, and \widehat{E}_l is a square matrix that contains some columns of E that correspond to all ends connected to joint l while the other columns of \widehat{E}_l are zero.² The matrix representing the process of releasing a joint l once can be written as

$$\widehat{B}_l = I + \widehat{E}_l . \quad (4.23)$$

The iterate matrix B_{GS} of Gauss-Seidel iteration can be written as

$$B_{GS} = \prod_{l=1}^N \widehat{B}_l = \prod_{l=1}^N (I + \widehat{E}_l) . \quad (4.24)$$

The iteration can be written as

$$\mathcal{M}^{(k+1)} = B_{GS} \mathcal{M}^{(k)} . \quad (4.25)$$

The limit of Eq. 4.25 and the Gauss-Seidel iteration is

$$\mathcal{M}^* = B_{GS}^{\infty} \mathcal{M}^{(0)} , \quad (4.26)$$

where \mathcal{M}^* again is the converged result.

4.2.2 Properties of the Iterate Matrix

Here are some properties of this type of iteration matrix.

Theorem 1. *Let \widehat{B}_l be the matrix representing the process of releasing a joint l once, then \widehat{B}_l is idempotent, i.e. $\widehat{B}_l^2 = \widehat{B}_l$.*

²For convenience, the indices of joints are numbered instead of labelled in some equations.

Proof. Due to Eq. 4.5, 4.6, 4.16, and 4.18,

$$(\widehat{E}_l^2)_{ij} = \sum_{k=1}^M (\widehat{E}_l)_{ik} (\widehat{E}_l)_{kj} = (\widehat{E}_l)_{ij} \sum_{k=1}^M D_{kj} = -(\widehat{E}_l)_{ij} , \quad (4.27)$$

which means ³

$$\widehat{E}_l^2 + \widehat{E}_l = 0 . \quad (4.28)$$

Then

$$\begin{aligned} 2\widehat{E}_l + \widehat{E}_l^2 &= \widehat{E}_l \\ \Rightarrow I + 2\widehat{E}_l + \widehat{E}_l^2 &= I + \widehat{E}_l \\ \Rightarrow (I + \widehat{E}_l)^2 &= I + \widehat{E}_l \\ \Rightarrow \widehat{B}_l^2 &= \widehat{B}_l . \end{aligned} \quad (4.29)$$

□

The physical interpretation of Theorem 1 is that releasing a particular joint consecutively twice is the same as doing so only once. This make sense because the unbalanced moment will be zero after the first release.

³Eq. 4.27 and Eq. 4.28 are still valid even if members are not symmetry, e.g. $K_i \neq K_{p(i)}$.

Chapter 5

Relaxed Hardy Cross Method Convergence Conditions

This chapter introduces and proves sufficient conditions for the convergence of two more relaxed forms of the Hardy Cross method. These iterations are referred to here as *arbitrary sequence iteration* and *elementary operation iteration*. The main idea of these proofs is summarized in the following lemma:

Lemma 1. *For a particular problem and type of relaxed iteration, convergence of the Hardy Cross method is guaranteed if, in the limit, the value of \mathcal{M}^* , is constant under any variations allowed by the relaxed iteration, and one of those variations can be shown to converge.*

The key mathematical expression used in the proofs below, i.e., the right-hand side of Eq. 5.6, was inspired by the results of expanding and simplifying the matrix productions of Jacobi and Gauss-Seidel iterations. Similar patterns emerged in both cases, and it seemed likely that the expressions would become identical as the matrix production series goes to infinity. This chapter first confirms this discovery by proving that, in the limit, arbitrary sequence iteration produces the same result regardless of any variations allowed. In this case, the allowed variations are any arbitrary selections of the subset of joints to be released simultaneously in each iteration, as long as every joint gets updated infinitely often. Since both simultaneous joint balancing and consecutive joint balancing ones are special cases of the arbitrary sequence iteration, and since the convergence of both has been proved, we know that arbitrary sequence iteration converges.

This chapter also demonstrates that the results of the elementary operation iteration are always the same regardless of any variation allowed. The variations allowed in this more relaxed iteration refer to arbitrarily choosing an available elementary operation, as long as every joint gets updated infinitely often and each joint finishes updating itself before it gets released again. It can be shown that the variations allowed in the arbitrary sequence iteration are always allowed

in the elementary operation iteration. As a result, the arbitrary sequence iteration is a special case of the elementary operation iteration, and the elementary operation iteration is indeed a more relaxed sufficient condition for the convergence of the method.

5.1 Relaxed Sufficient Condition

This section describes one relaxed sufficient condition for the convergence of the Hardy Cross method. The main idea of this relaxed sufficient condition is that the joints to be released in each iteration can be selected arbitrarily, as long as every joint get updated infinitely often and all the selected joints in each iteration are released simultaneously. In other words, the order in which joints to be released over iterations will not affect the result, and completion of one cycle before start another one is not required. Such type of iteration of the method is referred to as arbitrary sequence iteration in this thesis.

5.1.1 Arbitrary Sequence Iteration Matrix Representation

An arbitrary sequence iteration defined in Section 3.3 could also be represented as an equation similar to Eq. 4.20 but with a different iterate matrix, $B_R^{(k)}$, at each iteration, where k is the index of iteration, and where $k = 0$ corresponds to the initial state that $B_R^{(0)} = I$. The iterate matrix $B_R^{(k)}$ of arbitrary sequence iteration can be written as

$$B_R^{(k)} = I + \sum_{l \in V^{(k)}} \widehat{E}_l, \quad k = 1, 2, 3, \dots \quad (5.1)$$

where $V^{(k)}$ is a schedule set containing a subset of joints to be released simultaneously at the k^{th} iteration. The iteration can be written as

$$\mathcal{M}^{(k+1)} = B_R^{(k)} \mathcal{M}^{(k)}. \quad (5.2)$$

The limit of Eq. 5.2 and the arbitrary sequence iteration is

$$\mathcal{M}^* = \prod_{k=0}^{\infty} B_R^{(k)} \mathcal{M}^{(0)}. \quad (5.3)$$

Actually both Jacobi and Gauss-Seidel iterations are special cases of arbitrary sequence iteration. They could be represented by Eq. 5.2 with special types of $V^{(k)}$. Particularly, when $V^{(k)}$ always includes all joints, then for Jacobi we have

$$B_R^{(k)} = I + \sum_{l=1}^N \widehat{E}_l = I + E = B_{Jacobi}. \quad (5.4)$$

When $V^{(k)}$ only includes a single joint $(k \bmod N) + 1$ at a time, then for Gauss-Seidel we have

$$\prod_{k=(n-1)N+1}^{nN} B_R^{(k)} = \prod_{l=1}^N (I + \widehat{E}_l) = B_{GS}, n = 1, 2, 3, \dots \quad (5.5)$$

5.1.2 Relaxed Sufficient Condition

A simple argument for the equivalence between Jacobi and Gauss-Seidel iterations could be that since both types of iterations can correctly solve the simultaneous equations of the displacement method for any loading conditions, if the solution is unique, then both of them always converge to the same result. This is not a coincidence. Actually, the Hardy Cross method can be performed with arbitrary sequence iteration and always yield the same result.

Theorem 2. *If all joints get updated infinitely often, then the limit of the matrix production representation of any arbitrary sequence iteration is always the same after simplification, which can be written as*

$$\prod_{k=0}^{\infty} B_R^{(k)} = \sum_{m=0}^{\infty} U_m, \quad (5.6)$$

where

$$U_0 = I \quad (5.7)$$

and

$$U_m = \sum_{l_1} \sum_{l_2 \neq l_1} \sum_{l_3 \neq l_2} \cdots \sum_{l_m \neq l_{(m-1)}} \widehat{E}_{l_1} \widehat{E}_{l_2} \widehat{E}_{l_3} \cdots \widehat{E}_{l_m}, \quad m = 1, 2, 3, \dots \quad (5.8)$$

Proof. The left-hand side of Eq. 5.6 captures all the variances allowed in the arbitrary sequence iteration. The right-hand side of Eq. 5.6 is a constant expression that only depends on the stiffness and layout of a structure. Substituting Eq. 5.7 and Eq. 5.8 into Eq. 5.6 gives

$$\prod_{k=0}^{\infty} B_R^{(k)} = I + \sum_{l_1} \widehat{E}_{l_1} + \sum_{l_1} \sum_{l_2 \neq l_1} \widehat{E}_{l_1} \widehat{E}_{l_2} + \sum_{l_1} \sum_{l_2 \neq l_1} \sum_{l_3 \neq l_2} \widehat{E}_{l_1} \widehat{E}_{l_2} \widehat{E}_{l_3} + \dots \quad (5.9)$$

In order to prove Eq. 5.6, the left-hand side will first be rewritten as a summation instead of a production. Then we will show the summation equals the right-hand side.

Define $G^{(k)}$ as

$$G^{(k)} = \prod_{i=0}^k B_R^{(i)}, \quad (5.10)$$

then $G^{(k)} \mathcal{M}^{(0)} = \prod_{i=0}^k B_R^{(i)} \mathcal{M}^{(0)}$ is the moment vector after the k^{th} iteration of any particular variance of arbitrary sequence iteration.

Let

$$\Delta G^{(k)} = G^{(k)} - G^{(k-1)}, k = 1, 2, 3, \dots \quad (5.11)$$

and

$$\Delta G^{(0)} = G^{(0)} = I, \quad (5.12)$$

which means the production can be rewritten as a summation as

$$\prod_{k=0}^{\infty} B_R^{(k)} = G^{(\infty)} = \sum_{k=0}^{\infty} \Delta G^{(k)}. \quad (5.13)$$

The physical interpretation of $\Delta G^{(k)} \mathcal{M}^{(0)}$ is the increments of moment vector in the k^{th} iteration.

Then it can be proved that

$$\sum_{k=0}^{\infty} \Delta G^{(k)} = \sum_{m=0}^{\infty} U_m \quad (5.14)$$

by first showing the limit, $\sum_{k=0}^{\infty} \Delta G^{(k)}$, contains exactly once of U_0 and U_1 , then illustrating that if the limit contains exactly once of U_n , it also contains exactly once of U_{n+1} . Lastly, it can be proved the limit does not have terms that have same index in any two consecutive E_l after simplification. Due to the nature of the $B_R^{(k)}$, i.e., it consists of I and some E_l , there is no other possible mathematical term in the limit.

1. Because of Eq. 5.1, Eq. 5.10, and Eq. 5.11, when $k \geq 1$,

$$\begin{aligned} \Delta G^{(k)} &= G^{(k)} - G^{(k-1)} \\ &= B_R^{(k)} G^{(k-1)} - G^{(k-1)} \\ &= \left(I + \sum_{l \in V^{(k)}} \widehat{E}_l \right) G^{(k-1)} - G^{(k-1)} \\ &= \sum_{l \in V^{(k)}} \widehat{E}_l G^{(k-1)}, \end{aligned} \quad (5.15)$$

which suggests I will not appear in $\Delta G^{(k)}$ when $k \geq 1$. Thus the limit, $\sum_{k=0}^{\infty} \Delta G^{(k)}$, contains $\Delta G^{(0)} = U_0 = I$ exactly once as the representation of the initial state.

2. Now consider U_1 . If $B_R^{(k)}$ is the first B_R term that contains \widehat{E}_l , then because of Eq. 5.15 and the fact that $G^{(k-1)} = \sum_{k=0}^{k-1} \Delta G^{(k)}$ has the term $\Delta G^{(0)} = I$, $\Delta G^{(k)}$ is also the first ΔG term that contains \widehat{E}_l . Because all joints get updated infinitely often, i.e., $\exists k : B_R^{(k)}$ has the term \widehat{E}_k , the limit contains all $\widehat{E}_1, \widehat{E}_2, \dots, \widehat{E}_N$. In addition, $\Delta G^{(k)}$ is the only ΔG term that contains \widehat{E}_l after simplification. The reason is if $\Delta G^{(j)}, j > k$ also contains

\widehat{E}_l , then $B_R^{(j)}$ must contain \widehat{E}_l . Since $G^{(j-1)}$ contains both I and \widehat{E}_l exactly once and Eq. 5.15, $\Delta G^{(j)}$ contains both \widehat{E}_l and \widehat{E}_l^2 exactly once. Due to Eq. 4.28, \widehat{E}_l and \widehat{E}_l^2 get canceled in $\Delta G^{(j)}$. Thus, all terms in U_1 , i.e., $\widehat{E}_1, \widehat{E}_2, \dots, \widehat{E}_N$, only occur exactly once.

