Implementation of a Python Version of a Scaled Boundary Finite Element Method for Plate Bending Analysis

by

Lingfeng Chen

B.S. in Civil and Environmental Engineering
University of California, Berkeley, 2012

Submitted to the Department of Civil and Environmental Engineering Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Civil and Environmental Engineering
at the
Massachusetts Institute of Technology

June 2014

Signature of Author: Signature redacted
Department of Civil and Environmental Engineering
May 9, 2014

Certified by: Signature redacted
Jerome J. Connor
Professor of Civil and Environmental Engineering
Thesis Supervisor

Certified by: Signature redacted
Heidi M. Nepf
Chair, Departmental Committee for Graduate Students

Implementation of a Python Version of a Scaled Boundary Finite Element Method for Plate Bending Analysis

By

Lingfeng Chen

Submitted to the
Department of Civil and Environmental Engineering on May 9, 2014
In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Civil and Environmental Engineering

ABSTRACT

Common finite element programs for plate bending analysis are complicated and limited by the common plate theories. Such programs are usually not user-friendly for designers to implement. Lately, Hou Man et al. from the University of New South Wales has developed a technique for unified 3D plate bending analysis using scaled boundary finite element analysis, which promises to have little restrictions on plate thicknesses and faster convergence. This thesis presents an implementation of a python version of their method, which, when combined with a modeling program like Rhinoceros, aids designers in studying plate bending behavior under static loading. It represents a first step in the development of interactive programs for structural design and analysis of plates.

Thesis Supervisor: Jerome J. Connor
Title: Professor of Civil and Environmental Engineering

ACKNOWLEDGEMENT

First, I would like to express my gratitude to my thesis supervisor, Prof. Jerome J. Connor, who has provided precious guidance and advice on my thesis.
Second, I am grateful to Hou Man from University of New South Wales, who has generously helped with my understanding of the scaled boundary finite element method quoted in this thesis.
Third, I want to thank my parents for supporting my study at MIT, both financially and mentally.
Last but not least, I want to extend my gratitude to my instructors, my classmates and my friends for their supports.

Table of Contents

CHAPTER 1: INTRODUCTION

1.1 Background

Analysis of plate and shell structures is an important topic in structural engineering since plates are used everywhere in structures. A plate is a three dimensional object that is bounded by two plane surfaces and a cylindrical surface [1]. Plates transfer loads through bending. Many structural elements, such as floor slabs, gusset plates and bridge decks, can be modeled as plate structures. As computer aided structural analysis for plates has become more viable, plate elements are now used extensively in structures [1]. The exact solutions to the plate bending problems requires solving fourth order partial differential equations, which is difficult for complex geometries. Thus, numerical procedures such as finite element methods are used [2].

1.2 Existing Finite Element Methods

Compared to other several common finite element methods for plate bending analysis, an extended scaled boundary finite element method yields general solutions to plate bending problems while maintaining the balance between accuracy and efficiency. Common finite element methods developed for plate bending problems tend to discretize objects with 2D plate finite elements because using general 3D finite elements with high aspect ratio, or distorted shapes, would cause large approximation errors while it is computationally expensive to use lots of low-aspect-ratio 3D elements [2]. Although Mindlin/Ressiner theory can be applied to both thick and thin plates, the stiffness matrices require additional stabilization process when reducing shear locking phenomena it introduces. Shear locking phenomena happen when the elements become overly stiff as they become thinner and their shear stiffness dominates [2]. Mixed/hybrid formulation includes additional mechanical variables to be approximated independently from displacements; the p-version finite element method looks into eliminating shear locking problems with high-order elements, which uses greater numbers of element nodes in approximation [2]. Methods that are based on the approaches mentioned above are still not sufficient to yield a generalized solution,

partly because the plate theories introduced in the governing equations in formulations are not consistent with the 3D theory [2]. On the other hand, a pq-adaptive procedure for plate structures is developed using 3D finite elements, but it remains questionable whether the approach is both accurate and efficient [2]. Recently, Hou Man et al. from University of New South Wales in Australia has proposed a generalized solution for plate bending problems by extending the scaled boundary finite element method, which is a hybrid involving both the boundary element method and the finite element method [2]. The technique uses 3D governing equations to avoid shear locking issues, a scaled boundary finite element method to reduce the numbers of degrees of freedom, high-order spectral elements, and Kirchhoff plate theory to further reduce the computational expenses. The technique is promising in practice, but has not yet been implemented in any commercial finite element analysis program at this time.

1.3 Existing Design Tools

When it comes to plate bending analysis, tools either are too complicated in setting up models or lack user interactions for design purpose. Architectural design and modeling programs, such as AutoCAD and Rhinoceros, usually do not come with finite element analysis tools. Multi-purpose finite element analysis programs, such as ADINA and ANSYS, are complicated in creating finite elements and setting up meshes. Structural finite analysis programs, such as SAP 2000 and GSA, are not user friendly for structural design and modeling from scratch. Both types of finite element analysis programs usually complement architectural modeling programs for complete structural design and analysis which involves importing and exporting models of different types, which is inconvenient for designers who desire rapid feedbacks on structural performance of their design. Specialized programs, such as RAM Structural System, and plugins in architectural design and modeling programs help remove the bottleneck in design process as stated above, but they still impose restrictions on component types and dimensions on plate structures. Designers, especially who don't often acquire the knowledge of plate theories, are still looking for an interactive tool that can aid their design and analysis of plate structures simultaneously.

1.4 Objective and Scope

This thesis presents an implementation of Hou Man et al.'s method for plate bending analysis using the programing language Python. It also presents an example of how the method can be extended to existing modeling programs such as Rhinoceros, which would allow users to create interactive tools for structural design and analysis of plates.

CHAPTER 2: METHODOLOGY

2.1 Scaled Boundary Finite Element Method

The scaled boundary finite element method (SBFEM) is a novel computational procedure with a better balance between accuracy and efficiency than conventional numerical methods. The SBFEM bases its coordinate system on three directions: a radial direction and two local circumferential directions [3]. The boundary of the object is discretized with surface finite elements in circumferential directions [3]. The governing partial differential equations can then be transformed into ordinary differential equations in the radial coordinates, with coefficients approximated using the finite elements created. The ordinary differential equations can then be solved analytically [3]. Both attractive features of finite element and boundary element methods are inherent in the SBFEM. Like the finite element methods, the SBFEM requires no fundamental solution and it is not complicated to extend its applications to other spectrum [3]. No singular integrals are introduced and the results are always symmetric [3]. Like the boundary element methods, the SBFEM approximates the boundary elements only, reduces the special dimension by one and strictly satisfies boundary conditions infinitely away [3]. Thus, extending the SBFEM to solve plate bending problems would benefit the implementation in this thesis, by cutting down computational expense and increasing accuracy with semi-analytical approaches.

2.2 Unified 3D-based Technique for Plate Bending Analysis

With SBFEM, a 3D-based finite element can be performed on plate structures without enforcing thickness restrictions on plates. Hou Man et al. has developed such method to analyze bending stress of plates under static loading and within linear elastic range of material properties [2]. The method utilizes weak formulations and 3D governing equation, and work for both thick and thin plates. The method is not limited by the kinematics of Kirchhoff plate theory, which is only used to reduce the number of degrees of freedom and relate results to the conventional plate theory. Analytical results on transverse direction can be obtained through ordinary differential equations. The method also utilizes spectral element methods on in-plane dimensions to deal with elements

with curved boundaries and converge faster with fewer elements than conventional finite element methods. In the following sections, selected equations from Hou Man et al.'s method are presented [2] [4].

