Master's thesis

NTNU
Norwegian University of Science and Technology
Department of Marine Technology

Embla Larsdotter Holten

Isogeometric contact analysis: Implementation of a penalty-based algorithm

Master's thesis in Marine Technology
Supervisor: Josef Kiendl
June 2019

NTNU
Norwegian University of
Science and Technology

Preface

Being able to analyse the behaviour of objects in contact has always been a subject of interest to structural engineers. Contact problems often involve large deformations and non-linearities and their complexity can be a challenge to classical FEM. Recently an alternative to FEM has been developed called isogeometric analysis, IGA. IGA has features that are advantageous in many fields of engineering and not least within structural contact problems.

The marine Department of NTNU is currently developing an IGA research code for which this thesis aims to add a contact implementation to. The scope of the thesis was provided by Assoc Prof Josef Kiendl, who has accordingly been my supervisor throughout the thesis work. This thesis gives an introduction to contact mechanics in general and provides the equations for the implemented contact solution procedure. A brief review of IGA and its basis functions are also provided. Most importantly, the implementation of a contact algorithm is described and its results are discussed.

It is assumed that the reader have some background knowledge within structural engineering and especially the Finite Element Method solution procedure.

Trondheim June 11, 2019

Embla Holten

Acknowledgement

I would like to thank my supervisor Assoc. Prof. Josef Kiendl for giving me the opportunity of investigating such an interesting and promising field of structural engineering and for the opportunity of staying at another university for further inspiration. His guidance and sharing of knowledge has been much helpful throughout the semester and kept me motivated. I would also like to acknowledge Prof. Dr. Laura De Lorenzis and Dr.-Ing. Marreddy Ambati for the guidance and inspiration they provided at TU Braunschweig.

A thank you is also directed to PhD candidate Davide Proserpio for the never ending patience with decoding and explaining his implementations. Your help has been greatly valued.

Lastly I am grateful for all the encouragement and optimism from the people around me that have made the final year at NTNU one of the best. Thank you to my office-mates, to my dear friends, my family and to my dear Sondre. Thank you for supporting me every day. And lets not forget Stephanie and Achim who welcomed me into their lives in Braunschweig with open arms.

Abstract

Structural analysis of contacting bodies is complex. They typically involve large deformations and as the contact interface is unknown in advance, they are unavoidably nonlinear. Finite element method has been used to solve contact problems almost since its beginning in the 1960's. Due to its discretisation scheme, it exhibits a low inter-element continuity of typically C^0 or C^1 and an approximated geometric model. This can especially be a challenge in contact analysis as contact problems are sensitive to the surface description. The solution procedures are consequently subject to a lack of robustness and accuracy. Isogeometric analysis is a recently developed alternative to FEM. It has the potential to improve some of the major challenges of the previous solution schemes with FEM. A variety of methods have thus been formulated for solving contact problems with IGA since its origin around 2006 by Hughes and coworkers, (Hughes et al., 2005).

The Marine Department of NTNU are developing an IGA research code for structural analysis in MATLAB. It mainly uses non-rational B-splines as basis functions, which are the typical basis functions of IGA, to describe the geometry and the solution field. It is essentially formulated with Kirchhoff-Love shell elements. This thesis aims to contribute to the research code by adding a first implementation of contact analysis. A contact algorithm is proposed, using the penalty method in combination with Gauss-point-to-segment, GPTS, contact discretization. The algorithm is coded to handle contact between multiple bodies, contact by external force and contact by moving rigid bodies. A two step point search algorithm is proposed that has the potential to increase robustness and speed of the analysis compared to a one step search.