- 3a. Assume that $\sum_{k=0}^{\infty} \Delta G^{(k)}$ contains exactly once of U_n .
- 3b. If $G^{(k)}$ contains a particular term u of U_n , say $u = \widehat{E}_l w$ that begins with \widehat{E}_l , then $B_R^{(k)}$ must contain \widehat{E}_l .
- (a) Let $B_R^{(j_1)}, j_1 > k$ be the next B_R term that contains \widehat{E}_i , then $\widehat{E}_i u$ first appears in $\Delta G^{(j_1)}$ due to Eq. 5.15. Moreover, $\Delta G^{(j_1)}$ is the only ΔG term that contains $\widehat{E}_i u$ after simplification. The reason is if $\Delta G^{(q)}, q > j_1$ also contains \widehat{E}_i , then $B_R^{(q)}$ must contain \widehat{E}_i . Since $G^{(q-1)}$ contains both u and $\widehat{E}_i u$ exactly once, $\Delta G^{(q)}$ contains both $\widehat{E}_i u$ and $\widehat{E}_i^2 u$ exactly once. Due to Eq. 4.28, $\widehat{E}_i u$ and $\widehat{E}_i^2 u$ get canceled in $\Delta G^{(q)}$. Thus, together with the fact that all joints get updated infinitely often, above statements prove the limit contains exactly once of U_{n+1} .
- (b) Let $B_R^{(j_2)}, j_2 > k$ be the next B_R term that contains \widehat{E}_l . Since $G^{(j_2-1)}$ contains both w and $\widehat{E}_l w$ exactly once and Eq. 5.15, $\Delta G^{(j_2)}$ contains both $\widehat{E}_l w$ and $\widehat{E}_l^2 w$ exactly once. Due to Eq. 4.28, $\widehat{E}_l w$ and $\widehat{E}_l^2 w$ get canceled in $\Delta G^{(j_2)}$. Hence, the limit cannot have terms that have same index in any two consecutive E_l after simplification.

As a result, $\sum_{k=0}^{\infty} \Delta G^{(k)}$ contains exactly once of U_{n+1} .

Hence Eq. 5.14 and equivalently Eq. 5.6 are proved. \square

Together with Lemma 1 and the previous proof of the convergence of the Jacobi iteration, i.e., Volokh's proof, Theorem 2 verifies the convergence of the arbitrary sequence iteration. Thus the arbitrary sequence iteration is indeed a relaxed sufficient condition for the convergence of the Hardy Cross method.

5.2 More Relaxed Sufficient Condition

This chapter introduces an even more relaxed version of the Hardy Cross method based on a concept of elementary operation. Then the convergence of this elementary operation iteration is proved. This more relaxed version is more inspired by asynchronous parallel implementations of the method rather than physical intuition.

5.2.1 Further Decomposition of E Matrix

The matrix E can be further decomposed in two dimensions. In other words, instead of Eq. 4.22, E can be written as

$$E = \sum_{i=1}^N \sum_{j=1}^N [i, j] \quad (5.16)$$

where N is the number of joints. Recall that M is the number of joints, and now we introduce the notation $[i, j]$, an M by M square matrix such that

$$[i, j]_{kl} = \begin{cases} E_{kl} \leq 0 & \text{if end } k \text{ and end } l \text{ is connected to joint } i \text{ and joint } j, \text{ respectively} \\ 0 & \text{otherwise} \end{cases} \quad (5.17)$$

$[i, j]$ contains some elements of E whose rows correspond to all ends connected to joint i , and whose columns correspond to all ends connected to joint j . The other elements of $[i, j]$ are zero. Due to the properties of matrix D , actually

$$[i, i]_{kl} = \begin{cases} D_{kl} \leq 0 & \text{if end } l \text{ is connected to joint } i \\ 0 & \text{otherwise} \end{cases} \quad (5.18)$$

and the column sum of $[i, i]$ observes

$$\sum_k [i, i]_{kl} = \begin{cases} -1 & \text{if end } l \text{ is connected to joint } i \\ 0 & \text{otherwise} \end{cases} \quad (5.19)$$

The physical interpretation of $[i, j]$ is how the moments at joint i are affected by joint j . Thus, when $i = j$, $[j, j]\mathcal{M}^{(k)}$ is the redistribution process of the unbalanced moment at joint j . When $i \neq j$, $[i, j]\mathcal{M}^{(k)}$ is the process of joint j giving a carryover moment to a particular end of joint i . Although $[i, j]$ is zero if joint i and joint j are not connected or joint j is fixed, this case will not be discussed separately for purposes of the proof.

There are two important properties of matrix $[i, j]$ that are required for proofs later in this chapter. Due to Eq. 5.17, Eq. 5.18 and Eq. 5.19,

$$[* , j][j, j] = -[* , j] \quad (5.20)$$

and

$$[* , i][k, j] = 0, i \neq k \quad (5.21)$$

Further, some wild card notations are useful to carry out the proof.

1. Let $[\ast, j]$ be any one of the set $\{[i, j], i = 1, 2, \dots, N\}$.

2. Let $[?, j]$ be any one of the set $\{[i, j], i \neq j\}$.
3. Let $\sum[* , j]$ be the sum of the set $\{[i, j], i = 1, 2, \dots, N\}$. Obviously,

$$\sum[* , j] = \widehat{E}_j . \quad (5.22)$$

4. Let $\sum[?, j]$ be the sum of the set $\{[i, j], i \neq j\}$.

5.2.2 Elementary Operation Iteration

An alternative and more relaxed description of the Hardy Cross method could be a series of elementary operations, which involves taking snapshots and a series of manipulations of $[i, j]$. From a computational point of view, taking a snapshot means remembering current member end moments. Here, an elementary operation is defined as a process that should not be interfered with by another elementary operation.

The process of releasing a single joint can be described as follows:

1. Release a joint, say joint j , starts with taking a snapshot, say \mathcal{M}' , of the current member end moments.
2. Then calculate the increments of moments based on \mathcal{M}' . The process of taking a snapshot and calculating increments of moments is considered as one type of elementary operation, because it does not affect other elementary operations (no data writing) and item 4 below. Increments for redistribution of the unbalanced moment within joint j are $[j, j]\mathcal{M}'$, i.e., the balancing moments. Meanwhile, the increment for moment at joint i due to the unbalanced moment at joint j is $[i, j]\mathcal{M}'$. Adding one of these matrix multiplication results, such as $[i, j]\mathcal{M}'$, to \mathcal{M} is an another type of elementary operation ¹.
3. These increments do not need to be added to \mathcal{M} immediately, and their execution order could be arbitrary. Actually, these increments can be added to an operation set, say O , first. Then the second type of elementary operation mentioned above really is picking one of these matrix multiplication results from O and adding it to \mathcal{M} .
4. The increments within joint j , $[j, j]\mathcal{M}'$ should be added to \mathcal{M} before releasing the same joint again.

Since $[?, j]$ has at most one row that contains non-zero elements, it only updates one value of the moment vector. Although $[j, j]$ could update multiple values of the moment vector because it could have multiple rows that contain non-zero elements, $[j, j]$ can still be considered as an elementary operation due to item 4 above. Thus from a computational implementation point of

¹From a computational point of view, this means $[i, j]\mathcal{M}'$ is executed.

view, if the update process in implementation is performed atomically, then the Hardy Cross method actually executes these elementary operations *sequentially* even if the implementation approach is concurrent. This suggests that the variations allowed in this more relaxed iteration refer to arbitrarily choosing an available elementary operation, as long as every joint gets updated infinitely often and each joint finishes updating itself before it gets released again. An available elementary operation could be either 1) a joint taking a snapshot and calculating increments of moments if no balancing moment for that joint is pending in O , or 2) moving a matrix multiplication result from O to \mathcal{M} .

These variations allowed can also be shown in a mathematical form. Due to the sequential nature of the iteration mentioned above, the iteration can be written as

$$\begin{aligned} \mathcal{M}^{(k+1)} &= W^{(k+1)}(\mathcal{M}^{(k)}) \\ &= \begin{cases} \mathcal{M}^{(k)} & \text{if an arbitrary joint takes a snapshot and} \\ & \text{calculates increments of moments} \\ \mathcal{M}^{(k)} + [i, j]\mathcal{M}^{(l)} & \text{if an arbitrary matrix production result} \\ & \text{is moved from } O \text{ to } \mathcal{M} \end{cases} \end{aligned} \quad (5.23)$$

where $W^{(k+1)}$ is a function that performs an arbitrary available elementary operation on $\mathcal{M}^{(k)}$ at step $k + 1$, as long as every joint gets updated infinitely often and each joint finishes updating itself before it gets released again. Then the limit of Eq. 5.23 and the elementary operation iteration is

$$\mathcal{M}^* = (W^{(\infty)} \dots W^{(3)}W^{(2)}W^{(1)})(\mathcal{M}^{(0)}) . \quad (5.24)$$

By applying one more restriction on W , it is clear that the arbitrary sequence iterations mentioned previously are special cases of this even more relaxed elementary operation iterations. The restriction is that once W chooses an arbitrary available second type of elementary operation, all available second type of elementary operations must be executed—until nothing left in O —before W chooses the first type of elementary operation again. When an arbitrary number of the first type of elementary operations is chosen consecutively, it has exactly the same effect as that of selecting a subset of joints to be released simultaneously. When all the corresponding matrix production results are added to \mathcal{M} , it has exactly the same effect as that of Eq. 5.2. In other words, the difference between the arbitrary sequence iterations and the elementary operation iterations is that the former one implies once the operations in O start to be executed, all operations in O have to be executed before taking moment snapshots for releasing another set of joints while the latter one does not require that.

5.2.3 More Relaxed Sufficient Condition

It can be proved that the Hardy Cross method using above elementary operation iterations always converges to the same result, as long as every joint get updated infinitely often and each joint finishes updating itself before it gets released again.

To define a constant expression to be used in the theorem and proof later in this section, let Z be a sequence as

$$Z_m = \left(\sum_{j_1} \sum[* , j_1] \right) \left(\sum_{j_2 \neq j_1} \sum[* , j_2] \right) \dots \left(\sum_{j_m \neq j_{m-1}} \sum[* , j_m] \right) \quad (5.25)$$

and

$$Z_0 = I . \quad (5.26)$$

Due to Eq. 5.22,

$$Z_m = U_m . \quad (5.27)$$

Define a similar sequence \hat{Z} as

$$\hat{Z}_m^j = \left(\sum_{j_1 \neq j} \sum[* , j_1] \right) \left(\sum_{j_2 \neq j_1} \sum[* , j_2] \right) \dots \left(\sum_{j_m \neq j_{m-1}} \sum[* , j_m] \right) \quad (5.28)$$

and

$$\hat{Z}_0^j = I . \quad (5.29)$$

The relationship between Z and \hat{Z} is

$$Z_m = \sum_j \sum[* , j] \hat{Z}_{m-1}^j . \quad (5.30)$$

Theorem 3. *If all joints get updated infinitely often, all operations are elementary, and each set of elementary operations of releasing a joint is complete, then the Hardy Cross method will always converge to the same result regardless of when each snapshot is taken and regardless of the order of executions of the elementary operations. The converged result is²*

$$\mathcal{M}^* = \sum_{m=0}^{\infty} Z_m \mathcal{M}^{(0)} \quad (5.31)$$

where \mathcal{M}^* and $\mathcal{M}^{(0)}$ are the final balanced and initial moment distributions, respectively.

Proof. This proof will firstly redescribe the elementary operation iteration and rewrite \mathcal{M}^* in

²Due to Eq. 5.27, the right-hand sides of Eq. 5.6 and $\sum_{m=0}^{\infty} Z_m$ are identical.

terms of the final snapshot, P^* . Then by showing $P^* = \sum_{m=0}^{\infty} Z_m$, Eq. 5.31 and the theorem is proved.

The above more relaxed description of the Hardy Cross method can be represented in matrix form as follows:

1. Let the current moment distribution be $\mathcal{M}^{(k)} = (W^{(k)} \dots W^{(3)}W^{(2)}W^{(1)})(\mathcal{M}^{(0)}) = P^{(k)}\mathcal{M}^{(0)}$.
2. $P^{(0)} = I$ at the beginning.
3. When releasing a joint j at step k , the snapshot is current $P^{(k)}$. Then an elementary operation set containing all $[\ast, j]P^{(k)}$ is added to O .
4. In an arbitrary order, elementary operations are picked from O and added to P .
5. $[j, j]P^{(k)}$ has to be executed before releasing joint j again.
6. All joints will be released infinitely often. Thus an elementary operation set containing all $[\ast, j]P^{(k)}$ will be added to O infinitely often.
7. Write the result moment distribution as

$$\mathcal{M}^* = (W^{(\infty)} \dots W^{(3)}W^{(2)}W^{(1)})(\mathcal{M}^{(0)}) = P^*\mathcal{M}^{(0)} \quad (5.32)$$

where P^* is the final snapshot of $P^{(k)}$.