2.2.1 Isoparametric Formulation

Figure 1 shows a plate of constant thickness t. A Cartesian coordinate system is applied in the model: the Z-axis lies along the transverse direction of the plate; the X- and Y- axes lie parallel to the plane surfaces. An isoperimetric element is chosen to be a 2D rectangular plate. A Cartesian coordinate system is also applied to the element, where η- and ζ- lie in-plane at the bottom of the element surface and parallel to the edge of the element.

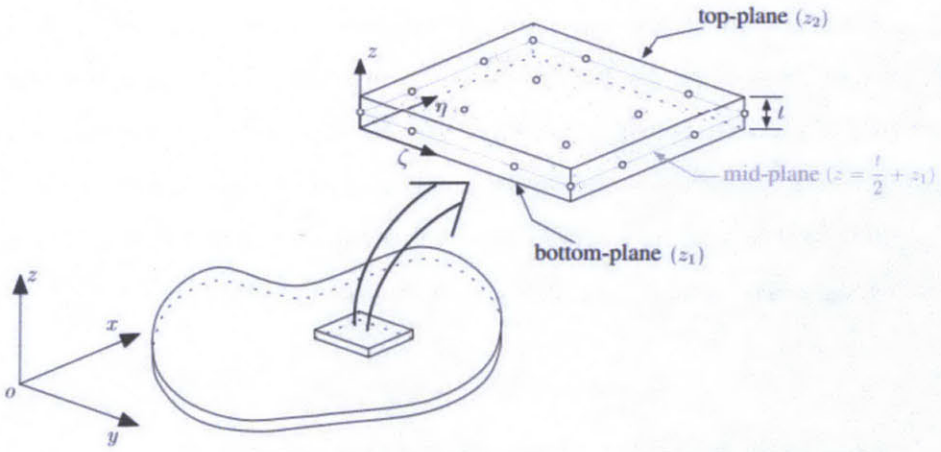

Figure 1 3D Scaled Boundary Coordinates with Scaling Center at Infinity [2]

$u_x = u_x(x, y, z)$, $u_y = u_y(x, y, z)$, $u_z = u_z(x, y, z)$ denote displacement components on the plate. $\{u\} = \{u(x, y, z)\} = [u_z, u_x, u_y]^T$ is the displacement vector. The strains $\{\varepsilon\} = \{\varepsilon(x, y, z)\}$ are arranged as

$$\{\varepsilon\} = \begin{bmatrix} \varepsilon_z \\ \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \\ \gamma_{yz} \\ \gamma_{xz} \end{bmatrix} = [L]\{u\} \tag{1}$$

With the differential operator

$$[L] = \begin{bmatrix} \frac{\partial}{\partial z} & 0 & 0 \\ 0 & \frac{\partial}{\partial x} & 0 \\ 0 & 0 & \frac{\partial}{\partial y} \\ 0 & \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} & 0 & \frac{\partial}{\partial z} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial z} & 0 \end{bmatrix} \tag{2}$$

From Hook's Law, the stresses $\{\sigma\} = \{\sigma(x, y, z)\}$ have linear relationships with $\{\varepsilon\}$

$$\{\sigma\} = \begin{bmatrix} \sigma_z \\ \sigma_x \\ \sigma_y \\ \tau_{xy} \\ \tau_{yz} \\ \tau_{xz} \end{bmatrix} = [C]\{\varepsilon\} \tag{3}$$

where $[C]$ is the elasticity matrix [5]

$$[C] = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1 & \frac{\nu}{(1-\nu)} & \frac{\nu}{(1-\nu)} & 0 & 0 & 0 \\ \frac{\nu}{(1-\nu)} & 1 & \frac{\nu}{(1-\nu)} & 0 & 0 & 0 \\ \frac{\nu}{(1-\nu)} & \frac{\nu}{(1-\nu)} & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} \end{bmatrix} \tag{4}$$

The in-plane dimensions of the plate are discretized and transformed to the isoparametric elements shown in Figure 1. Coordinates are interpolated using shape functions

$[N] = [N(\eta, \zeta)] = [N_1(\eta, \zeta), N_2(\eta, \zeta), \ldots]$ formulated in the coordinates η and ζ [2]

$$x(\eta,\zeta)=[N]\{x\} \tag{5a}$$

$$y(\eta,\zeta)=[N]\{y\} \tag{5b}$$

Only x and y coordinates are required to transform to and from η and ζ coordinates. The SBFEM here employs a scaling center at the infinity and translates the 2D mesh along z-direction. So

$$\begin{Bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{Bmatrix} = \frac{1}{|J|}\begin{bmatrix} y_{,\zeta} & -y_{,\eta} \\ -x_{,\zeta} & x_{,\eta} \end{bmatrix}\begin{Bmatrix} \frac{\partial}{\partial \eta} \\ \frac{\partial}{\partial \zeta} \end{Bmatrix} \tag{6}$$

where $|J|$ is the determinant of the Jacobean matrix

$$[J]=\begin{bmatrix} x_{,\eta} & x_{,\zeta} \\ y_{,\eta} & y_{,\zeta} \end{bmatrix} \tag{7}$$

$$|J| = x_{,\eta}\cdot y_{,\zeta} - x_{,\zeta}\cdot y_{,\eta} \tag{8}$$

An element in *[J]*, such as $x_{,\eta}$ can be determined as

$$x_{,\eta}=[N]_{,\eta}\{x\}=\left[N_{1,\eta}(\eta,\zeta),N_{2,\eta}(\eta,\zeta),\ldots\right]\begin{bmatrix} x_1 \\ x_2 \\ \vdots \end{bmatrix} \tag{9}$$

An infinitesimal area in the x-y plane

$$dS = dxdy = |J|d\eta d\zeta \tag{10}$$

From equations (2) and (7), the differential operator is now

$$[L]=\left[b^1\right]\frac{\partial}{\partial z}+\left[b^2\right]\frac{\partial}{\partial \eta}+\left[b^3\right]\frac{\partial}{\partial \zeta} \tag{11}$$

where

$$\left[b^1\right]=\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{12}$$

$$\left[b^2\right]=\frac{1}{|J|}\begin{bmatrix} 0 & 0 & 0 \\ 0 & y_{,\zeta} & 0 \\ 0 & 0 & -x_{,\zeta} \\ 0 & -x_{,\zeta} & y_{,\zeta} \\ -x_{,\zeta} & 0 & 0 \\ y_{,\zeta} & 0 & 0 \end{bmatrix} \tag{13}$$

$$\left[b^3\right]=\frac{1}{|J|}\begin{bmatrix} 0 & 0 & 0 \\ 0 & -y_{,\eta} & 0 \\ 0 & 0 & x_{,\eta} \\ 0 & x_{,\eta} & -y_{,\eta} \\ x_{,\eta} & 0 & 0 \\ -y_{,\eta} & 0 & 0 \end{bmatrix} \tag{14}$$