Sammendrag

Strukturanalyse av gjensander som kommer i kontakt med hverandre er komplisert. Kontaktproblemer er ofte relatert til store deformasjoner og ettersom kontaktoverflaten ikke er kjent på forhånd, er de uunngåelig ikke-lineære. FEM har blitt brukt til å løse kontaktproblemer nesten helt siden det ble oppfunnet på 60-tallet. Finite element-diskretiseringen av overflater fører til lav kontinuitet mellom elementer, typisk C^0 og C^1, og en tilnærmet geometribeskrivelse. Dette kan være ekstra utfordrende når det kommer til å løse kontaktproblemer ettersom de er sensitive for hvordan kontaktoverflaten til objektene er beskrevet. Løsningsmetodene innen kontaktanalyse med FEM er derfor preget av unøyaktighetere og å være lite robuste. Isogeometrisk analyse, IGA, er et nylig utviklet alternativ til FEM. IGA har vist evne til å minske de typiske problemente som oppstår ved bruk av FEM på kontaktproblemer. Det har blitt viet en stor forskningsinnsats innen temaet og mange metoder for å løse kontaktproblemer med IGA har blitt formulert siden opprinnelsen rundt 2006 av Hughes og kolleger, (Hughes et al., 2005).

Institutt for Marin Teknikk, NTNU, holder for øyeblikket på med å utvikle en IGA forskningskode i MATLAB. Den benytter hovedsakelig *non-rational B-splines*, NURBS, som basisfunksjoner for å beskrive geometri og deformasjonsfelt. I hovedsak er den utviklet med bruk av Kirchhoff-Love skall-elementer. Denne masteroppgaven har som mål å bidra til forskningskoden ved å implementere en kontaktalgoritme til den eksisterende IGA-formuleringen. *Penalty*-metoden blir brukt for å legge til grensebetingelsene fra kontakt og *Gauss-point-to-segment*, GPTS, diskretiseringsteknikk. Algoritmen blir utviklet slik at den kan håndtere kontakt mellom flere objekter samtidig, kontaktanalyse med ytre krefter og kontaktanalyse ved å flytte på stive legemer. En to-stegs kontaktsøkealgoritme er implementert som viser potensiale for å øke både robustheten til implementeringen og hastigheten.

Contents

List of Figures

List of Tables

Abbreviations and symbols

FEM - Finite element method

FEA - Finite element analysis

IGA - Isogeometric analysis

CAD - Computer aided design

CAE - Computer aided engineering

NURBS - Non-rational B-splines

NTS - Node-to-segment

GPTS - Gauss-point-to-segment

–NURBS and IGA–

ξ, η - Parametric coordinates

Ξ - Knot vector in parametric direction ξ

H - Knot vector in parametric direction η

p, q - Polynomial degree in parametric direction ξ, η

N - B-spline basis function

$C(\xi)$ - B-spline or NURBS curves

$\boldsymbol{P}$ - Control point coordinates

w_i - NURBS basis functions weights

R - NURBS basis function

–Penalty and GPTS formulations–

Superscript *m, s* - Term related to *s=slave, m=master*

Ω - Body

$\boldsymbol{x}$ - Point in current configuration

$\boldsymbol{X}$ - Point in reference configuration

$\boldsymbol{u}$ - Displacement of point

γ_C - Contact interface in the current configuration

$\bar{\boldsymbol{x}}^m$ - Normal projection point (denoted simply $\boldsymbol{x}^m$, from *Section 4.2*)

$\bar{\boldsymbol{n}}$ - Surface normal directed towards the slave body, at normal projection point (denoted simply $\boldsymbol{n}$ from *Section 4.2*). Also denoted *contact normal*.

Γ_C - Contact interface in reference configuration

g_N - Normal gap function

t_N - Normal traction due to contact

$\boldsymbol{t}$ - The Piola traction vector

W - Potential energy of a system

ϵ_N - Penalty parameter

W_c^P - Contact penalty contribution to the weak form

$\Delta\delta W_c^P$ - Linearisation of the contact contribution to the weak form

$\boldsymbol{N}_N$ - Slave and master NURBS basis functions in the normal direction.

$\boldsymbol{k}_{geo}$ - Geometric stiffness due to contact

$\boldsymbol{m}^{-1}$ - Inverse metric tensor calculated at master surface

$\boldsymbol{k}$ - Curvature tensor calculated at master surface

$\boldsymbol{T}$ - Slave and master NURBS basis functions multiplied with covariant vectors of the master surface $\boldsymbol{\tau}$

K_{gp} - Contact stiffness contribution at a Gauss point, GPTS formulation

$\boldsymbol{R}_{gp}$ - Contact force residual contribution at a Gauss point, GPTS formulation

$\boldsymbol{F_r}$ - Total force residual due to contact

$\boldsymbol{K}_C$ - Total stiffness contribution due to contact

z_{gp} - Gauss-Legendre weight for a specific Gauss point.