Eq. 5.31 is true if

$$P^* = \sum_{m=0}^{\infty} Z_m, \quad (5.33)$$

where the left-hand side captures all variances allowed in the elementary operation iteration and the right-hand side is a constant expression that only depends on the stiffness and layout of a structure.

The idea of this proof is similar to the proof for Theorem 2. It can be proved by firstly showing Z_0 and Z_1 exist exactly once in P^* . Then if Z_m exists exactly once in P^* and there is no extra term, then Z_{m+1} also exists exactly once in P^* and there is no extra term.

1. Obviously, $Z_0 = I$ exists exactly once in P^* as the representation of the initial state. Z_1 is also added to P^* when each joint get released first time.
- 2a. Assume Z_m exists exactly once in P^* , and there is no extra term.
- 2b. Need to prove Z_{m+1} exists in P^* and there are no extra or duplicated terms.

- (a) Since all joints will be released infinitely often, all terms in Z_m will be pre-multiplied by $\sum_j [* , j]$ and Z_{m+1} exists in P^* .
- (b) Because of Eq. 5.30, the only possible extra terms that may be created in this process are $[* , j][j , j]\widehat{Z}_{m-1}^j$ or $[* , j][? , j]\widehat{Z}_{m-1}^j$. The existence of $[* , j]\widehat{Z}_{m-1}^j$ implies the existence of \widehat{Z}_{m-1}^j that will also get pre-multiplied by $[* , j]$ at the same time, which results a duplicated $[* , j]\widehat{Z}_{m-1}^j$. Since $[* , j][j , j]\widehat{Z}_{m-1}^j = -[* , j]\widehat{Z}_{m-1}^j$ according to Eq. 5.20 above, $[* , j][j , j]\widehat{Z}_{m-1}^j$ gets canceled by the duplicated $[* , j]\widehat{Z}_{m-1}^j$. Meanwhile, $[* , j][? , j]\widehat{Z}_{m-1}^j$ is always zero due to Eq. 5.21 above. As a result, P^* doesn't have any extra term other than Z_{m+1} .
- (c) Obviously, adding any term from Z_{m+1} , say $[* , j]\widehat{Z}_m^j$, to P^* first time will not create any duplicated term. Because $[j , j]P^{(k)}$ has to be executed before releasing joint j again, when $[* , j]\widehat{Z}_m^j$ get created again, $[j , j]\widehat{Z}_m^j$ must exists and $[* , j][j , j]\widehat{Z}_m^j = -[* , j]\widehat{Z}_m^j$ must also be created at the same time. Hence the duplicated $[* , j]\widehat{Z}_m^j$ will be canceled.

The duplicated $[* , j]\widehat{Z}_m^j$ and its corresponding extra term $[* , j][j , j]\widehat{Z}_m^j$ will always be moved from O to $P^{(k)}$ at the same time, due to the fact that \widehat{Z}_m^j and $[j , j]\widehat{Z}_m^j$ are in the same snapshot. Thus Z_{m+1} exists in P^* and there are no extra or duplicated terms.

Hence according to Lemma 1, Eq. 5.6, and the fact that the arbitrary sequence iteration is a special case of the elementary operation iteration, the elementary operation iteration is indeed a more relaxed sufficient condition for the convergence of the method. \square

In both theorems, it seems the fact that the column sum of matrix D is -1 for non-fixed joint plays an important role for simplification. However, whether it is necessary in the general case still remains a question.

Chapter 6

Python Implementation of Hardy Cross Method

This chapter presents and discusses several possible computational implementations of the Hardy Cross method in the Python programming language based on a graph representation of a structure. Since the iterate matrix of the method has a dimension $M \times M$, it may become memory intensive when the size of the structure gets larger. Actually it is not necessary for a computational implementation to explicitly construct the iterate matrix. The relationship between these implementations¹ and the matrix form of the method is demonstrated.

6.1 Graph Representaion of Structure

A structure can be defined as a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where \mathcal{V} is a finite vertex set of all joints and \mathcal{E} is a finite edge set of all member ends. Consider a simple structure [West, 1989] consisting of 3 joints (a, b, and c) and 4 ends (1, 2, 3, and 4) as shown in Figure 6.1. The initial fixed-end moments are -172.8, 115.2, -416.7, and 416.7 kN/m, the distribution factors are 0, 0.5, 0.5, and 1, and the carryover factors are 0, 0.5, 0.5, and 0.5 for ends 1, 2, 3, and 4, respectively. These fixed-end moments, distribution factors, and carryover factors are calculated based on the loads and properties of the structure. However, since the focus of this thesis is on the convergence properties of the Hardy Cross method, the detailed procedure for determining these quantities is not outlined here.

¹See Appendix B for the complete Python source code.

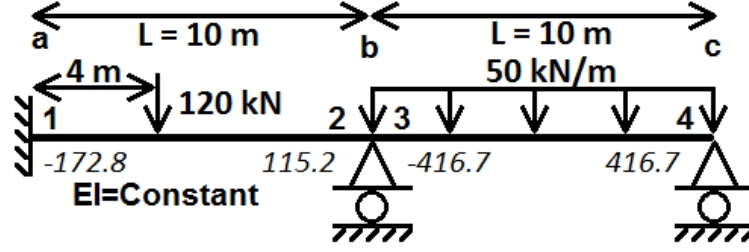


Figure 6.1: Two Members and Three Joints Structure with Initial Fixed-End Moments

This structure can be represented by $\mathcal{G} = (\mathcal{V}, \mathcal{E}, f)$ where $\mathcal{V} = \{a, b, c\}$ and $\mathcal{E} = \{1, 2, 3, 4\}$ as shown in the Figure 6.2.

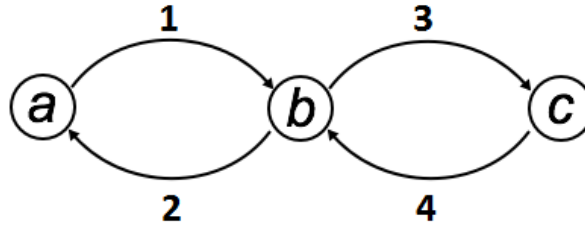


Figure 6.2: Graph Representation of Two Members and Three Joints Structure

Let function $f(j, k) = i$ be a one-to-one mapping function from node indexes j and k to the index of the directed edge from node j to node k . Let function $\hat{f}(i) = (j, k)$ be the reverse one-to-one mapping function of f . Let function $g(j, k) = (f_d, f_c, m)$ be a mapping function from the directed edge from node j to node k to a tuple, where m , f_d , and f_c are the moment, distribution factor, and carryover factor of the member end with index i , respectively. In this case,

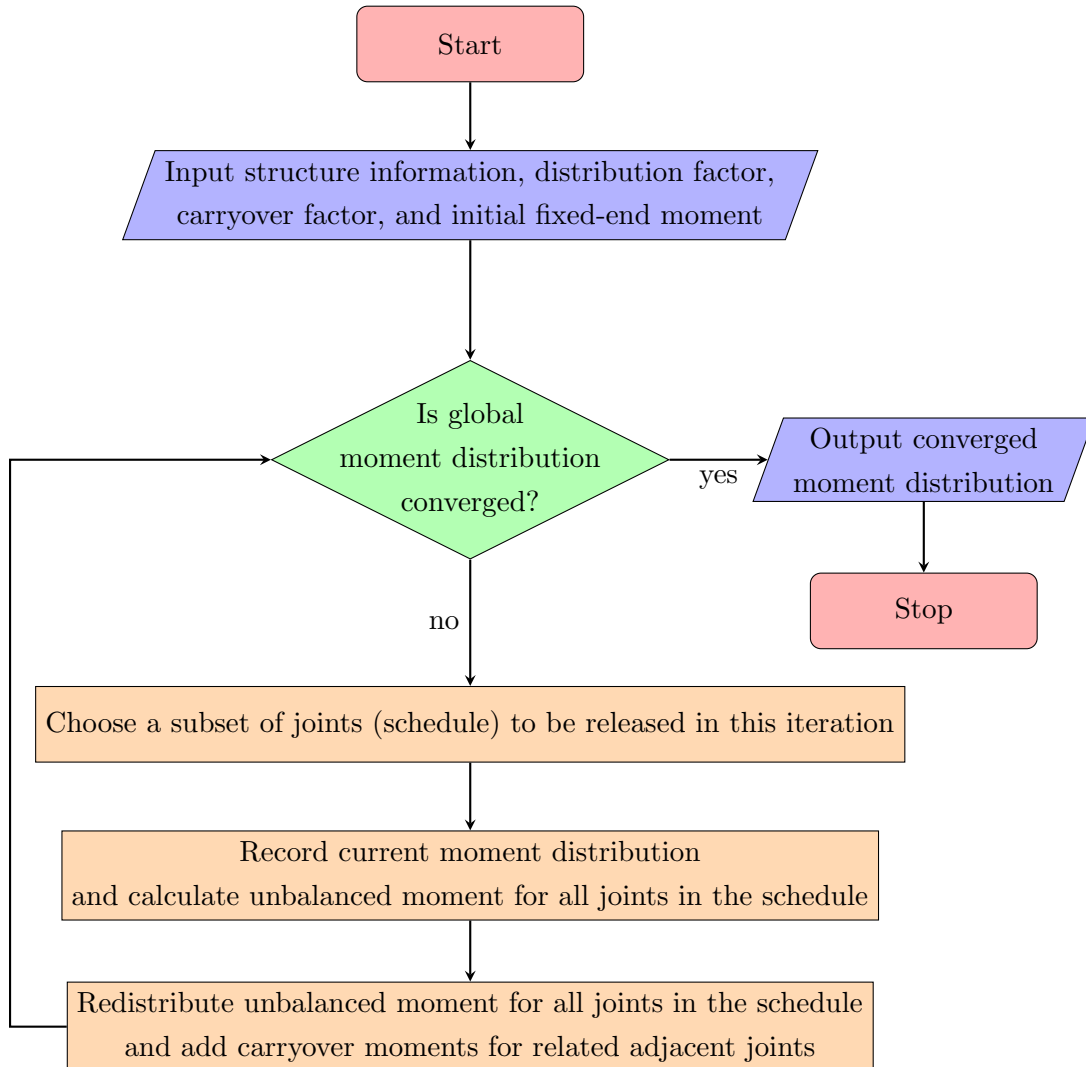
$$\begin{aligned}
 g(\hat{f}(f(a, b))) &= g(\hat{f}(1)) = g(a, b) = (0, 0.5, -172.8) \\
 g(\hat{f}(f(b, a))) &= g(\hat{f}(2)) = g(b, a) = (0.5, 0.5, 115.2) \\
 g(\hat{f}(f(b, c))) &= g(\hat{f}(3)) = g(b, c) = (0.5, 0.5, -416.7) \\
 g(\hat{f}(f(c, b))) &= g(\hat{f}(4)) = g(c, b) = (1, 0.5, 416.7)
 \end{aligned}$$

6.2 Sequential Implementation

This section describes one possible sequential implementation of the method.

6.2.1 Flow Chart

The sequential implementation could be represented as the following flow chart.



6.2.2 Python Code Snippets

Based on the graph representation, a structure can be stored as a dictionary of dictionaries of tuples in an implementation. For example, the structure shown in Figure 6.1 and Figure 6.2 can be stored as

```

1 def ex1():
2     a, b, c = 0, 1, 2
3     g = {a: {}, b: {}, c: {}}
4     g[a][b] = End(0.0, 0.5, -172.8)
5     g[b][a] = End(0.5, 0.5, 115.2)
6     g[b][c] = End(0.5, 0.5, -416.7)
7     g[c][b] = End(1.0, 0.5, 416.7)
8     return g

```

Releasing a joint is a two-step process in order to mimic matrix multiplication where each element is calculated independently. The first step is calculating and recording each joint's unbalanced moment, and the second step is to redistribute the unbalanced moment. The unbalanced moment of a joint is calculated by summing moments of member ends connecting to the joint as

```

1 def get_unbalanced_moment(g, i):
2     unbalanced_moment = 0
3     for j in g[i]:
4         unbalanced_moment += g[i][j].moment
5     return unbalanced_moment

```

The redistribution process distributes the unbalanced moment and carryover moments by

```

1 def release(g, i, unbalanced_moment):
2     for j in g[i]:
3         my_share = g[i][j].distribution_factor * unbalanced_moment
4         g[i][j].moment -= my_share # mimic matrix D
5         g[j][i].moment -= g[i][j].carryover_factor * my_share # mimic matrix CD

```

One iteration based on a particular schedule (a particular schedule set $V^{(k)}$ in Eq. 5.1) can be performed by

```

1 # moment distribution method with schedule
2 def mdm_schedule(g, schedule, tolerance):
3     converged = True
4     unbalanced_moment = {}
5     # prepare releasing a joint by calculating and recording its unbalanced moment
6     for joint in schedule:
7         if has_nonzero_dfs(g, joint):
8             unbalanced_moment[joint] = get_unbalanced_moment(g, joint)
9     # complete releasing a joint by redistributing its unbalanced moment
10    for joint in unbalanced_moment:
11        if abs(unbalanced_moment[joint]) >= tolerance: # stopping criteria
12            converged = False
13            release(g, joint, unbalanced_moment[joint])
14    return converged

```

Different from the infinite sequence of matrix multiplication in theory, the above function **mdm_schedule** includes a stopping criteria for practicality. In fact, a computer has limited precision for floating point numbers, which means it is almost impossible to perfectly and accurately calculate and store the infinite sequence. Moreover, it is very rare that an engineering project requires a large number of significant digits. Thus, in order to get an acceptable result within a reasonable time, a tolerance is necessary. The program terminates when the desired convergence is achieved.