So the strain field can be expressed as

$$\{\varepsilon\}=\left[b^1\right]\{u\}_{,z}+\left[b^2\right]\{u\}_{,\eta}+\left[b^3\right]\{u\}_{,\zeta} \tag{15}$$

2.2.2 Scaled Boundary Finite Element Equations

The displacement field in the plate is represented semi-analytically in the z direction [2].

$$\{u\}=\begin{bmatrix} [N] & [0] & [0] \\ [0] & [N] & [0] \\ [0] & [0] & [N] \end{bmatrix}\begin{Bmatrix} \{u_z(z)\} \\ \{u_x(z)\} \\ \{u_y(z)\} \end{Bmatrix}\equiv[\mathbf{N}]\{u(z)\} \tag{16}$$

From equations (15) and (16), the strain field can be represented semi-analytically by displacement by *{u (z)}*

$$\{\varepsilon\}=\left[B^1\right]\{u(z)\}_{,z}+\left[B^2\right]\{u(z)\} \tag{17}$$

where

$$\left[B^1\right]=\left[b^1\right][\mathbf{N}] \tag{18a}$$

$$\left[B^2\right]=\left[b^2\right][\mathbf{N}]_{,\eta}+\left[b^3\right][\mathbf{N}]_{,\zeta} \tag{18b}$$

and the stress field is expressed as

$$\{\sigma\}=[C]\left(\left[B^1\right]\{u(z)\}_{,z}+\left[B^2\right]\{u(z)\}\right) \tag{19}$$

Hou Man et al.'s derivation shows that both conditions in the following need to be satisfied:

$$\{F(z_1)\}=-\{q(z_1)\} \tag{20a}$$

$$\{F(z_2)\}=\{q(z_2)\} \tag{20b}$$

and

$$\left[E^0\right]\left\{u(z)\right\}_{,zz}+\left(\left[E^1\right]^T-\left[E^1\right]\right)\left\{u(z)\right\}_{,z}-\left[E^2\right]\left\{u(z)\right\}=\{0\} \tag{21}$$

where *{F (z)}* is the external nodal force, *{q (z)}* is the internal nodal force expressed as

$$\left\{q(z)\right\}=\left[E^0\right]\left\{u(z)\right\}_{,z}+\left[E^1\right]^T\left\{u(z)\right\} \tag{22}$$

and *[E⁰], [E¹] and [E²]* denote the coefficient matrices

$$\left[E^0\right]=\int_S\left[B^1\right]^T[C]\left[B^1\right]|J|d\eta d\zeta \tag{23a}$$

$$\left[E^1\right]=\int_S\left[B^2\right]^T[C]\left[B^1\right]|J|d\eta d\zeta \tag{23b}$$

$$\left[E^2\right]=\int_S\left[B^2\right]^T[C]\left[B^2\right]|J|d\eta d\zeta \tag{23c}$$

Equation (21) is the SBFEM equation in displacement for plate structure. *[E⁰], [E¹]* and *[E²]* are to be assembled using conventional finite element method.

2.2.3 High-Order Spectral Element

Hou Man et al.'s method discretizes the in-plane dimensions of the plate structure with high-order spectral elements [2]. The 2D shape function [N] in the previous section is determined as the product of two 1D shape functions with respect to local η and ζ coordinates, respectively. The two 1D shape functions are defined similarly. With the local coordinate, take η as example, in a range of $-1 \leq \eta \leq 1$, a 1D element of order p has p+1 nodes. The nodal locations are determined using Legendre Polynomials Ln. Ln is defined by the recurrence relation [6]

$$\begin{cases} L_0(\eta) = 1 \\ L_1(\eta) = \eta \\ L_{n+1}(\eta) = \frac{2n+1}{n+1}\eta L_n(\eta)-\frac{n}{n+1}L_{n-1}(\eta) \end{cases} \tag{24}$$

Except for the nodes on the two end, the *p-1* nodes in between are located at the Gauss-Lobatto-Legendre points, which are the roots of

$$L'_p(\eta)=0 \tag{25}$$

where L'_p is the derivative of L_p [2]. The 1D shape function is Lagrange polynomials

$$N_i(\eta)=\prod_{k=1,k\neq 1}^{p+1}\frac{\eta-\eta_k}{\eta_i-\eta_k} \tag{26}$$

and it exhibits such property at the node η_j

$$N_i(\eta_j) = \delta_{ij} \tag{27}$$

where δ is the Kronecker delta

$$\delta_{ij} = \begin{cases} =1 & (i=j) \\ =0 & (i \neq j) \end{cases} \tag{28}$$

The integration points happen to be at the nodes when Gauss-Lobatto-Legendre quadrature is used for numerical integration, so the integration can be approximated with respect to individual node η_i [7]

$$\int_{-1}^{1} F(\eta) d\eta \approx \sum_{i}^{p+1} w_i F(\eta_i) \tag{29}$$

The weight w_i at the point η_i is

$$w_i = \frac{2}{p(p+1)\left(L_p(\eta_i)\right)^2} \tag{30}$$

So, for 2D integration, the shape function is obtained by

$$N_i(\eta,\zeta) = N_k(\eta) N_l(\zeta) \tag{31}$$

where the local nodal number i is obtained from the nodal numbers k and l of the two 1D shape functions

$$i = (l-1)(p+1)+k \tag{32}$$

Also, the 2D shape function exhibits such property

$$N_i(\eta_\alpha, \zeta_\beta) = \delta_{k\alpha}\delta_{l\beta} = \delta_{i\omega} \tag{33}$$

where

$$\omega = (\beta-1)(p+1)+\alpha \tag{34}$$

The weight can be obtained as [4]

$$w_i = w_k w_l \tag{35}$$

The 3 × 3 submatrix of $[E^0]$ related to the ith and jth nodes is simplified as

$$\left[E^0\right]_{ij} = \delta_{ij} w_i \left(\left[b^1\right]^T [C]\left[b^1\right]\right)|J| \tag{36}$$

with

$$\left[b^1\right]^T[C]\left[b^1\right]=G\begin{bmatrix}\frac{2(1-\nu)}{(1-2\nu)} & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix} \tag{37}$$

where G is the shear modulus

$$G=\frac{E}{2(1+\nu)} \tag{38}$$

Note that $[E^o]$ only consists of positive diagonal blocks.

Similarly, the 3 × 3 submatrix of $[E^1]$ can be expressed as

$$\left[E^1\right]_{ij}=w_j\left[B_i^2\right]^T[C]\left[b^1\right]|J| \tag{39}$$

The 3 × 3 submatrix of $[E^2]$ has to be evaluated element-by-element

$$\left[E^2\right]_{ij}=\sum_{\alpha=1}^{p+1}\sum_{\beta=1}^{p+1}w_\alpha w_\beta\left(\left[B_i^2\left(\eta_\alpha,\zeta_\beta\right)\right]^T[C]\left[B_j^2\left(\eta_\alpha,\zeta_\beta\right)\right]\right)|J| \tag{40}$$

2.2.4 Solution Custom for Thin to Moderately Thick Plate

Hou Man et al. have developed a general solution to the previous SBFEM equations for general 3D plate structures [2]. They have also obtained a tailored solution for thin plates that utilizes the kinematics of the Kirchhoff plate theory to reduce the number of degrees of freedom and simplify the results, where displacements and force resultants of the plate can be obtained directly [4]. The tailored method is later implemented in this thesis.