–Implementation–

L_{ref} - The absolute distance between centre point a slave and master element.

L_{min} - The smallest out of all L_{ref} related to a specific slave element.

L_{acc} - The absolute acceptable distance between a slave element and master surface for them to be considered for contact.

Chapter 1

Introduction

Contact mechanics is a natural part of engineering problems and even affects our daily lives in numerous ways. Contact occur between joints in our bodies, between the car tire and the road and between the ground and bridge foundations. The science of contact mechanics accordingly play a significant role in deciding the design of various structures and has since beginning of history. The knowledge of friction and lubricants made the Egyptians able to transport huge blocks of stones to build the pyramids just with human force, (Carnes, 2005).

The origin of the modern contact mechanics can be traced back to the 1800's. Poisson studied the the ability of a body to restore itself to its original shape after undergoing deformation, its elasticity. Hertz applied this knowledge to contacting bodies and was able to develop an analytic solution to the problem. The Hertz problem consists of two spheres contacting in the normal direction, see Figure (1.1). The spheres are exposed to an external force F. The analytic solution is expressed by the elastic modulus, E, the Poisson ratio ν and the radii of the surfaces, r_1 and r_2. The result is the width of the final contact interface b and the contact pressure p, expressed in (Zavarise and De Lorenzis, 2009*b*) as in Eq. (1.1) and (1.2).

$$b = \sqrt{\frac{4F}{\pi l}\frac{(1-\nu_1^2)/E_1 + (1-\nu_2^2)/E_2}{(1/r_1)+(1/r_2)}} \tag{1.1}$$

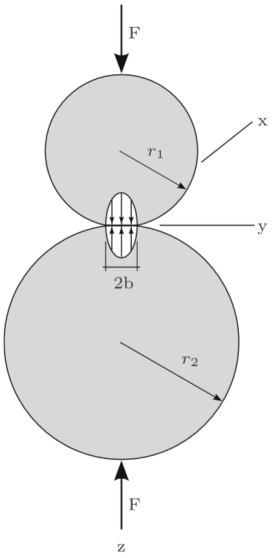


Figure 1.1: The Hertz contact problem, (Frankie et al., 2010)

$$p(y) = \frac{E}{2r(1-\nu^2)} \sqrt{4F\frac{(1-\nu^2)r}{\pi l E} - y^2} \tag{1.2}$$

With advancement of computational technology came the breakthrough of numerical methods used in structural analysis. From its beginning in the 1950's and 60's, (Frankie et al., 2010), CAE - Computer-Aided Engineering with the finite element method has been the most widespread method to solve complicated structural problems. Contact problems were incorporated into FEM soon after its invention, for which one of the earlier works can be seen in (Parsons and Wilson, 1970). Finite element methods solves the structure in a weak sense, using typically simple linear or quadratic polynomials to interpolate the surface over a parametric domain, the elements. The simplified surface representation can be a challenge for contact problems. Imagine the edge of a body sliding across another surface. Due to the FEM representation, the edge will slide through element domains and for each new element experience a small abrupt change in force and direction.

Another tool playing a significant role in a structural design processes is CAD, Computer-Aided Design. It represents surfaces using smooth functions which