Since only non-fixed joints are released, the function

```

1 def has_nonzero_dfs(g, i):
2     for j in g[i]:
3         if g[i][j].distribution_factor > 0:
4             return True
5     return False

```

could be used to determine whether a joint is fixed.

The above Python code snippets actually implicitly achieve the same effect as the matrix multiplication in Eq. 5.1 by mimicking the matrix E that includes D and CD . Because the unbalanced moment used in the lines

```

1 my_share = g[i][j].distribution_factor * unbalanced_moment
2 g[i][j].moment -= my_share # mimic matrix D

```

is calculated by summing all moments of member ends connecting to the end, multiplying this unbalanced moment by the corresponding distribution factor is exactly the same as multiplying a row of D . Similarly,

```

1 g[j][i].moment -= g[i][j].carryover_factor * my_share # mimic matrix CD

```

is exactly the same as multiplying a row of CD by the moment vector \mathcal{M} . Moreover, unbalanced moments of all joints in the current schedule are calculated before any distribution process, so the second step is independent for each joint. Thus by executing **release(g, i, moment)** and **mdm_schedule(g, schedule, tolerance)**, the matrix multiplication in Eq. 5.1 is implicitly performed for one iteration.

The sequential implementation repeats the above processes until global convergence is reached. With the above functions, the Hardy Cross method with simultaneous joint balancing distribution sequence and Jacobi iteration as shown in Section 3.1 can be implemented as the function **mdm_simultaneous(g, tolerance)**.² In addition, the method with consecutive joint balancing distribution sequence and Gauss-Seidel iteration as shown in Section 3.2 can be implemented as the function **mdm_consecutive(g, tolerance)**.

²Because the moments are simultaneously read, the order in which joints are updated as dictated by the schedule does not impact the final solution.

```

1 # moment distribution method with simultaneous joint balancing and Jacobi schedule
2 def mdm_simultaneous(g, tolerance):
3     n = 0 # number of iterations performed
4     converged = False
5     while not converged:
6         converged = mdm_schedule(g, g, tolerance)
7         n += 1
8     return n
9
10 # moment distribution method with consecutive joint balancing and Gauss–Seidel schedule
11 def mdm_consecutive(g, tolerance):
12     n = 0 # number of iterations performed
13     converged = False
14     while not converged:
15         for i in g:
16             schedule = {i}
17             converged = mdm_schedule(g, schedule, tolerance)
18         n += 1
19     return n

```

Since the schedule used in `mdm_schedule(g, schedule, tolerance)` corresponds to the schedule set $V^{(k)}$ in Eq. 5.1, according to Theorem 2, this implementation always generates the same solution (within tolerance) as long as all joints are included in the schedule infinitely often.

6.2.3 Example

To illustrate, consider the same structure as shown in Chapter 3. Its matrix representations are

$$\begin{aligned}
 E &= \widehat{E}_a + \widehat{E}_b + \widehat{E}_c \\
 &= \begin{pmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{pmatrix} + \begin{pmatrix} 0 & \mathbf{-0.25} & \mathbf{-0.25} & 0 \\ 0 & \mathbf{-0.5} & \mathbf{-0.5} & 0 \\ 0 & \mathbf{-0.5} & \mathbf{-0.5} & 0 \\ 0 & \mathbf{-0.25} & \mathbf{-0.25} & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & \mathbf{0} \\ 0 & 0 & 0 & \mathbf{0} \\ 0 & 0 & 0 & \mathbf{-0.5} \\ 0 & 0 & 0 & \mathbf{-1} \end{pmatrix} \quad (6.1)
 \end{aligned}$$

where elements in bold are obtained from matrix E . The initial moments are

$$\mathcal{M}^{(0)} = \begin{pmatrix} m_{ab} \\ m_{ba} \\ m_{bc} \\ m_{cb} \end{pmatrix} = \begin{pmatrix} -172.8 \\ 115.2 \\ -416.7 \\ 416.7 \end{pmatrix}. \quad (6.2)$$

Assume only joint b is included in the first schedule. On one hand, the result from the matrix

form is

$$\mathcal{M}_1 = (I + \widehat{E}_b)\mathcal{M}^{(0)} = \begin{pmatrix} -97.4 \\ 266.0 \\ -266.0 \\ 492.1 \end{pmatrix}. \quad (6.3)$$

On the other hand, the computational implementation first calculates the unbalanced moment of joint b , which is

$$g[b][a].moment + g[b][b].moment + g[b][c].moment = 115.2 + 0 - 416.7 = -301.5, \quad (6.4)$$

in functions `get_unbalanced_moment(g, i)` by the lines

```

1 def get_unbalanced_moment(g, i):
2     unbalanced_moment = 0
3     for j in g[i]:
4         unbalanced_moment += g[i][j].moment
5     return unbalanced_moment

```

This unbalanced moment is then compared against the tolerance in `mdm_schedule(g, schedule, tolerance)` by the line

```

1 if abs(unbalanced_moment[joint]) >= tolerance: # stopping criteria

```

If it is greater than the tolerance, function `release(g, i, moment)` distributes this unbalanced moment to ends ba and bc , as well as calculate and record carryover moments for ends ab and cb . Because a dictionary data structure is used, the order of these updates is indeterminate, but the result is the same. Particularly, the function

```

1 def release(g, i, unbalanced_moment):
2     for j in g[i]:
3         my_share = g[i][j].distribution_factor * unbalanced_moment
4         g[i][j].moment -= my_share # mimic matrix D
5         g[j][i].moment -= g[i][j].carryover_factor * my_share # mimic matrix CD

```

first calculates

$$my_share = 0.5 \times (-301.5) = -150.8, \quad (6.5)$$

then it increases $m_{ba} = g[b][a].moment$ by 150.8, $m_{ab} = g[a][b].moment$ by $0.5 \times 150.8 = 75.4$, $m_{bc} = g[b][c].moment$ by 150.8, and $m_{cb} = g[c][b].moment$ by $0.5 \times 150.8 = 75.4$. This can be

expressed in matrix form as

$$\mathcal{M}_1 = \begin{pmatrix} m_{ab} + 75.4 \\ m_{ba} + 150.8 \\ m_{bc} + 150.8 \\ m_{cb} + 75.4 \end{pmatrix} = \begin{pmatrix} -172.8 + 75.4 \\ 115.2 + 150.8 \\ -416.7 + 150.8 \\ 416.7 + 75.4 \end{pmatrix} = \begin{pmatrix} -97.4 \\ 266.0 \\ -266.0 \\ 492.1 \end{pmatrix}. \quad (6.6)$$

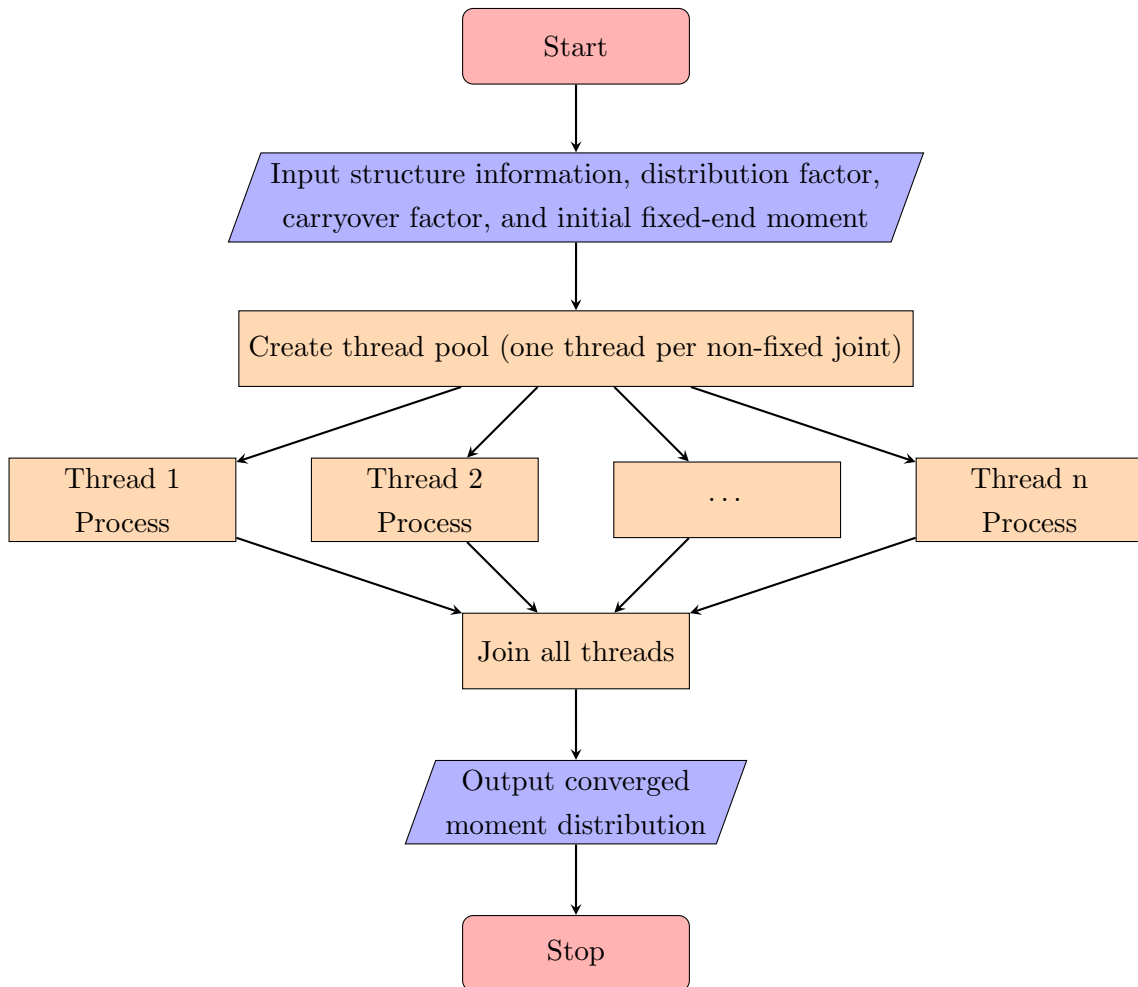
Hence, the computational implementation gives exactly the same result as the matrix form.