General Solution for 3D Plate Structure

By introducing the variable [2]

$$\left\{X(z)\right\}=\begin{Bmatrix}\left\{u(z)\right\}\\ \left\{q(z)\right\}\end{Bmatrix} \tag{41}$$

the SBFEM equation (21) can be transformed as

$$\left\{X(z)\right\}_{,z}=-[Z]\left\{X(z)\right\} \tag{42}$$

with the coefficient matrix *[Z]* as

$$[Z]=\begin{bmatrix} \left[E^0\right]^{-1}\left[E^1\right]^T & -\left[E^0\right]^{-1} \\ -\left[E^2\right]+\left[E^1\right]\left[E^0\right]^{-1}\left[E^1\right]^T & -\left[E^1\right]\left[E^0\right]^{-1} \end{bmatrix} \tag{43}$$

The general solution to *{X (z)}* is in the form

$$\left\{X(z)\right\}=e^{-[Z]z}\left\{c\right\} \tag{44}$$

where *{c}* are the integration constants and $e^{-[Z]z}$ is a matrix exponent, which can be approximated using Taylor series

$$e^{-[Z]z}=\sum_{i=0}^{\infty}\frac{\left(-[Z]z\right)^i}{i!} \tag{45}$$

or Padé expansion, which converges more rapidly to the solution [4]

$$e^{-[Z]z}\approx\left[Q(z)\right]^{-1}\left[P(z)\right] \tag{46}$$

where *[P (z)]* and *[Q (z)]* are polynomials of *–[Z]z*.

When substitute in $z_1 = 0$ for the bottom surface and $z_2 = t$ for the top surface, the SBFEM equation (42) can be rewritten as [2]

$$[S]\begin{Bmatrix}\{u_B\}\\ \{u_T\}\end{Bmatrix}=\begin{Bmatrix}\{F_B\}\\ \{F_T\}\end{Bmatrix} \tag{47}$$

with the stiffness matrix *[S]*

$$[S]=\begin{bmatrix} [\psi_{12}]^{-1}[\psi_{11}] & -[\psi_{12}]^{-1} \\ [\psi_{21}]-[\psi_{22}][\psi_{12}]^{-1}[\psi_{11}] & [\psi_{22}][\psi_{12}]^{-1} \end{bmatrix} \tag{48}$$

where

$$[\psi]=e^{-[Z]t}=\begin{bmatrix}[\psi_{11}] & [\psi_{12}]\\ [\psi_{21}] & [\psi_{22}]\end{bmatrix} \tag{49}$$

The subscripts B and T denotes the bottom surface and the top surface respectively. *[ψ]* is approximated using either Taylor series as equation (45) shows or Padé expansion as equation (46) shows.

Tailored Solution Using Kinematics of Plate Theory

Hou Man et al. use Padé expansion of order (2, 2) in their research since this is accurate enough for thin to moderately thick plates; when "$t/L << 1$, where L is shortest span of the plate, higher order terms of $-[Z]z$ become negligible" [4]. In Padé expansion of order (2, 2), $[P(z)]$ and $[Q(z)]$ in equation (46) are picked as below

$$[P(z)]=[I]-\frac{1}{2}z[Z]+z^2[V] \tag{50}$$

$$[Q(z)]=[I]+\frac{1}{2}z[Z]+z^2[V] \tag{51}$$

with

$$[V]=\frac{1}{12}[Z]^2 \tag{52}$$

By applying a transformation matrix $[T]$ and kinematics of plate theory, the number of degrees of freedom in the SBFEM equation (47) can be further reduced.

With

$$[T]=\begin{bmatrix} -\frac{t}{2}[I] & [I] \\ \frac{t}{2}[I] & [I] \end{bmatrix} \tag{53}$$

the displacement vector in SBFEM equation (47) can be transformed

$$\begin{Bmatrix} \{u_B\} \\ \{u_T\} \end{Bmatrix}=[T]\begin{Bmatrix} \frac{1}{t}\left(\{u_T\}-\{u_B\}\right) \\ \frac{1}{2}\left(\{u_T\}+\{u_B\}\right) \end{Bmatrix}\equiv[T]\begin{Bmatrix} \{\theta\} \\ \{\bar{u}\} \end{Bmatrix} \tag{54}$$

where $\{\theta\}$ means the rate of change of displacement over the thickness and $\{\bar{u}\}$ means the average of displacements on the top and bottom surfaces. And with

$$[T]^{-T}=\begin{bmatrix} -\frac{1}{t}[I] & \frac{1}{2}[I] \\ \frac{1}{t}[I] & \frac{1}{2}[I] \end{bmatrix} \tag{55}$$

the forces in the SBFEM equation (47) can be transformed

$$\begin{Bmatrix}\{F_B\}\\ \{F_T\}\end{Bmatrix}=[T]^{-T}\begin{Bmatrix}\frac{t}{2}\left(\{F_T\}-\{F_B\}\right)\\ \{F_T\}+\{F_B\}\end{Bmatrix}\equiv[T]^{-T}\begin{Bmatrix}\{M\}\\ \{\bar{F}\}\end{Bmatrix} \tag{56}$$

where $\{M\}$ is the difference of the forces of the surfaces multiplying half of the thickness and $\{\bar{F}\}$ means the average of the forces on the surfaces.

For $t/L << 1$, using the transformations (54) and (56), the SBFEM equation (47) can be simplified as

$$[s]\begin{Bmatrix}\{\theta\}\\ \{\bar{u}\}\end{Bmatrix}=\begin{Bmatrix}\{M\}\\ \{\bar{F}\}\end{Bmatrix} \tag{57}$$

with the stiffness matrix

$$[s]=t\begin{bmatrix}[E^0]\left([I]+t^2[V_{11}]\right) & [E^1]^T\\ [E^1]\left([I]+t^2[V_{11}]\right)-t^2[V_{21}] & [E^2]\end{bmatrix} \tag{58}$$

When applying the kinematics of plate theory, it is assumed that the out-of-plane deformations are symmetric with respect to the mid-plane

$$\{u_{Tz}\}=\{u_{Bz}\}=\{\bar{u}_z\} \tag{59a}$$

And the in-plane deformations $\{u_x\}$ and $\{u_y\}$ are equal and opposite for the top and the bottom surfaces

$$\{u_{Tx}\}=-\{u_{Bx}\}=\{\bar{u}_x\} \tag{59b}$$

$$\{u_{Ty}\}=-\{u_{By}\}=\{\bar{u}_y\} \tag{59c}$$

With the assumptions (59), the SBFEM equation (57) can be rewritten as

$$\begin{bmatrix}[s_{11}] & [s_{12}] & [s_{13}] & [s_{14}] & [s_{15}] & [s_{16}]\\ [s_{21}] & [s_{22}] & [s_{23}] & [s_{24}] & [s_{25}] & [s_{26}]\\ [s_{31}] & [s_{32}] & [s_{33}] & [s_{34}] & [s_{35}] & [s_{36}]\\ [s_{41}] & [s_{42}] & [s_{43}] & [s_{44}] & [s_{45}] & [s_{46}]\\ [s_{51}] & [s_{52}] & [s_{53}] & [s_{54}] & [s_{55}] & [s_{56}]\\ [s_{61}] & [s_{62}] & [s_{63}] & [s_{64}] & [s_{65}] & [s_{66}]\end{bmatrix}\begin{Bmatrix}\{0\}\\ \frac{2}{t}\{u_x\}\\ \frac{2}{t}\{u_y\}\\ \{u_z\}\\ \{0\}\\ \{0\}\end{Bmatrix}=\begin{Bmatrix}\frac{t}{2}\left(\{F_{Tz}\}-\{F_{Bz}\}\right)\\ \frac{t}{2}\left(\{F_{Tx}\}-\{F_{Bx}\}\right)\\ \frac{t}{2}\left(\{F_{Ty}\}-\{F_{By}\}\right)\\ \{F_{Tz}\}+\{F_{Bz}\}\\ \{F_{Tx}\}+\{F_{Bx}\}\\ \{F_{Ty}\}+\{F_{By}\}\end{Bmatrix} \tag{60}$$