are able to accurately represent geometry. The standard smooth functions used in CAD tools today are non-rational B-splines, NURBS. They result in a greatly flexible modelling scheme for which anything from straight lines to spheres and kinks can be accurately described. The geometrical model of a structure is thus first made in a CAD-program in order to have an accurate geometric representation. The model then have to be meshed into finite elements before it can be applied a structural analysis using FEM. In FEM, the geometry as well as solution field is represented by simple interpolation polynomials. Furthermore it is necessary to go back to the original model for every new mesh generation due to the FE discretisation. Meshing is for complex geometries a time consuming procedure and often requires to be manually adjusted, (Breitenberger, 2016). In order to reduce the effort needed for the meshing process, a new procedure was proposed in the 2000's by Wriggers and coworkers, (De Lorenzis, Hughes and Wriggers, 2014) in (Hughes et al., 2005) called isogeometric analysis, IGA. It aimed to create a bridge between CAD and CAE by using the discretisation and solution procedure in FEM and the smooth basis functions from CAD. In addition to reduce the time consuming procedure of meshing a model, it showed potential to further improve the analysis by increased accuracy and robustness on a per-degree of freedom basis, see(Grossmann et al., 2012), as described in (De Lorenzis, Hughes and Wriggers, 2014).

Contact problems are complex. They are unavoidably nonlinear in nature as the contact interface is not known in advance and are often subjected to large deformation and sliding. IGA has the potential to increase accuracy, produce more physically acting contact pressures and improve convergence of contact problems, which is shown in (Fischer and Wriggers, 2005). Due to its promise to improve some of the challenges related to contact analysis, it has been a significant scientific effort in the field the last decade for which a variety of methods have been developed. Most of them are directly translated from original FEM techniques and accordingly inherit the original limitations. Even though the robustness and accuracy are improved, there are still issues related to the transferring of accurate contact pressures between bodies, convergence rates and efficiency and there is no single method that is recognised as the superior contact solution scheme.

Objective and scope

A structural IGA research code at the department of Marine Technology, NTNU is under development. It is mainly formulated with Kirchhoff-Love shell elements and NURBS basis functions. There exist per now no commercial IGA code including Kirchhoff-Love shell elements with implemented contact analysis. The objective of this thesis is to add a first contact implementation into the IGA research code. A simplified contact implementation is provided as starting point for the work. The implementation is to be based on established contact formulations using penalty method to impose contact constraints and a chosen contact discretisation. The implementation should include the calculation of a geometric stiffness matrix for large deformation contact between flexible bodies, contact between rigid and flexible bodies and a point search algorithm.

The proposed algorithm is coded to handle multiple bodies in contact and contact is imposed by external force and by moving rigid bodies. The code is tested for numerical examples of different geometry and complexity. Numerical studies are conducted in order to investigate effects of different modelling choices such as the penalty parameter and choice of master and slave body.

Structure of thesis

This thesis first presents the fundamental concept of NURBS as a basis for surface modelling, (*Ch. 2*) and IGA as a method (*Ch. 3*) for which the NURBS based Kirchhoff-Love shell element is briefly reviewed. Further the computational contact formulations used in the contact implementation are presented and discussed (*Ch. 4*). The complete contact algorithm proposed is described in (*Ch. 5*). In *Ch. 6* three numerical examples that are implemented are presented and discussed. The results are concluded in (*Ch. 7*)

Chapter 2

Modelling geometry with NURBS

Mathematical expressions that are able to model free-form shapes were first developed by Bèzier in the 1960's, (Rogers, 2001). Later came the B-spline functions and Non Uniform Rational B-splines, NURBS. NURBS have the advantageous ability to accurately model a variety of curves and surfaces extending from straight lines to curves and spheres and also describe free-form shapes. They are thus the standard functions used in CAD tools today.

It is first later, in the 2000's, that they are transferred to the finite element environment, resulting in IGA. The IGA research code of the Marine Department is based on NURBS functions and this chapter further aims to give the reader a basic understanding of NURBS and how they are used to model geometry. As NURBS are derived from complicated mathematical formulations, only the end product is described in this chapter.

2.1 B-splines

In order to introduce the reader to non Uniform Rational B-splines, it is natural to start with the simpler B-splines.

In FEM, interpolation polynomials are used to approximate a set of points called nodes. The interpolation polynomials represents the given points accurately,

while oscillations might occur between them, as described in (Kiendl, 2011). As a result, the geometry in FEM is only represented exactly at the nodes, while it is approximated between them. B-splines are other functions used to approximate a set of given points, called control points. They stand in contrast to the polynomials used in FEM. They are not interpolation functions and only approximates the control points. Furthermore, they do not result in oscillations between the points and form a smooth curve from the first control point to the last one defined. Only the first and last control points are described accurately as it is beneficial from a design perspective to be able to define the exact starting and ending point of a curve. B-spline curves are defined by a linear combination of control points and basis functions called B-splines, over a parametric space.