6.3 Asynchronous Shared Memory Parallel Implementation with Simple Termination

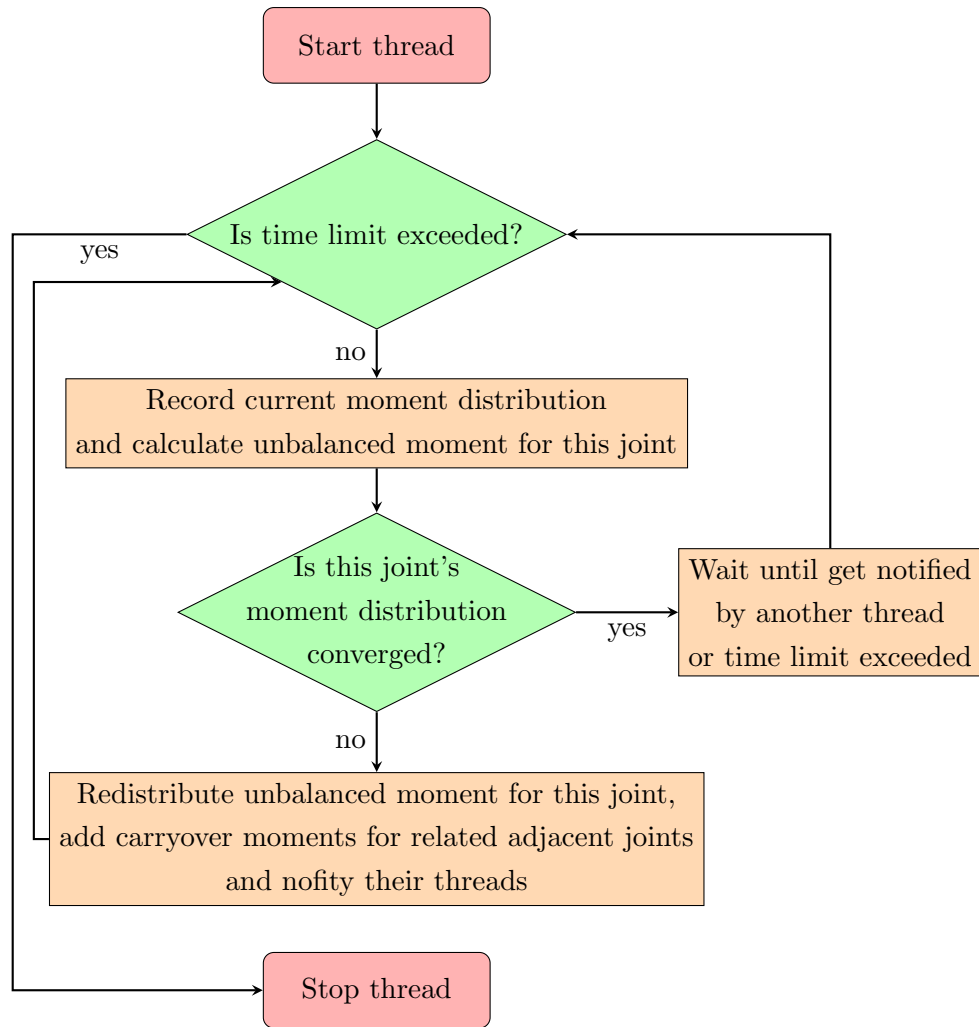
This section describes an asynchronous shared memory parallel implementation of the method that terminates after a period of time, with the assumption that global convergence is achieved by then.

6.3.1 Flow Chart

This asynchronous shared memory parallel implementation could be represented as the following flow chart.



The flow chart for the each thread process is



6.3.2 Python Code Snippets

For asynchronous shared memory parallel implementation, exactly one thread will be created for each non-fixed joint. The following function creates and starts these threads. The program exits after a predefined amount of time.

```
1 # moment distribution method using concurrent processes
2 def mdm_par(g, cond, tolerance):
3     for i in g:
```

```

4     if has_nonzero_dfs(g, i):
5         print('Starting', i)
6         t = Thread(target=joint_thread, args=(g, cond, i, tolerance))
7         t.daemon = True # make thread die when main thread completes
8         t.start()
9     sleep(1) # simple timed termination: keep main thread alive until processes are (assumed to be)
           ↪ done

```

The process of each thread is

```

1 # define a concurrent process for each joint i
2 def joint_thread(g, cond, i, tolerance):
3     while True:
4         with cond[i]: # require condition for wait/notification in Python
5             unbalanced_moment = get_unbalanced_moment(g, i)
6             while abs(unbalanced_moment) < tolerance: # local stopping criteria
7                 cond[i].wait()
8                 unbalanced_moment = get_unbalanced_moment(g, i)
9             release(g, cond, i, unbalanced_moment)

```

The statement

```

1 moment = get_unbalanced_moment(g, i)

```

takes a snapshot of the current member end moments, which will be used to create elementary operations.

Each thread keeps executing a slightly different **release** function compared to the one in sequential implementation until a global converge is achieved.

```

1 def release(g, cond, i, unbalanced_moment):
2     for j in g[i]:
3         my_share = g[i][j].distribution_factor * unbalanced_moment
4         g[i][j].decr_moment(my_share) # mimic matrix D, atomic
5         g[j][i].decr_moment(g[i][j].carryover_factor * my_share) # mimic matrix CD, atomic
6         with cond[j]: # notify neighboring joint that it may now have an unbalanced moment
7             cond[j].notify()

```

The wait-notify pattern in the **release** function can avoid thread busy spinning and make sure that every thread gets updated infinitely often. Also, instead of changing the moment directly, an atomic function **decr_moment** is used to avoid losing an update.

```

1 def decr_moment(self, moment):
2     # In Python, a lock is required to make this atomic.
3     # Use a different one for each moment to avoid limiting concurrency.
4     with self.lock:
5         self.moment -= moment

```

Since there is no synchronization between all threads, some threads could run faster than others. In fact, the statement

```
1 g[i][j].decr_moment(my_share) # mimic matrix D, atomic
```

exactly executes the elementary operation $[i, i]$, and the statement

```
1 g[j][i].decr_moment(g[i][j].carryover_factor * my_share) # mimic matrix CD, atomic
```

exactly executes the elementary operation $[j, i]$. Moreover, a thread must finish the loop

```
1 for j in g[i]:
```

before starting another one, which grants that the increments within a particular joint will be added to \mathcal{M} before releasing the same joint again. Thus, this asynchronous shared memory parallel implementation always provides same solution as the one in the sequential version (within tolerance) according to Theorem 2.

6.3.3 Example

To illustrate, consider the same structure as shown in Chapter 3 again. According to Eq. 5.16, the matrix E can be further decomposed in two dimensions as

$$E = [a, a] + [b, a] + [c, a] + [a, b] + [b, b] + [c, b] + [a, c] + [b, c] + [c, c] \quad (6.7)$$

where

$$[a, a] = \begin{pmatrix} \mathbf{0} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad (6.8)$$

$$[b, a] = \begin{pmatrix} 0 & 0 & 0 & 0 \\ \mathbf{0} & 0 & 0 & 0 \\ \mathbf{0} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad (6.9)$$

$$[c, a] = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \mathbf{0} & 0 & 0 & 0 \end{pmatrix} \quad (6.10)$$

$$[a, b] = \begin{pmatrix} 0 & \mathbf{-0.25} & \mathbf{-0.25} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad (6.11)$$

$$[b, b] = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & \mathbf{-0.5} & \mathbf{-0.5} & 0 \\ 0 & \mathbf{-0.5} & \mathbf{-0.5} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad (6.12)$$

$$[c, b] = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \mathbf{-0.25} & \mathbf{-0.25} & 0 \end{pmatrix} \quad (6.13)$$

$$[a, c] = \begin{pmatrix} 0 & 0 & 0 & \mathbf{0} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad (6.14)$$

$$[b, c] = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{0} \\ 0 & 0 & 0 & \mathbf{-0.5} \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad (6.15)$$

$$[c, c] = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{-1} \end{pmatrix} . \quad (6.16)$$

Elements in bold are obtained from matrix E . As expected, some of the elementary matrices are all zeros if two joints are not connected or at least one of them are fixed.

First consider the effects of releasing one joint. The three elementary operation matrices that are associated with releasing joint b are $[a, b]$, $[b, b]$, and $[c, b]$. When thread b executes the function `get_unbalanced_moment`, a snapshot, \mathcal{M}' , of the current member end moments is

created as

$$\mathcal{M}' = \mathcal{M}^{(0)} = \begin{pmatrix} -172.8 \\ 115.2 \\ -416.7 \\ 416.7 \end{pmatrix}. \quad (6.17)$$

Function **release** will generate increments of moments based on \mathcal{M}' . These increments are

$$[a, b]\mathcal{M}' = \begin{pmatrix} 75.4 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad (6.18)$$

$$[b, b]\mathcal{M}' = \begin{pmatrix} 0 \\ 150.8 \\ 150.8 \\ 0 \end{pmatrix} \quad (6.19)$$

$$[c, b]\mathcal{M}' = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 75.4 \end{pmatrix}. \quad (6.20)$$

They will be added to the current moments when

```

1 for j in g[i]:
2     g[i][j].decr_moment(my_share) # mimic matrix D, atomic
3     g[j][i].decr_moment(g[i][j].carryover_factor * my_share) # mimic matrix CD, atomic

```

is executed, which has the same effect as the one in the sequential implementation. No matter whether other threads modify the current moments during this process, the increments are determined when the snapshot is taken.

Here, we illustrate the process when multiple threads are involved. In addition to the three elementary operation matrices for joint b , joint c and its elementary operation matrices $[a, c]$, $[b, c]$, and $[c, c]$ are considered.

Assume the thread for joint c starts later than the thread for joint b . Say thread c executes the function **get_unbalanced_moment** after thread b takes a snapshot and adds

$$[c, b]\mathcal{M}' = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 75.4 \end{pmatrix} \quad (6.21)$$

to joint c but no other increments are added yet. Hence the snapshot thread c gets is

$$\mathcal{M}' = \begin{pmatrix} -172.8 \\ 115.2 \\ -416.7 \\ 492.1 \end{pmatrix}. \quad (6.22)$$

Function **release** from thread c generates increments of moments based on \mathcal{M}' . These increments are

$$[a, c]\mathcal{M}' = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad (6.23)$$

$$[b, c]\mathcal{M}' = \begin{pmatrix} 0 \\ 0 \\ -264.1 \\ 0 \end{pmatrix} \quad (6.24)$$

and

$$[c, c]\mathcal{M}' = \begin{pmatrix} 0 \\ 0 \\ 0 \\ -492.1 \end{pmatrix}. \quad (6.25)$$

They will be added to the current moments when

```

1 for j in g[i]:
2     g[i][j].decr_moment(my_share) # mimic matrix D, atomic
3     g[j][i].decr_moment(g[i][j].carryover_factor * my_share) # mimic matrix CD, atomic

```

is executed by thread c . Although the sequence of executions of these increments and the rest of increments from thread b is indeterminate, they will all be added eventually. These processes satisfy Theorem 3, so the program will converge to the same result regardless the sequence of thread executions.

Chapter 7

Termination Detection

Termination detection can be a nontrivial part of concurrent program design, especially for asynchronous ones. While the implementation in Section 6.3 uses a simple time-based termination condition, other approaches are possible. This section describes a few of them along with their applicability, mechanisms, and/or performance. Because complexity and implementation considerations vary widely between them, however, detailed evaluations are not given here.

7.1 Process Priorities

Some languages, like Java, allow the assignment of priorities to processes, while others, like Python, do not.¹ There are some important properties of a thread in an asynchronous implementation that uses no more than one thread per joint. Recall that one important fact about the method is that a joint should only be released at most once unless there are carryover moments passed from adjacent joints, as suggested by Theorem 1. Thus a thread will become idle when it reaches local convergence and it will not become active again unless notified by another thread.

One possible solution for the termination is via priorities of processes. A master thread is a thread that involves no actual computation but simply handles inputs and outputs of the program. Like threads in Section 6.3, a worker thread performs the calculations of the method and communicates with other worker threads. If the priority of the master thread is lower than the worker threads, the master thread yields to them while they are performing their computations. Once their work is complete and they become idle, however, the master thread is scheduled, which then produces the final output.

The implementation of this type of termination algorithm is relatively straightforward and

¹Python threads are intended to deal with I/O bound processes. Threads are managed by the host operating system, and a running thread holds the global interpreter lock until it reaches an I/O operation, at which point it releases the global interpreter lock and either it or another thread continues. So, there is no priority setting that could accomplish a graceful termination in Python.

any overhead largely depends on the language itself.

7.2 Quiescence Detection

Some high level concurrent packages have been developed in order to allow programmers to focus on solving actual problems instead of explicitly writing complicated code to terminate concurrent programs. Among them, PySy is a high level tool for convenient concurrent programming in Python, although it comes with a cost [Williamson and Olsson, 2014].

PySy is built on the basis of the SR and JR concurrent programming languages. By default, PySy programs do not terminate like normal Python programs. The idea behind the approach is *quiescence detection*, which allows the run-time system to automatically detect when all processes are idle, as well as determine if a program is deadlocked. Quiescence in a program is defined as if and only if no threads are runnable (sleeping threads are still considered runnable) and all network messages that have been sent have also been received. The availability of this language feature lessens the amount of the code needed to terminate the program, but it is traded off with performance degradation. Some experiments report that quiescence detection slows down PySy programs by a factor of 10 [Williamson,].

Generally, high level concurrent packages using quiescence detection, like PySy, provide a more sophisticated solution for concurrent termination than that in Section 7.1. Quiescence detection can handle deadlock, which includes but is not limited to the situations where all processes merely become idle. As a result, however, quiescence detection is more expensive and the performance hit may be unacceptable in some cases.

7.3 Snapshot of Global State

Maintaining a snapshot of global state is another approach for detecting global convergence. There are a few possible ways of maintaining the snapshot.