Eliminating the rows and columns with zero entries in the displacement vector, the SBFEM equation (60) becomes

$$\begin{bmatrix} [s_{22}] & [s_{23}] & [s_{24}] \\ [s_{32}] & [s_{33}] & [s_{34}] \\ [s_{42}] & [s_{43}] & [s_{44}] \end{bmatrix} \begin{Bmatrix} \{\theta_y\} \\ \{\theta_x\} \\ \{w\} \end{Bmatrix} = \begin{Bmatrix} \{M_x\} \\ \{M_y\} \\ \{\overline{F}_z\} \end{Bmatrix} \tag{61}$$

where the matrix on the left hand side is the submatrix of the stiffness matrix *[s]* defined in equation (58), θ_y and θ_x are rotations about the in-plane axis, and w is the out-of-plane deflection, or

$$\begin{Bmatrix} \{\theta_y\} \\ \{\theta_x\} \\ \{w\} \end{Bmatrix} = \begin{Bmatrix} \frac{2}{t}\{u_x\} \\ \frac{2}{t}\{u_y\} \\ \{u_z\} \end{Bmatrix} \tag{62}$$

The stiffness matrix *[s]* defined in equation (58) can be assembled using submatrices of *[V]* defined in equation (52) and submatrices of *[Z]* defined in equation (43).

$$[V_{21}] = \frac{1}{12}\left([Z_{11}]^2 + [Z_{12}][Z_{21}]\right) \tag{63}$$

$$[V_{21}] = \frac{1}{12}\left([Z_{21}][Z_{11}] - [Z_{11}]^T[Z_{21}]\right) \tag{64}$$

$$[Z_{11}] = [E^0]^{-1}[E^1]^T \tag{65}$$

$$[Z_{12}] = -[E^0]^{-1} \tag{66}$$

$$[Z_{21}] = -[E^2] + [E^1][E^0]^{-1}[E^1]^T \tag{67}$$

CHAPTER 3: PYTHON IMPLEMENTATION

3.1 Python, Rhinoceros and Grasshopper

Based on Hou Man et al.'s method presented in the sections above, the implementation in this thesis uses the programing language Python. Python is a general purpose, object-oriented scripting language that enforces the readability of codes [8]. Python can run on lots of different platforms. It is free, even for commercial purposes [8]. The syntax is simple and high-level data structures can be used [8]. A large amount of extension modules maintained by members of Python user community can be installed to expand the scope of the application of Python. Many programs could include a Python script editor and compiler to interpret written codes and provide customizable functions. Thus, designers with or without programing background would embrace Python for its simplicity and powerful capability.

Rhinoceros and grasshopper, prevalent tools in the design industry, can incorporate Python scripts to allow desired functions. Rhinoceros is a commercial computer-aided design program aimed to provide better user experience in 3D modeling. Grasshopper, a plugin of Rhinoceros, is a graphical algorithm editor [9]. Users may create automated work flows by dragging Grasshopper components into the Grasshopper canvas and connecting them. The Grasshopper components may prompt inputs, perform arithmetic, implement commands in Rhinoceros, or display outputs. Users may find the combination of Rhinoceros and Grasshopper convenient with form generating, basic structural analysis and parametric modeling, which enables design options generated from initial parameters. Both Rhinoceros and Grasshopper have numerous plugins users can find to extend the usage. One of the Grasshopper plugins, GhPython allows user-defined Grasshopper components using Python scripts [10]. With the aforementioned tools, the Python implementation of the SBFEM can be compiled, or interpreted and run, in the modeling program for rapid feedback on plate bending performance.

3.2 Other Utilized tools

Besides Rhinoceros, Grasshopper and GhPython, some other non-built-in tools are also used in the example of Python implementation below. NumPy is a fundamental package for scientific computing with Python [11] [12]. It provide modules that can create array objects of N-dimensions and manipulate array objects resembling matrices with matrix operations. "utility.py" is a script developed to contain handy functions for users of GhPython to use [13], functions including unifying types of inconsistent Rhinoceros geometry objects. These tools, although not mandatory for the implementation, would expedite the Python script writing in GhPython.

3.3 Grasshopper Canvas

Besides the Python script writing, the example below uses Grasshopper components to interact with users. An example of Grasshopper definition is shown in Figure 2. Each link in between components transfers the data from one output of the component on the left end of the link to the input of that on the right end of the link. Titles of the components can be renamed for ease of readability. On the left hand side of Figure 2, number sliders and panels are Grasshopper components permitting users to input numbers. "Element Nodes" is a point geometry component that can set and collect points in Rhinoceros modeling world. Analogously, "Element Edges" and "Integration Points" are geometry components that collect the output lines and points. The white color shows that the previews of the component are turned on and they are drawn in Rhinoceros modeling world. "Python Lglnodes" and "Python Integration" are customized components created through GhPython. The GhPython components can store Python scripts and compile them. They also allows adding or removing inputs or outputs with editable names.

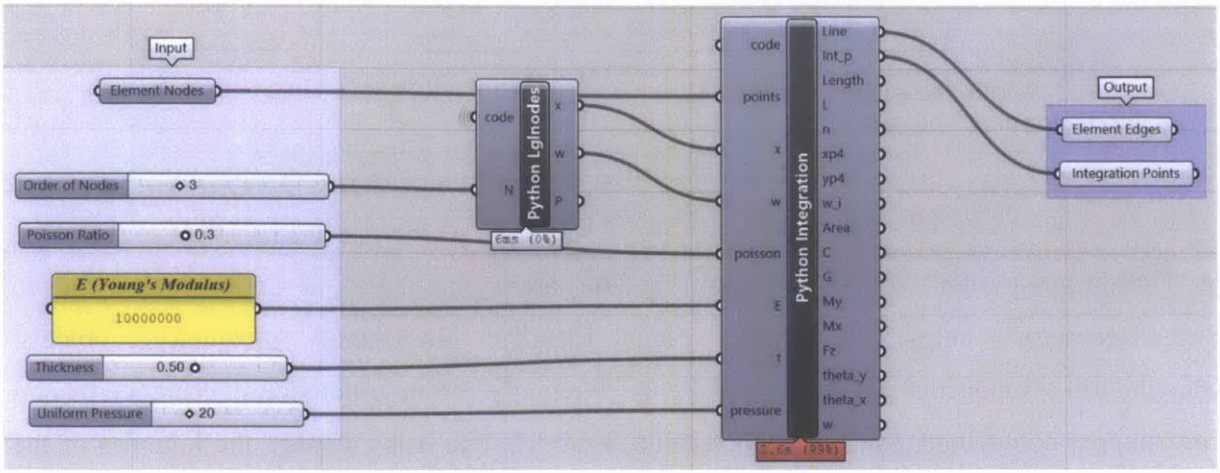

Figure 2 Example Grasshopper Definition with GhPython Components

The example built here would allow people to compute the bending moments and transverse deflections of a rectangular plate element with proper boundary conditions and loading inputs. "Element Nodes" takes in four vertexes of the rectangular plate element either clockwise or counterclockwise. "Python Lglnodes" takes in the number of node order to generate the positions and the respective integration weights of integration nodes of a 1D spectral element. "Python Integration" then receives the existing information and generates the corresponding integration nodes on the elements and the stiffness matrix. With users manually input the boundary conditions and loading information, "Python Integration" can then approximate the bending moments and transverse deflections of each node. The following sections explain the functions of different parts of the codes.