Knot vector

B-spline curves are defined over a parametric space which is divided into intervals. The intervals are defined by the knot vector and B-splines are defined piecewise on these intervals. The knot vector consist of a set of parametric coordinates ξ_i between 0 and 1, $\Xi = [\xi_1, \xi_2, ..., \xi_{n+p+1}]$. p is the polynomial degree of the basis function and n is the number of basis functions. The knot entries are either increasing in order or repeated. A *knot span* is the parametric space between two distinct knots in the knot vector. B-splines are C^∞ continuous inside a knot span and C^{p-k} continuous on knots repeated k times. In this way it is possible to model a curved surface with a kink, by repeating one of the knot entries $k = p$ times. If the first and the last knot in the knot vector is repeated $p+1$ times, it is called an *open* knot vector, (Kiendl, 2011). The first and last control points of the open knot vector are interpolated. Consequently the start and end point of the curve can easily be chosen, as the B-spline curve exactly represents the points instead of approximating them.

The B-splines mathematical expression

The B-spline basis functions, N are formulated in (Kiendl, 2011). They are computed by the *Cox-de Boor* recursion formula and are defined by the knot vector and polynomial degree p. It starts for $p = 0$,

$$B_{i,0}(\xi) = \begin{cases} 1 & \xi_i \leq \xi \leq \xi_{i+1} \\ 0 & otherwise. \end{cases} \tag{2.1}$$

For $p \geq 1$ the basis functions are,

$$B_{i,p}(\xi) = \frac{\xi - xi_i}{\xi_{i+p} - \xi_i} B_{i,p-1}(\xi) + \frac{\xi_{i+p+1} - xi}{\xi_{i+p+1} - \xi_{i+1}} B_{i+1,p-1}(\xi) \tag{2.2}$$

B-spline curves, $C(\xi)$ are as mentioned defined as a linear combination of control points, $\boldsymbol{P}$, and basis functions,

$$C(\xi) = \sum_{i=1}^{n} N_{i,p}(\xi) \boldsymbol{P}_i. \tag{2.3}$$

2.2 NURBS

NURBS are piece-wise rational polynomials while B-splines are piece-wise polynomials. Each control point has in addition to its coordinates a weight, w_i. The weight act as a "pulling force" between the related control point and NURBS curve. The larger the weight, the more the curve is pulled towards the control point. This result in an increased local and global control of the geometric representation. The NURBS functions are formulated in (Kiendl, 2011). The basis functions are a combination of B-spline basis functions, N and weights,

$$R_{i,p}(\xi) = \frac{N_{i,p}(\xi) w_i}{\sum_{i=1}^{n} N_{i,p}(\xi) w_i} \tag{2.4}$$

and NURBS curves are formulated in the same manner as the B-spline curves,

$$\boldsymbol{C}(\xi) = \sum_{i=1}^{n} R_{i,p}(\xi)(P)_i. \tag{2.5}$$

A NURBS surface definition includes basis functions in two parametric directions, ξ, η and a grid of control points of size $m \times n$. There are two related knot vectors and polynomial degrees p, q. One for each parametric direction.

$$\boldsymbol{S}(\xi, \eta) = \sum_{i=1}^{n} \sum_{j=1}^{m} R_{i,j}^{p,q}(\xi, \eta) \boldsymbol{P}_{i,j} \tag{2.6}$$

with basis functions

$$R_{i,j}^{p,q}(\xi,\eta) = \frac{N_{i,p}(\xi) M_{j,q}(\eta) w_{i,j}}{\sum_{i=1}^{n} \sum_{j=1}^{m} N_{i,p}(\xi) M_{j,q}(\eta) w_{i,j}} \tag{2.7}$$

Chapter 3

IGA

During a structural design process, two models of the geometry are commonly made. One in a Computer Aided Design, CAD, tool in order to accurately model the geometry, and then in a finite element tool where the meshing and analysis is executed. The two tools uses different mathematical expressions to describe the geometries and a CAD model can thus not be directly transferred to a finite element program. FEM typically uses simple Lagrange polynomials as basis functions and CAD uses a set of smooth functions, NURBS. The finite element geometric representation is only an approximation of the CAD geometry.