One way is using message passing and a helper process to keep track of process states. A helper process maintains a snapshot of global state and checks whether the program should exit or not. Each worker process communicates with the helper process by sending its status to the helper process and receiving a termination signal from the helper process. The overhead cost of this type of implementation is related to the mechanism and frequency of the checking in the helper process. Also, since all worker processes need to communicate with the helper process, performance and network bandwidth of the helper process may become a bottleneck.

Another way is to use global variables as a snapshot of global state, at least on shared memory architectures, which obviate the need for a helper process. However, each process is then responsible to check the global variables and make termination decisions.

Sometimes the implementation in either way could be tricky, mainly because the snapshot of global state needs the status of each process and communication. Performance of this approach relies on both the language itself and the design of the concurrent algorithm.

7.4 Generic Techniques for Distributed Computation

There are a few generic techniques can be used for distributed computation. They may have different preconditions and complexities. Programmers should justify their necessity before embedding these generic techniques in a program. This section introduces a relatively simple ring-based termination detection algorithm [Dijkstra et al., 1986].

7.4.1 Dijkstra's Ring-Based Termination Detection Algorithm

Dijkstra's ring-based termination detection algorithm is a suitable one to be used, mainly due to its simplicity and the nature of the Hardy Cross method. Recall that once a thread goes passive, it will not become active again unless it gets notified by another thread. This satisfies the precondition of the algorithm, thus a more complicated termination detection algorithm is not necessary. This algorithm can be described as follows:

1. A thread is active if it is performing a calculation and local convergence is not achieved.
2. A thread is passive if local convergence, which may be temporary, is achieved.
3. When all threads are passive, then program is in a stable state. Thus, termination should be detected and the program should exit.
4. All threads are linked as a ring and a token is propagated between them in order to detect termination.
5. There are six rules for initialization and propagation of the token. These six rules are

Rule 0. When active, thread $i + 1$ keeps the token; when passive, it hands over the token to thread i .

Rule 1. A thread sending a message makes itself black.

Rule 2. When thread $i + 1$ propagates the probe, it hands over a black token to thread i if it is black itself, whereas while being white it leaves the color of the token unchanged.

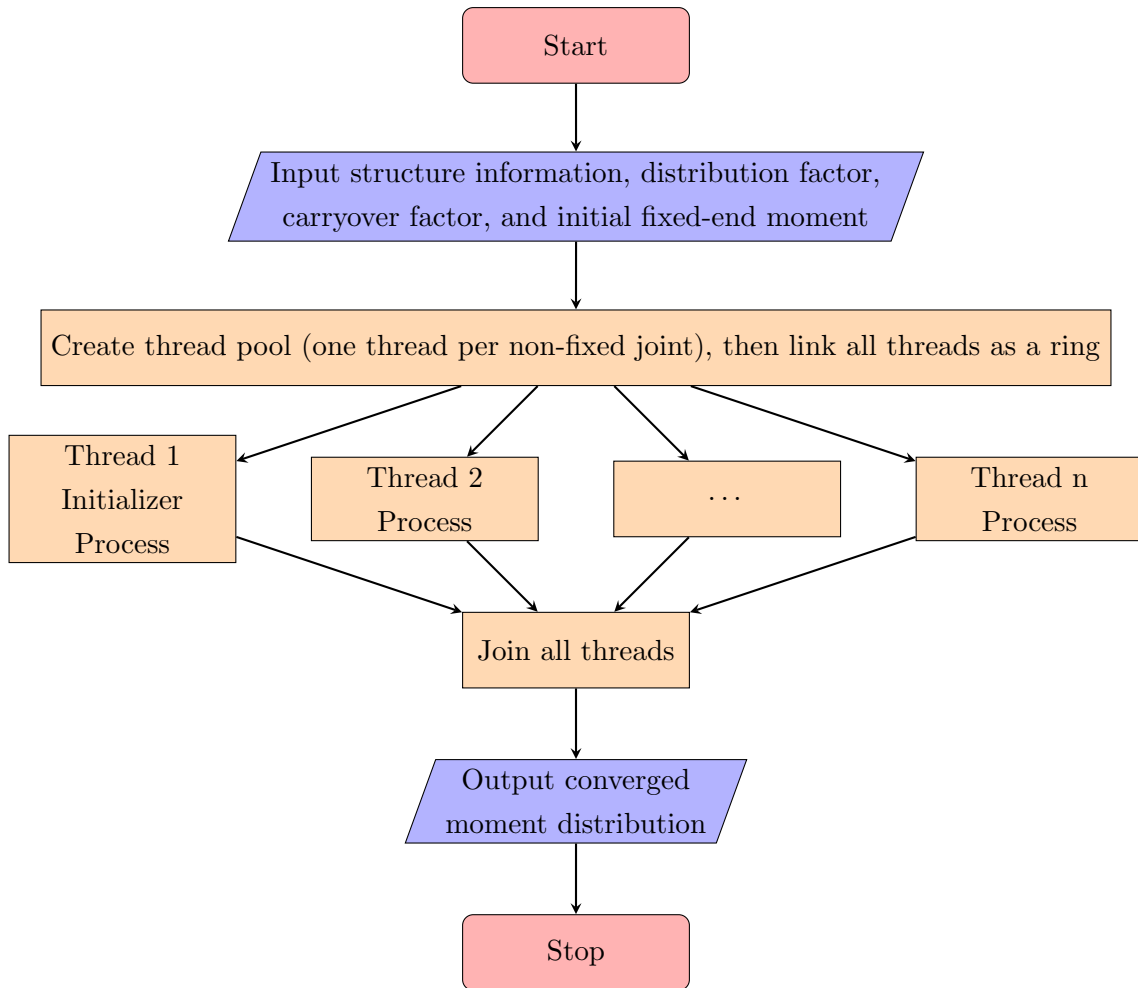
Rule 3. If thread 0 receives a white token, then all threads are passive and termination is detected. Otherwise, after the completion of an unsuccessful probe (thread 0 receives a black token), thread 0 initiates a next probe.

- Rule 4. Thread 0 initiates a probe by making itself white and sending a white token to thread $N - 1$.
- Rule 5. Upon transmission of the token to thread i , thread $i + 1$ becomes white. (Note that its original color may have influenced the color of the token.)

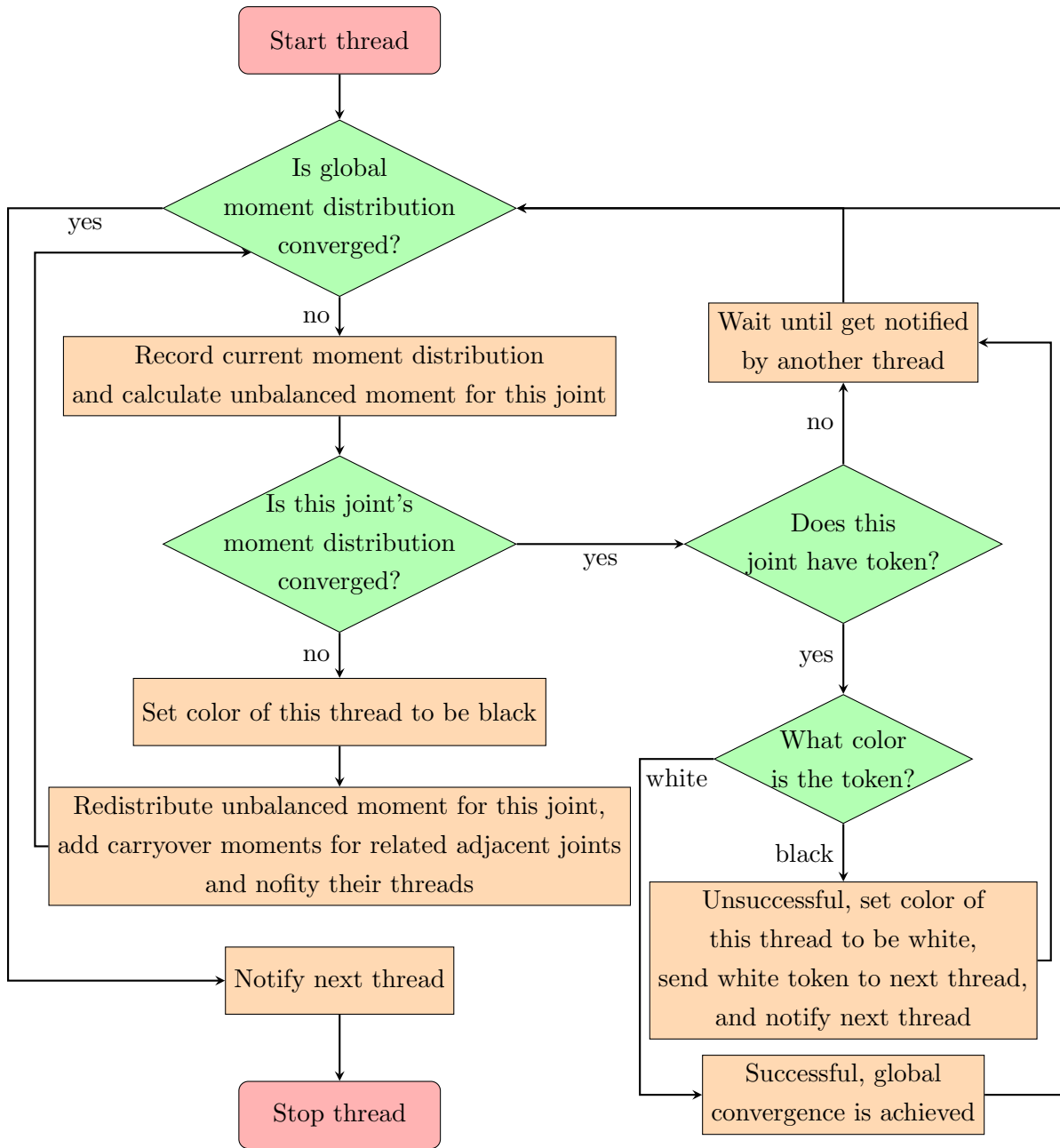
7.4.2 Asynchronous Shared Memory Parallel Implementation with Dijkstra's Termination Detection Algorithm

This section demonstrates one possible computational implementation of the method that adopts Dijkstra's ring-based termination detection algorithm for distributed computations. Instead of going through the implementation in detail, only flow charts are given. See Appendix B for the full code.

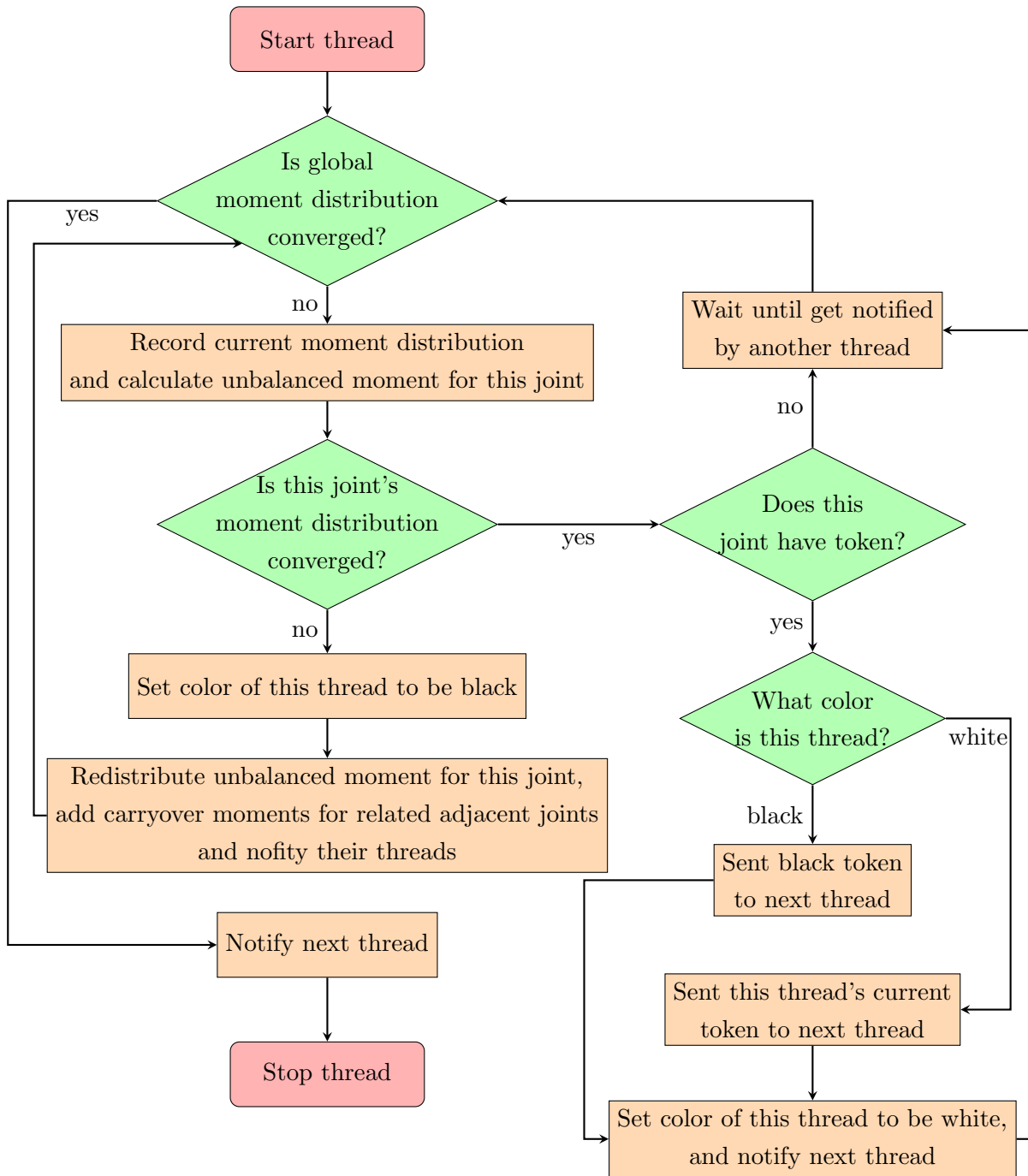
The flow chart for the master thread process is