3.4 Python Codes

3.4.1 Import

In both "Python Lglnodes" and "Python Integration" components shown in Figure 2, the module NumPy is imported at the beginning and renamed as "np" temporarily for shorthand as Code Block 1 shows. Due to some incompatibility issues at this time, a built in module called "clr" is imported and add reference to "mtrand" to help the compiler import NumPy smoothly [12].

```
import clr
clr.AddReference("mtrand")
import numpy as np
```

Code Block 1 Importing the Module NumPy

In "Python Integration", some other useful modules are imported also, as Code Block 2 shows. Rhino.Geometry helps to create and identify Rhinoceros geometry objects. Ghpythonlib.components and rhinoscriptsyntax contain python interpretation of Rhinoceros and Grasshopper commands. Sys is a Python built-in module that helps manage the logistics of the compiler.

```
import sys
import utility
import ghpythonlib.components as ghc
import rhinoscriptsyntax as rs
import Rhino.Geometry as rg
```

Code Block 2 Importing Modules other than NumPy

Code Block 3 in "Python Integration" pre-processes its inputs. The points taken in are forced to be converted to general Rhinoceros point geometry objects. "xp", "yp" and "zp" are the x, y, z coordinates of the points in Rhinoceros modeling space. The positions and the integration weights of the 1D element nodes are converted to row vectors. "n" denotes the number of 1D element nodes.

```
##Import
points=utility.coerce3dpointlist(points)
xp,yp,zp=ghc.Deconstruct(points)
x=np.array(x)
w=np.array(w)
n=len(x)
```

Code Block 3 Pre-Processing of Inputs

3.4.2 Legendre-Gauss-Lobatto Integration Points

Code Block 4 in "Python Lglnodes" is a Python version of Greg von Winckel's MatLab code: Legendre-Gauss-Lobatto nodes and weights [14]. Using recurrence definition of Legendre polynomials, the codes generate 1D Legendre-Gauss-Lobatto node positions and integration weights based on the number of orders, and stored in "x" and "w" respectively.

```
N1=N+1
x=-np.cos(np.pi*np.arange(0,N1).T/N)
P=np.empty((N1,N1))
xold=2
while (np.abs(x-xold)).max()>np.spacing(1):
    xold=x;
    P[:,0]=1
    P[:,1]=x
    for k in range(2,N1):
        P[:,k]=((2*k-1)*x*P[:,(k-1)]-(k-1)*P[:,(k-2)])/k
    x=xold-(x*P[:,N]-P[:,(N-1)])/(N1*P[:,N])
w=2/(N*N1*P[:,N]**2)
x=x.tolist()
w=w.tolist()
P=P.T
```

Code Block 4 Legendre-Gauss-Lobatto Integration Points for 1D Elements

2D integration points and their integration weights are then generated with Code Block 5 in "Python Integration". "xp4" and "yp4" stores the x and y coordinates of the element nodes respectively. "Eta1" and "Zeta1" denotes the local coordinates of the element nodes. "w_i" stores the integration weights of the element nodes.

```
##Generate Integration Points
a=(1+x)/2
xp1=(xp[3]-xp[0])*a+xp[0]
xp2=(xp[2]-xp[1])*a+xp[1]
yp1=(yp[3]-yp[0])*a+yp[0]
yp2=(yp[2]-yp[1])*a+yp[1]

xp3=np.empty((n,n))
yp3=xp3.copy()
w_=xp3.copy()
Eta=xp3.copy()
i=0
while i<n:
    xp3[i]=(xp2[i]-xp1[i])*a+xp1[i]
    yp3[i]=(yp2[i]-yp1[i])*a+yp1[i]
    w_[i]=w[i]*w
    Eta[i]=x
    i+=1
```

```
xp4=map(float,xp3.flatten(0))
yp4=map(float,yp3.flatten(0))
w_i=map(float,w_.flatten(0))
Eta1=map(float,Eta.flatten(0))
Zeta1=map(float,Eta.flatten(1))
```

Code Block 5 Legendre-Gauss-Lobatto Integration Points for 2D Elements

3.4.3 Geometry

For preview of the element shape, Code Block 6 in "Python Integration" generate the edges, nodes, and the rectangle surface of the element in Rhinoceros modeling space. "L" denotes the shortest span of the element.

```
##Show Geometry of the element
Int_p=[]
for (i,j) in zip(xp4,yp4):
    Int_p.append(rg.Point2d(i,j))

Line=[]
Length=[]
i=0
while i<len(points):
    a=ghc.ListItem(points,i)
    b=ghc.ListItem(points,i+1)
    Line.append(rg.Line(a,b))
    Length.append(rs.Distance(a,b))
    i+=1
L=float(np.mean(sorted(Length)[:2]))

Rec=ghc.Rectangle3Pt(points[0],points[1],points[2])
Area,Centorid=ghc.Area(Rec)
```

Code Block 6 Geometry of Elements

3.4.4 Generating Shape Functions, Derivatives and Jacobean Matrices

Code Block 7 in "Python Integration" defines a new object type called OneDShape which takes in two arguments, or inputs, 1D element node positions and position of one 1D element node. The OneDShape object can store the 1D shape functions and the derivatives of the 1D shape functions it generates as attributes of the object. The rest of the codes in Code Block 7 generates 2D shape

functions and their derivatives with respect to either local coordinate directions. It also generates the Jacobean matrices for each node.

```
##Generate Shape Functions, Derivatives, and Jacobean
Matrices
class OneDShape(object):
    def __init__(self, x, x_i):
        self.x=x
        self.x_i=x_i
        self.n=len(self.x)
        self.f=np.empty((1,self.n))
        self.df=self.f.copy()
        i=0
        while i<self.n:
            a=x_i-x
            a[i]=1
            b=x[i]-x
            b[i]=1
            self.f[0,i]=np.prod(a/b)
            ind=np.nonzero(a==0)
            if np.size(ind)==0:
                self.df[0,i]=self.f[0,i]*(np.sum(1/a)-1)
            else:
                a[ind]=1
                self.df[0,i]=np.prod(a/b)
            i+=1

N=np.empty((n**2,n**2))
N_eta=N.copy()
N_zeta=N.copy()
det_J=[]
J=np.empty((n**2,2,2))
inv_J=J.copy()
i=0
while i<(n**2):
    a=OneDShape(x,Eta1[i])
    b=OneDShape(x,Zeta1[i])
    N[i,:]=np.dot(a.f.T,b.f).flatten(1) #Each line for each
node
    N_eta[i,:]=np.dot(a.df.T,b.f).flatten(1)
    N_zeta[i,:]=np.dot(a.f.T,b.df).flatten(1)
    J[i][0][0]=np.dot(xp4,N_eta[i].T)
    J[i][0][1]=np.dot(xp4,N_zeta[i].T)
    J[i][1][0]=np.dot(yp4,N_eta[i].T)
    J[i][1][1]=np.dot(yp4,N_zeta[i].T)
    det_J.append(float(np.linalg.det(J[i])))
    inv_J[i]=np.linalg.det(J[i])
    i+=1
```