The geometric modelling in FEA is related to multiple challenges. Geometric information is lost in the meshing process since the geometry is approximated by interpolating using simple polynomials and nodes. In order to increase the accuracy of the geometric model, the mesh density is increased. However the refinement can not be done without increasing the number of degrees of freedom and such the unknown variables that needs to be solved for. Moreover, the whole process of meshing a structure needs to be redone if the mesh needs to be changed. Intuitively this is a time consuming business. Isogeometric analysis, IGA, was proposed in (Hughes et al., 2005) to serve as a bridge between FEA and CAD. It sought to reduce the time extensive procedure of modelling and meshing in a design process by using the geometric representation of the structural model from CAD throughout the process of design and analysis. With the exact surface representation of NURBS it is possible to reduce the mesh size and fur-

ther the computational effort. IGA thus has the potential to greatly improve the design process and has been a topic of great interest among researcher since its beginning.

IGA is recognised to feature several advantages compared to FEM. It has shown to improve both accuracy and robustness of algorithms, (De Lorenzis, Hughes and Wriggers, 2014). The smooth basis functions result in more precise representation of geometry and higher inter-element continuity. The representation is flexible and the continuity can be adapted at specific locations to create kinks in surfaces. The accurate geometrical description is an obvious advantage when analysing contacting bodies. The accuracy of the solution is directly dependant on the accuracy of the surface representation. Since two approximated surfaces are involved, the error becomes more significant. Contact problems often include large deformation and sliding. The low inter-element continuity of FEA can be a challenge to the convergence of the analysis when elements slide past each other and experience abrupt changes in values and force directions. A higher inter-element continuity is thus advantageous. IGA also exhibits an increased per-degree of freedom accuracy and robustness,shown in (Grossmann et al., 2012). Consequently it is possible to use a coarser mesh to capture the same level of accuracy as FEM when using IGA. The complexity of contact problems often result in a large computational effort needed for the analysis, and it is thus desirable to reduce the number of degrees of freedom in the system.

In this chapter NURBS based IGA and the NURBS based Kirchoff-Love shell element is briefly described. It is presented to get a basic understanding of the IGA research code of the Marine Department of NTNU, which the contact algorithm proposed in this thesis is implemented into.

3.1 NURBS based IGA

NURBS have become the most prevalent functions for CAD due to their ability to accurately model complicated surfaces. They are therefore a natural choice of functions for an IGA code. In NURBS based IGA, the NURBS basis functions are used both to describe the geometry and the unknown solution field, just as the same mathematical formulation is used in FEM to describe the solution

field and the geometry. The isogeoemtric consept can thus be considered an enhancement to the isoparametric concept of FEM, (Kiendl, 2011). IGA is setup in the same way as FEM. It contains the same analysis steps and relateable concepts. In this section the NURBS based elements and meshing procedure is described.

Elements

The structural system is in IGA such as in FEM discretized by subdivision into elements. The stiffness matrix is both for FEM and IGA evaluated at a local element-level and assembled into a global system. The elements are though not as intuitively separated as in FEM as the position of the control points and basis functions are not directly related to the position of the element. The control points are situated outside of the geometric surface and the basis functions can span over multiple elements.

In (Kiendl, 2011), Isogeometric NURBS based elements are defined by knot-spans. A knot-span is a non-zero difference between two subsequent entries in the knot vector. A structure can be defined by multiple knot vectors. Each subdomain that a knot vector is defined on is called a NURBS patch. The full domain can thus consist of multiple patches that again are divided into elements. The element boundaries are illustrated in Figure 3.1.

A set of control points are defined for a patch. Each control point is related to a unique knot. The degrees of freedom and boundary conditions are as in FEM defined at the control points. Each control point has in three dimensions three unique degrees of freedom related to it, in respectively x-, y- and z-direction.