The flow chart for the initializer thread process is



The flow chart for other threads is



Chapter 8

Conclusions

Engineers use simultaneous joint balancing and consecutive joint balancing distribution sequences interchangeably in the Hardy Cross method of moment distribution. However, distribution sequences are not limited to these two types. This thesis shows that if all joints get updated infinitely often, then alternative distribution sequences and iterations are equivalent. Such types of iteration are referred to as arbitrary sequence iteration in this thesis. Since both simultaneous joint balancing and consecutive joint balancing types are special cases of the arbitrary sequence iteration, and the convergence of both has been proved, arbitrary sequence iteration always converges. In other words, as long as all joints get updated infinitely often, the distribution sequences can be arbitrary and the Hardy Cross method converges to the same result for any particular problem statement.

In addition, inspired by an asynchronous parallel implementation of the Hardy Cross method and based on a concept of elementary operation, this thesis also introduces and proves an even more relaxed sufficient condition for the convergence of the method. Its corresponding iteration is referred to as elementary operation iteration. Similarly, this thesis proves that this more relaxed version always converges to the same result under any particular initial conditions, which can be helpful in giving a more general characterization of the Hardy Cross method. All these proofs are performed by working with the mathematical properties of their corresponding iterate matrices instead of starting with and relying on physical intuition.

Thus the characterization of the Hardy Cross method can be broadened, and computer implementations of the method can be more relaxed. Several possible computational implementations are given after the proofs. Some of these implementations are more relaxed than traditional ones that are based on Jacobi or Gauss-Seidel iterations. The relationship between these implementations and their mathematical representations is also explained, providing a basis for ensuring that the implementations converge.

Although this thesis demonstrates a more relaxed sufficient condition for the convergence of

the Hardy Cross method, whether it is a necessary condition remains a question, which could be investigated in the future.

REFERENCES

- [Black et al., a] Black, N., Moore, S., and Weisstein, E. W. Gauss-seidel method. From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/Gauss-SeidelMethod.html>, Last visited on 12/04/2014.
- [Black et al., b] Black, N., Moore, S., and Weisstein, E. W. Jacobi method. From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/JacobiMethod.html>, Last visited on 12/04/2014.
- [Bronshtein et al., 2007] Bronshtein, I. N., Semendyayev, K. A., Musiol, G., and Muehlig, H. (2007). *Handbook of mathematics*, volume 3. Springer.
- [Cross, 1930] Cross, H. (1930). Analysis of continuous frames by distributing fixed-end moments. *American Society of Civil Engineers Transactions*.
- [Cross and Morgan, 1932] Cross, H. and Morgan, N. D. (1932). *Continuous Frames of Reinforced Concrete*. John Wiley & Sons, Inc.
- [Dijkstra et al., 1986] Dijkstra, E. W., Feijen, W. H. J., and Van Gasteren, A. J. M. (1986). Derivation of a termination detection algorithm for distributed computations. In *Control Flow and Data Flow: concepts of distributed programming*, pages 507–512. Springer.
- [Dowell, 2009] Dowell, R. K. (2009). Closed-form moment solution for continuous beams and bridge structures. *Engineering structures*, 31(8):1880–1887.
- [Eaton, 2001] Eaton, L. K. (2001). Hardy Cross and the moment distribution method. *Nexus Network Journal*, 3(2):15–24.
- [Guo, 1987] Guo, Y. (1987). On a method of structural analysis. *Applied Mathematics and Mechanics*, 8(6):489–495.
- [Lopes et al., 2005] Lopes, A. P., Lopes, R. C., and de Castro, L. C. L. B. (2005). An exact solution for the Hardy Cross method through a matricial formulation applied to continuous beams. In *Proceedings of the COBEM 2005: 18th International Congress of Mechanical Engineering*.
- [McCormac, 1975] McCormac, J. C. (1975). *Structural Analysis*. Intext Educational Publishers, third edition.
- [Mozingo, 1968] Mozingo, R. R. (1968). Matrix distribution. *Journal of the Structural Division Proceedings of the American Society of Civil Engineers*, 94(ST4).
- [Volokh, 2002] Volokh, K. Y. (2002). On foundations of the Hardy Cross method. *International journal of solids and structures*, 39(16):4197–4200.
- [West, 1989] West, H. H. (1989). *Analysis of Structures: an Integration of Classical and Modern Methods*. John Wiley & Sons, Inc.

[Williamson,] Williamson, T. Introduction to PySy. <http://csiflabs.cs.ucdavis.edu/~twilliam/PySy/intro/intro.html>, Last visited on 03/01/2015.

[Williamson and Olsson, 2014] Williamson, T. and Olsson, R. A. (2014). PySy: a Python package for enhanced concurrent programming. *Concurrency and Computation: Practice and Experience*, 26(2):309–335.

APPENDICES

Appendix A

Equivalence of Jacobi and Gauss-Seidel Iterations Example

Consider the structure as shown in Figure A.1 again [West, 1989].

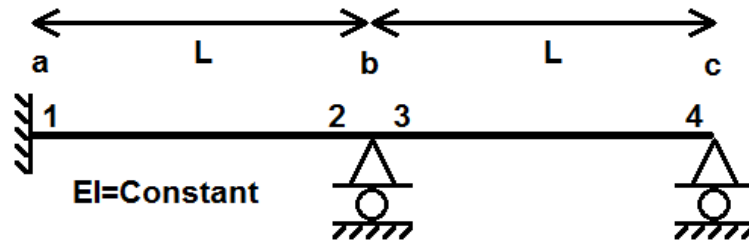


Figure A.1: Two Members and Three Joints Structure

Recall that the E matrix for this structure is

$$E = \begin{pmatrix} 0 & -0.25 & -0.25 & 0 \\ 0 & -0.5 & -0.5 & 0 \\ 0 & -0.5 & -0.5 & -0.5 \\ 0 & -0.25 & -0.25 & -1 \end{pmatrix},$$

which can be broken down to

$$\begin{aligned}
E &= \widehat{E}_a + \widehat{E}_b + \widehat{E}_c \\
&= \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & -0.25 & -0.25 & 0 \\ 0 & -0.5 & -0.5 & 0 \\ 0 & -0.5 & -0.5 & 0 \\ 0 & -0.25 & -0.25 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -0.5 \\ 0 & 0 & 0 & -1 \end{pmatrix}.
\end{aligned}$$

Then we have

$$\begin{aligned}
B_a = I + \widehat{E}_a &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
B_b = I + \widehat{E}_b &= \begin{pmatrix} 1 & -0.25 & -0.25 & 0 \\ 0 & 0.5 & -0.5 & 0 \\ 0 & -0.5 & 0.5 & 0 \\ 0 & -0.25 & -0.25 & 1 \end{pmatrix} \\
B_c = I + \widehat{E}_c &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -0.5 \\ 0 & 0 & 0 & 0 \end{pmatrix}.
\end{aligned}$$

Since $\widehat{E}_a^2 + \widehat{E}_a = 0$, $\widehat{E}_b^2 + \widehat{E}_b = 0$, $\widehat{E}_c^2 + \widehat{E}_c = 0$ and $B_a^2 = B_a$, $B_b^2 = B_b$, $B_c^2 = B_c$, both Eq. 4.28 and Eq. 4.29 are verified in this case.

Though *joint a* is not released, \widehat{E}_a and B_a will still be kept in equations in order to see patterns. For Jacobi iteration, the iterate matrix B_{Jacobi} would be

$$B_{Jacobi} = I + \widehat{E}_a + \widehat{E}_b + \widehat{E}_c. \quad (\text{A.1})$$

The limit of Eq. A.1 is

$$\begin{aligned}
B_{Jacobi}^\infty &= (I + \widehat{E}_a + \widehat{E}_b + \widehat{E}_c)(I + \widehat{E}_a + \widehat{E}_b + \widehat{E}_c) \dots \\
&= (I + \widehat{E}_a + \widehat{E}_b + \widehat{E}_c + \cancel{\widehat{E}_a} + \cancel{\widehat{E}_b} + \widehat{E}_b\widehat{E}_a + \widehat{E}_c\widehat{E}_a + \cancel{\widehat{E}_b} + \widehat{E}_a\widehat{E}_b + \cancel{\widehat{E}_c} + \widehat{E}_c\widehat{E}_b + \widehat{E}_c\widehat{E}_c + \widehat{E}_a\widehat{E}_c + \widehat{E}_b\widehat{E}_c) \dots \\
&= (I + \widehat{E}_a + \widehat{E}_b + \widehat{E}_c + \widehat{E}_b\widehat{E}_a + \widehat{E}_c\widehat{E}_a + \widehat{E}_a\widehat{E}_b + \widehat{E}_c\widehat{E}_b + \widehat{E}_a\widehat{E}_c + \widehat{E}_b\widehat{E}_c) \dots
\end{aligned}$$

For Gauss-Seidel iteration, assume the distribution sequence is *joint a* \rightarrow *joint b* \rightarrow *joint c*,

then B_{GS} would be

$$\begin{aligned} B_{GS} &= B_a B_b B_c \\ &= (I + \hat{E}_a)(I + \hat{E}_b)(I + \hat{E}_c) . \end{aligned} \quad (\text{A.2})$$

Limit of Eq. A.2 can be derived from $\sum_{k=1}^{\infty} \Delta G_k$ that

$$\begin{aligned} \Delta G_1 &= G_1 = I + \hat{E}_a \\ \Delta G_2 &= \hat{E}_b + \hat{E}_a \hat{E}_b \\ \Delta G_3 &= \hat{E}_c + \hat{E}_a \hat{E}_c + \hat{E}_b \hat{E}_c + \hat{E}_a \hat{E}_b \hat{E}_c \\ \Delta G_4 &= \cancel{\hat{E}_a} + \cancel{\hat{E}_a^2} + \hat{E}_b \hat{E}_a + \hat{E}_a \hat{E}_b \hat{E}_a \\ &\quad + \hat{E}_c \hat{E}_a + \hat{E}_a \hat{E}_c \hat{E}_a + \hat{E}_b \hat{E}_c \hat{E}_a + \hat{E}_a \hat{E}_b \hat{E}_c \hat{E}_a \\ \Delta G_5 &= \cancel{\hat{E}_b} + \cancel{\hat{E}_a \hat{E}_b} + \cancel{\hat{E}_b^2} + \cancel{\hat{E}_a \hat{E}_b^2} \\ &\quad + \hat{E}_c \hat{E}_b + \hat{E}_a \hat{E}_c \hat{E}_b + \hat{E}_b \hat{E}_c \hat{E}_b + \hat{E}_a \hat{E}_b \hat{E}_c \hat{E}_b \\ &\quad + \hat{E}_b \hat{E}_a \hat{E}_b + \hat{E}_a \hat{E}_b \hat{E}_a \hat{E}_b \\ &\quad + \hat{E}_c \hat{E}_a \hat{E}_b + \hat{E}_a \hat{E}_c \hat{E}_a \hat{E}_b + \hat{E}_b \hat{E}_c \hat{E}_a \hat{E}_b + \hat{E}_a \hat{E}_b \hat{E}_c \hat{E}_a \hat{E}_b \\ &\quad \vdots \end{aligned}$$

Hence the limits of both Eq. A.1 and Eq. A.2 can be expanded and simplified with Eq. 4.28 to the same

$$B_{Jacobi}^{\infty} = B_{GS}^{\infty} = I + \sum_{l_1} \hat{E}_{l_1} + \sum_{l_1} \sum_{l_2 \neq l_1} \hat{E}_{l_1} \hat{E}_{l_2} + \sum_{l_1} \sum_{l_2 \neq l_1} \sum_{l_3 \neq l_2} \hat{E}_{l_1} \hat{E}_{l_2} \hat{E}_{l_3} + \dots \quad (\text{A.3})$$

Appendix B

Python Implementation Source Code

Here provides Python implementation source code for both sequential implementation and asynchronous shared memory parallel implementation.