Code Block 7 Generating Shape Functions, Derivatives, and Jacobean Matrices

3.4.5 Generating $[E^0]$, $[E^1]$, $[E^2]$

Coefficients $[E^0]$, $[E^1]$ and $[E^2]$ defined in section 2.2.3 are then estimated and assembled with Code Block 8, denoted as "E0", "E1" and "E2" respectively.

```
a1=E/(1+poisson)/(1-2*poisson)
a2=1-poisson
a3=(1-2*poisson)/2
C=np.zeros((6,6))
C[0:3][:,0:3]=np.array([[a2,poisson,poisson],[poisson,a2,poisson],[po
isson,poisson,a2]])
C[3:6][:,3:6]=np.diag([a3,a3,a3])
C=a1*C

b1=np.zeros((6,3))
b2=np.zeros((n**2,6,3))
b3=b2.copy()
B2i1=np.empty((6,3))
B2i2=B2i1.copy()
B2j2=B2i1.copy()

b1[0,0]=1;b1[5,1]=1;b1[4,2]=1;
G=E/2/(1+poisson)
a1=G*np.diag([2*(1-poisson)/(1-2*poisson),1,1])
Cb1=np.dot(C,b1)

E0=np.zeros((3*n**2,3*n**2))
E1=E0.copy()
E2=E0.copy()
a2=np.empty((1,n**2))
ind1=[]
ind2=[]
i=0
while i<(n**2):
    a2[0,i]=w_i[i]*det_J[i]
    ind1=[i,(n**2+i),(2*n**2+i)]
    j=0
    while j<(n**2):
        p=0
        E2ij=np.zeros((3,3))
        while p<(n**2):
            J1=J[p][0,0]
            J2=J[p][0,1]
            J3=J[p][1,0]
            J4=J[p][1,1]
            b2[p][1,1]=J4;b2[p][3,2]=J4;b2[p][5,0]=J4
            b2[p][2,2]=-J2;b2[p][3,1]=-J2;b2[p][4,0]=-J2
            b3[p][1,1]=-J3;b3[p][3,2]=-J3;b3[p][5,0]=-J3
            b3[p][2,2]=J1;b3[p][3,1]=J1;b3[p][4,0]=J1
            B2i2=(1/det_J[p])*(b2[i]*N_eta[p,i]+b3[i]*N_zeta[p,i])
            B2j2=(1/det_J[p])*(b2[j]*N_eta[p,j]+b3[j]*N_zeta[p,j])
            E2ij=E2ij+(w_i[p]*np.dot(B2i2.T,np.dot(C,B2j2))*det_J[p])
```

```
            p+=1
        B2i1=b2[i]*N_eta[j,i]+b3[i]*N_zeta[j,i]
        E1ij=w_i[j]*np.dot(B2i1.T,Cb1)
        ind2=[j,(n**2+j),(2*n**2+j)]
        E1[np.ix_(ind1,ind2)]=E1ij
        E2[np.ix_(ind1,ind2)]=E2ij
        j+=1
    i+=1
for i in range(0,3):

E0[(i*n**2):(i+1)*n**2][:,(i*n**2):(i+1)*n**2]=np.diag((a2*a1[i,i]).f
latten())
```

Code Block 8 Generating $[E^0]$, $[E^1]$, $[E^2]$

3.4.6 Assembly of Stiffness Matrix

Code Block 9 in "Python Integration" assembles the stiffness matrix defined in section 2.2.4. "s_" denotes the submatrix of the stiffness matrix that works with kinematics of Kirchhoff plate theory.

```
#Assembly of Z, V and s
Z_11=np.linalg.solve(E0,E1.T)
Z_12=-np.linalg.inv(E0)
Z_21=-E2+np.dot(E1,np.linalg.solve(E0,E1.T))

V_11=1./12*(np.dot(Z_11,Z_11)+np.dot(Z_12,Z_21))
V_21=1./12*(np.dot(Z_21,Z_11)-np.dot(Z_11.T,Z_21))


temp=np.eye(3*n**2)+(t**2)*V_11
a=np.dot(E0,temp)
b=E1.T
c=np.dot(E1,temp)-t**2*V_21
d=E2

s=t*np.vstack([np.hstack([a,b]),np.hstack([c,d])])
ind_b=np.arange(n**2,4*n**2)
ind_a=np.hstack([range(0,n**2),range(4*n**2,6*n**2)])
s_=s[np.ix_(ind_b,ind_b)]
```

Code Block 9 Stiffness Matrix Assembly

3.4.7 Solver

Vectors of displacements, rotations about in-plane axes, transverse deflections, moments about in-plane-axes, and transverse shear are initialed in Code Block 10 in "Python Integration". The vectors are prefilled with zeros to save effort in inputting fixed displacements or zero loads.

```
##Initial Vectors
uz=np.zeros((n**2,1))
ux=uz.copy()
uy=uz.copy()
theta_y=uz.copy()
theta_x=uz.copy()
w=uz.copy()
My=uz.copy()
Mx=uz.copy()
Fz=uz.copy()
```

Code Block 10 Initial Vectors

Then a user has to input the boundary conditions and loading information into the "Python Integration" either through direct insertion of Python scripts or with the help of Grasshopper components. This part needs to provide the indices of the free degree of freedom of "theta_y", "theta_x" and "w" as "ind_f1" "ind_f2" and "ind_f3" respectively, and those of the fixed degree of freedom as "ind_d1", "ind_d2" and "ind_d3". Displacements and Loads on free degree of freedom nodes needs to be filled in the vectors in Code Block 10. Note that in order to solve the matrix, "ind_f1", "ind_f2" and "ind_f3" cannot be all empty, which implies there may be a lower limit on the order of nodes.