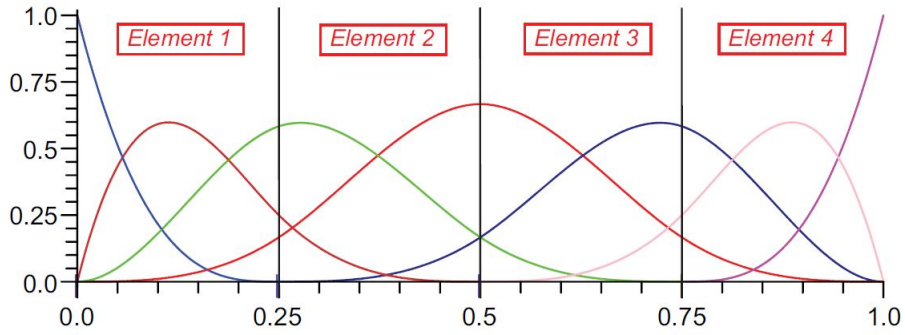


Figure 3.1: NURBS based IGA elements,(Kiendl, 2011)

Mesh

There are two ways to execute a mesh refinement of the geometry in NURBS based IGA. The first is called knot-insertion and the second is called order elevation. Both procedures add control points to the geometric representation. Knot insertion is executed by dividing the knot-spans into smaller intervals by inserting new knots. For each new knot inserted, a control point is added. In order elevation, the polynomial degree is increased. Knot insertion and order elevation can be executed in both parametric directions separately for surfaces.

Knot insertion and order elevation affects the continuity at a knot. Say for example that the polynomial degree of a patch is originally $p = 2$ and the continuity at a knot is $C^{p-k} = C^{2-1} = C^1$. Then order elevation is executed so that $p = 3$. The continuity at the same knot is now $C^{3-1} = C^2$. The original continuity can be restored by adding a repeated knot, $C^{p-k-1} = C^{3-1-1} = C^1$.

The main difference between FEM and NURBS based IGA mesh refinement is as stated in (Kiendl, 2011), that the geometry is represented exactly for all mesh refinements. In FEM, the geometric representation is dependent on the mesh refinement, and it is thus necessary to go back to the original geometric model when changing the mesh density. A drawback of the refinement with NURBS is described in (De Lorenzis, Scott, Wriggers, Taylor and Zavarize, 2014). Since the knots are globally defined, a local mesh refinement can not be done straightforwardly. Contact problems have often a very local need for mesh refinement as the contacting surfaces can be only a small part of the NURBS patch.

3.2 The NURBS based Kirchoff-Love shell element

The shell formulation implemented in the IGA research code of the Marine Department is only briefly reviewed in order to have a basis to interpret the result of the numerical examples. The element is described in detail in (Kiendl, 2011).

Kirchoff-Love shell formulation express curvature as the second derivative of geometric terms of the surface. In order to express the curvature correctly it is thus necessary to have a C^1 inter-element continuity. This degree of continuity is in general not possible for the most commonly used Lagrange polynomials in

FEM. Thus the Kirchhoff-Love shell is not the natural choice of shell elements for FEA. NURBS basis functions on the other hand offer C^1 and higher inter-element continuity. It is thus more straightforward to implement the Kirchoff-Love shell element in IGA than FEM. In (Kiendl, 2011) the kinematics of the Kirchoff-Love thin shell element are used together with NURBS basis functions to create a NURBS based Kirchoff-Love shell element. This is the formulation that is adopted in the IGA research code of the Marine Department of NTNU and thus are the elements used for the numerical contact examples in this thesis.

Assumptions

Kirchhoff-Love shells are only applicable to thin structures. This is due to the assumptions that its formulation is based on. Cross sections normal to the middle surface are assumed to remain normal after deformation and all cross sections stay straight throughout the deformation. The last assumption is equivalent to having a linear strain distribution through the shell thickness. The first assumption corresponds to neglecting transverse shear strains. In thick shells, transverse shear strains can not be neglected. In order for a shell to be considered thin, the ration $R/t > 20$, (Kiendl, 2011), needs to be fulfilled, where R is the radius of curvature and t is the shell thickness. This is the case for most of shells in practical applications. Reisser-Mindlin shells, a formulation made for thick shells, are though more used in FEM. That is due to the less strict inter-element continuity criteria.