B.1 Sequential Implementation File

```
1  # moment distribution method using a graph representation
2  # sequential solver
3
4  #-----
5
6  # Example 16.5.1 from Analysis of Structures by H.H. West, p. 534
7  # ab -27.13  ba 406.54  bc -406.54  cb 00.00
8
9  # initial FEMs: {ab: -172.8, ba: 115.2, bc: -416.7, cb: 416.7}
10
11 def ex1():
12     a, b, c = 0, 1, 2
13     g = {a: {}, b: {}, c: {}}
14     g[a][b] = End(0.0, 0.5, -172.8)
15     g[b][a] = End(0.5, 0.5, 115.2)
16     g[b][c] = End(0.5, 0.5, -416.7)
17     g[c][b] = End(1.0, 0.5, 416.7)
18     return g
19
20 #-----
21
22 class End(object):
```

```

23     def __init__(self, d, c, m):
24         self.distribution_factor = d
25         self.carryover_factor = c
26         self.moment = m
27
28     def has_nonzero_dfs(g, i):
29         for j in g[i]:
30             if g[i][j].distribution_factor > 0:
31                 return True
32         return False
33
34     def get_unbalanced_moment(g, i):
35         unbalanced_moment = 0
36         for j in g[i]:
37             unbalanced_moment += g[i][j].moment
38         return unbalanced_moment
39
40     def release(g, i, unbalanced_moment):
41         for j in g[i]:
42             my_share = g[i][j].distribution_factor * unbalanced_moment
43             g[i][j].moment -= my_share # mimic matrix D
44             g[j][i].moment -= g[i][j].carryover_factor * my_share # mimic matrix CD
45
46     # moment distribution method with simultaneous joint balancing and Jacobi schedule
47     def mdm_simultaneous(g, tolerance):
48         n = 0 # number of iterations performed
49         converged = False
50         while not converged:
51             converged = mdm_schedule(g, g, tolerance)
52             n += 1
53         return n
54
55     # moment distribution method with consecutive joint balancing and Gauss-Seidel schedule
56     def mdm_consecutive(g, tolerance):
57         n = 0 # number of iterations performed
58         converged = False
59         while not converged:
60             for i in g:
61                 schedule = {i}
62                 converged = mdm_schedule(g, schedule, tolerance)
63             n += 1
64         return n
65
66     # moment distribution method with schedule
67     def mdm_schedule(g, schedule, tolerance):
68         converged = True

```

```

69 unbalanced_moment = {}
70 # prepare releasing a joint by calculating and recording its unbalanced moment
71 for joint in schedule:
72     if has_nonzero_dfs(g, joint):
73         unbalanced_moment[joint] = get_unbalanced_moment(g, joint)
74 # complete releasing a joint by redistributing its unbalanced moment
75 for joint in unbalanced_moment:
76     if abs(unbalanced_moment[joint]) >= tolerance: # stopping criteria
77         converged = False
78         release(g, joint, unbalanced_moment[joint])
79 return converged
80
81 # return a list of the end moments in g
82 def moments(g):
83     return [g[i][j].moment for i in g for j in g[i]]
84
85 if __name__ == '__main__':
86     g = ex1()
87     n = mdm_simultaneous(g, 0.001)
88     print(n, moments(g))
89
90     g = ex1()
91     n = mdm_consecutive(g, 0.001)
92     print(n, moments(g))

```

B.2 Asynchronous Shared Memory Parallel Implementation with Simple Termination File

```
1 # moment distribution method using a graph representation
2 # concurrent solver with threading module
3 # simple timed termination
4
5 from threading import Condition, Lock, Thread
6 from time import sleep
7
8 #-----
9
10 # Example 16.5.1 from Analysis of Structures by H.H. West, p. 534
11 # AB -27.13 BA 406.54 BC -406.54 CB 00.00
12
13 # initial FEMs: {ab: -172.8, ba: 115.2, bc: -416.7, cb: 416.7})
14
15 def ex1():
16     a, b, c = 0, 1, 2
17     g = {a: {}, b: {}, c: {}}
18     g[a][b] = End(0.0, 0.5, -172.8)
19     g[b][a] = End(0.5, 0.5, 115.2)
20     g[b][c] = End(0.5, 0.5, -416.7)
21     g[c][b] = End(1.0, 0.5, 416.7)
22
23     # let each joint have its own condition for process communication
24     # using the wait/notify pattern (to avoid busy-waiting, aka spinning)
25     joint_conds = {k: Condition(Lock()) for k in g.keys()}
26
27     return g, joint_conds
28
29 #-----
30
31 class End(object):
32
33     def __init__(self, d, c, m):
34         self.distribution_factor = d
35         self.carryover_factor = c
36         self.moment = m
37         self.lock = Lock() # <- used only to make decr_moment atomic
38
39     def decr_moment(self, moment):
40         # In Python, a lock is required to make this atomic.
41         # Use a different one for each moment to avoid limiting concurrency.
42         with self.lock:
```

```

43         self.moment -= moment
44
45     def has_nonzero_dfs(g, i):
46         for j in g[i]:
47             if g[i][j].distribution_factor > 0:
48                 return True
49         return False
50
51     def get_unbalanced_moment(g, i):
52         unbalanced_moment = 0
53         for j in g[i]:
54             unbalanced_moment += g[i][j].moment
55         return unbalanced_moment
56
57     def release(g, cond, i, unbalanced_moment):
58         for j in g[i]:
59             my_share = g[i][j].distribution_factor * unbalanced_moment
60             g[i][j].decr_moment(my_share) # mimic matrix D, atomic
61             g[j][i].decr_moment(g[i][j].carryover_factor * my_share) # mimic matrix CD, atomic
62             with cond[j]: # notify neighboring joint that it may now have an unbalanced moment
63                 cond[j].notify()
64
65     # define a concurrent process for each joint i
66     def joint_thread(g, cond, i, tolerance):
67         while True:
68             with cond[i]: # require condition for wait/notification in Python
69                 unbalanced_moment = get_unbalanced_moment(g, i)
70                 while abs(unbalanced_moment) < tolerance: # local stopping criteria
71                     cond[i].wait()
72                     unbalanced_moment = get_unbalanced_moment(g, i)
73                 release(g, cond, i, unbalanced_moment)
74
75     # moment distribution method using concurrent processes
76     def mdm_par(g, cond, tolerance):
77         for i in g:
78             if has_nonzero_dfs(g, i):
79                 print('Starting', i)
80                 t = Thread(target=joint_thread, args=(g, cond, i, tolerance))
81                 t.daemon = True # make thread die when main thread completes
82                 t.start()
83                 sleep(1) # simple timed termination: keep main thread alive until processes are (assumed to be)
84                 ↪ done
85
86     # return a list of the end moments in g
87     def moments(g):
88         return [g[i][j].moment for i in g for j in g[i]]

```

```
88
89 if __name__ == '__main__':
90     g, cond = ex1()
91     mdm_par(g, cond, 0.001)
92     print(moments(g))
```

B.3 Asynchronous Shared Memory Parallel Implementation with Dijkstra's Termination Detection Algorithm File

```
1 # moment distribution method using a graph representation
2 # concurrent solver with threading module
3 # termination using Dijkstra's termination detection algorithm
4
5 from threading import Condition, Lock, Thread
6
7 NONE = 0
8 WHITE = 1
9 BLACK = 2
10
11 #-----
12
13 # Example 16.5.1 from Analysis of Structures by H.H. West, p. 534
14 # AB -27.13 BA 406.54 BC -406.54 CB 00.00
15
16 # initial FEMs: {ab: -172.8, ba: 115.2, bc: -416.7, cb: 416.7}
17
18 def ex1():
19     a, b, c = 0, 1, 2
20     g = {a: {}, b: {}, c: {}}
21     g[a][b] = End(0.0, 0.5, -172.8)
22     g[b][a] = End(0.5, 0.5, 115.2)
23     g[b][c] = End(0.5, 0.5, -416.7)
24     g[c][b] = End(1.0, 0.5, 416.7)
25
26     # let each joint have its own condition for process communication
27     # using the wait/notify pattern (to avoid busy-waiting, aka spinning)
28     joint_conds = {k: Condition(Lock()) for k in g.keys()}
29
30     return g, joint_conds
31
32 #-----
33
34 class End(object):
35     def __init__(self, d, c, m):
36         self.distribution_factor = d
37         self.carryover_factor = c
38         self.moment = m
39         self.lock = Lock() # <- used only to make decr_moment atomic
40
41     def decr_moment(self, moment):
42         # In Python, a lock is required to make this atomic.
```

```

43     # Use a different one for each moment to avoid limiting concurrency.
44     with self.lock:
45         self.moment -= moment
46
47     def has_nonzero_dfs(g, i):
48         for j in g[i]:
49             if g[i][j].distribution_factor > 0:
50                 return True
51         return False
52
53     def get_unbalanced_moment(g, i):
54         unbalanced_moment = 0
55         for j in g[i]:
56             unbalanced_moment += g[i][j].moment
57         return unbalanced_moment
58
59     def release(g, cond, i, unbalanced_moment):
60         for j in g[i]:
61             my_share = g[i][j].distribution_factor * unbalanced_moment
62             g[i][j].decr_moment(my_share) # mimic matrix D, atomic
63             g[j][i].decr_moment(g[i][j].carryover_factor * my_share) # mimic matrix CD, atomic
64             with cond[j]: # notify neighboring joint that it may now have an unbalanced moment
65                 cond[j].notify()
66
67     # define a concurrent class for each joint i
68     class JointThread():
69         def __init__(self, condition):
70             self.token = NONE
71             self.color = BLACK
72             self.isInitializer = False
73             self.condition = condition
74             self.finish = False;
75
76         # notify joint global converge
77         def setFinish(self):
78             if not self.finish:
79                 self.finish = True
80                 self.nextJoint.setFinish()
81
82         # process for each thread
83         def run(self, g, i, tolerance):
84             while not self.finish:
85                 if abs(get_unbalanced_moment(g, i)) < tolerance: # passive
86                     if self.isInitializer:
87                         if self.token == BLACK:
88                             self.token = NONE # unsuccessful, start another probe, rule 3,4

```

```

89         self.color = WHITE
90         self.nextJoint.token = WHITE # pass token to next joint
91         with self.nextJoint.condition:
92             self.nextJoint.condition.notify()
93         elif self.token == WHITE:
94             self.setFinish() # notify all joints global converge
95             break
96         elif self.token != NONE:
97             tokenToBeSend = BLACK if self.color == BLACK else self.token # rule 2
98             self.token = NONE
99             self.color = WHITE # rule 5
100            self.nextJoint.token = tokenToBeSend # pass token to next joint
101            with self.nextJoint.condition:
102                self.nextJoint.condition.notify()
103            with self.condition: # require condition for wait/notification in Python
104                self.condition.wait()
105        else: # active
106            self.color = BLACK # rule 1
107            unbalanced_moment = get_unbalanced_moment(g, i)
108            release(g, cond, i, unbalanced_moment)
109        # finished, notify next joint
110        with self.nextJoint.condition:
111            self.nextJoint.condition.notify()
112
113    # moment distribution method using concurrent processes
114    def mdm_par(g, cond, tolerance):
115        needSetInit = True
116        pool = []
117        joints = []
118        for i in g:
119            if has_nonzero_dfs(g, i):
120                print('Starting', i)
121                joint = JointThread(cond[i])
122                if needSetInit:
123                    joint.isInitializer = True
124                    joint.token = BLACK
125                    needSetInit = False
126                else:
127                    joints[-1].nextJoint = joint # link all joints as a ring
128                    pool.append(Thread(target=joint.run, args=(g, i, tolerance)))
129                    joints.append(joint)
130            joints[-1].nextJoint = joints[0]
131
132        for t in pool:
133            t.start()
134        for t in pool:

```

```
135     t.join()
136     print("All_joined")
137
138     # return a list of the end moments in g
139     def moments(g):
140         return [g[i][j].moment for i in g for j in g[i]]
141
142     if __name__ == '__main__':
143         g, cond = ex1()
144         mdm_par(g, cond, 0.001)
145         print(moments(g))
```