Code Block 11 in "Python Integration" can then solve the SBFEM equations based on the information mentioned above. The rotations about in-plane axes, transverse deflections, moments about in-plane axes and transverse shear of each node can be output in lists "theta_y", "theta_x", "w", "My", "Mx" and "Fz".

```
ind_f=np.hstack((ind_f1,np.hstack((ind_f2+n**2,ind_f3+2*n**2))))
ind_d=np.hstack((ind_d1,np.hstack((ind_d2+n**2,ind_d3+2*n**2))))
s_ff=s_[np.ix_(ind_f,ind_f)]
F_f=np.vstack((My[np.ix_(ind_f1,[0])],np.vstack((Mx[np.ix_(ind_f2,[0]
)],Fz[np.ix_(ind_f3,[0])]))))
u_f=np.linalg.solve(s_ff,F_f)

s_df=s_[np.ix_(ind_d,ind_f)]
F_d=np.dot(s_df,u_f)
```

```
theta_y[np.ix_(ind_f1,[0])]=u_f[0: np.size(ind_f1)]
theta_x[np.ix_(ind_f2,[0])]=u_f[np.size(ind_f1): np.size(ind_f2)]
w[np.ix_(ind_f3,[0])]=u_f[np.size(ind_f2): np.size(ind_f3)]

My[np.ix_(ind_d1,[0])]=F_d[0: np.size(ind_d1)]
Mx[np.ix_(ind_d2,[0])]=F_d[np.size(ind_d1): np.size(ind_d2)]
Fz[np.ix_(ind_d3,[0])]=F_d[np.size(ind_d2): np.size(ind_d3)]
```

Code Block 11 Solving Matrices

CHAPTER 4: DISCUSSION

With the example of Python implementation shown in Chapter 3, users of Rhinoceros can study the plate bending behavior in any style. Typically, when the whole structure is not fully designed, designers can apply the Python implementation to perform customized check on the local plate components in the design. Using Grasshopper components for various display techniques, one can express the results in colors or some graphical patterns, which is visually straightforward. Furthermore, when combined with a built-in Galapagos evolutionary solver in Grasshopper, dimensions of plates can be optimized under designed boundary conditions and loadings. The Python codes can be adapted to other modeling applications, which compile Python scripts as well.

Since it is preliminary, various aspects of the Python implementation need to be improved to be practical. The codes can be extended to discretize plates with multiple elements and assemble the stiffness matrices element-by-element. Plates with curved boundaries, uneven thickness or complicated boundary conditions shall be identified with developed codes in the future. In addition, Hou Man et al.'s method can be extended into the field of dynamics or shells to expand the scope of the application of the methods.

CHAPTER 5: CONCLUSION

Although the Python implementation presented in this thesis can only perform static structural analysis on a single rectangular plate with uniform thickness, simple boundary conditions and simple loading, the Python codes can be extended easily to fully implement Hou Man et al.'s technique for plate bending analysis. The technique permits little restriction on plate dimensions, faster convergence and great reduction in computational expenses, which implies the future development of the Python implementation will enable more practical, faster and generalized procedures for analysis of plate structures. The implementation presented in this thesis represents the first step in development of interactive programs for structural design and analysis of plates. With highly customizable algorithm tools such as Python and Grasshopper and abundant sources of their add-ons, designers will be able to integrate the design and analysis processes of plate structures seamlessly with rapid feedbacks.

REFERENCE

[1] Ventsel, Eduard, and Theodor Krauthammer. *Thin Plates and Shells: Theory, Analysis, and Applications*. New York: Marcel Dekker, 2001. Print.

[2] Man, H., C. Song, W. Gao, and F. Tin-Loi. "A Unified 3D-based Technique for Plate Bending Analysis Using Scaled Boundary Finite Element Method."*International Journal for Numerical Methods in Engineering* 91.5 (2012): 491-515. Print.

[3] Wolf, John P. *The Scaled Boundary Finite Element Method*. Chichester, West Sussex, England: J. Wiley, 2003. Print.

[4] Man, H., C. Song, T. Xiang, W. Gao, and F. Tin-Loi. "High-order Plate Bending Analysis Based on the Scaled Boundary Finite Element Method."*International Journal for Numerical Methods in Engineering* 95.4 (2013): 331-60. Print.

[5] Bathe, Klaus-Jürgen. *Finite Element Procedures*. S.l.: S.n., 2006. Print.

[6] Vosse, F.N Van De., and P.D Minev. *Spectral Element Methods: Theory and Applications/ F.N. Van De Vosse and P.D. Minev*. Eindhoven: Eindhoven U of Technology, Faculty of Mechanical Engineering, 1996. Print.

[7] "17 Isoparametric Quadrilaterals." Web. 29 Apr. 2014. <http://www.colorado.edu/engineering/cas/courses.d/IFEM.d/IFEM.Ch17.d/IFEM.Ch17.pdf >.

[8] Sanner, Michel F. "Python: a programming language for software integration and development." J Mol Graph Model 17.1 (1999): 57-61.

[9] "Grasshopper." - Algorithmic Modeling for Rhino. Web. 29 Apr. 2014. <http://www.grasshopper3d.com/>

[10] "GhPython." Food4Rhino. Web. 29 Apr. 2014. <http://www.food4rhino.com/project/ghpython>.

[11] "NumPy." NumPy — Numpy. Web. 30 Apr. 2014. <http://www.numpy.org/>.

[12] "Numpy and Scipy in RhinoPython." Steve Baers Notes. Web. 30 Apr. 2014. <http://stevebaer.wordpress.com/2011/06/27/numpy-and-scipy-in-rhinopython/comment-page-1/>.

[13] "Utility.Py." GitHub. Web. 30 Apr. 2014. <https://github.com/mcneel/rhinopython/blob/master/scripts/rhinoscript/utility.py>.

[14] Von Winckel, Greg. "Legende-Gauss-Lobatto Nodes and Weights." MATLAB Central. 20 Apr. 2004. Web. <http://www.mathworks.com/matlabcentral/fileexchange/4775-legende-gauss-lobatto-nodes-and-weights/content/lglnodes.m>.

[15] "16 The Isoparametric Representation." Web. 30 Apr. 2014. <http://www.colorado.edu/engineering/cas/courses.d/IFEM.d/IFEM.Ch16.d/IFEM.Ch16.pdf >.

[15] Birman, V. *Plate Structures*. N.p.: Springer, 2011. Print.

[17] Coldwell, Robert. "Lagrange Interpolation." N.p., n.d. Web. 30 Apr. 2014. <http%3A%2F%2Fwww.phys.ufl.edu%2F~coldwell%2Fwsteve%2FFDERS%2FThe%2520Lagrange%2520Polynomial.htm>.

[18] Felippa, Carlos A. "A Compendium of FEM Integration Formulas for Symbolic Work." *Engineering Computations* 21.8 (2004): 867-90. Print.

[19] Ferreira, A. J. M. *MATLAB Codes for Finite Element Analysis: Solids and Structures*. Dordrecht: Springer Science & Business Media, 2009. Print.

[20] Kaveh, A. *Computational Structural Analysis and Finite Element Methods*. N.p.: n.p., n.d. Print.

[21] Lepi, Steven M. *Practical Guide to Finite Elements: A Solid Mechanics Approach*. New York: Marcel Dekker, 1998. Print.

[22] "Mcneel/rhinopython/utility.py." *GitHub*. N.p., n.d. Web. 30 Apr. 2014. <https://github.com/mcneel/rhinopython/blob/master/scripts/rhinoscript/utility.py>.

[23] Natarajan, Sundararajan, Chongmin Song, Ean T. Ooi, and Irene Chiong. "Displacement Based Finite Element Formulations over Polygons: A Comparison between Laplace Interpolants, Strain Smoothing and Scaled Boundary Polygon Formulation." N.p., n.d. Web. 30 Apr. 2014. <http://arxiv.org/abs/1309.1329>.

[24] Song, Chongmin. "The Scaled Boundary Finite Element Method in Structural Dynamics." *International Journal for Numerical Methods in Engineering*77.8 (2009): 1139-171. Print.