Chapter 4

Computational contact mechanics

Contact problems often involve multiple bodies, large deformations and multiple non-linearities. A computational solution scheme accordingly have to be especially robust and efficient to handle the complexity of the problems. The finite element method is the most widespread computational solution procedure. The approximated geometry and low inter-element continuity of FEM can be a challenge to the convergence of the solution. More recently contact problems have been solved by isogeometric analysis. It is shown that it has the potential to increase both accuracy and robustness of contact problems, see e.g (Hughes, 2011) where the NURBS based contact algorithm greatly improves the iterative convergence of the FEM equivalent. The great potential has resulted in various methods that have been developed for solving contact problems using IGA. Many of the methods are achieved by adapting procedures already existing for FEM. In this chapter, the methods used in the proposed contact algorithm in IGA are described. As the IGA methods inherit many of the features of the FEA equivalent it is relevant to also describe the finite element equivalent of the methods.

The contact solution procedures can be divided into two main parts. One being calculating the addition from contact to the weak form of the system equilibrium equation and the other being the contact interface discretization. In contact mechanics, the structural and contact contribution to the weak form are typically found and added to the equation separately, as is described in (Wrig-

gers, 2006). This chapter first presents how the contact contribution to the weak form is found and then how the contact interface and weak form is discretized.

4.1 The normal, frictionless contact problem

Assume two elastic bodies are subjected to large deformations and contact. One of the bodies is denoted as slave, Ω^s and the other as master, Ω^m. The contact problem is formulated in the perspective of the slave surface coming into contact with the master surface and not the opposite. This is the classical formulation of contact problems, even though it introduces a dependence on which body is given master status and which is given the slave status. The results are not equal when interchanging the status of the bodies and carefulness needs to be taken when making the choice.

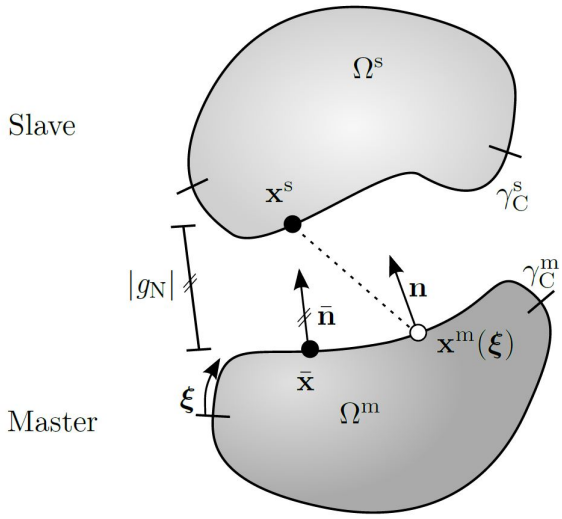


Figure 4.1: Description of contacting bodies, (Matzen, 2015)

The kinematic expressions are formulated in terms of the reference configuration of the bodies and their displacements. The reference configuration refer to the body at the last known position, or in terms of the Newton-Raphson iterative procedure it is the configuration of the body in the previous load step. The current configuration refer to the position and deformations of the body at the current load step in the Newton-Raphson procedure. Commonly and as is

done in (De Lorenzis, Scott, Wriggers, Taylor and Zavarize, 2014), a point in the current configuration of the body is expressed through the reference configuration point and the displacements up until the current position, $\boldsymbol{x}^i = \boldsymbol{X}^i + \boldsymbol{u}^i$, where $\boldsymbol{X}$ is the coordinate of a point in the initial configuration, $\boldsymbol{u}$ is the displacement of the point and the superscript i refers to either the slave or master body, $i = (s, m)$.

The contact problem is evaluated by locating contacting points on the master and slave surfaces, i.e. pairs of points in the contact interface. The contact interface in the current configuration is denoted γ_C. An assumption of perfect contact is made, which means for every point on the slave surface there is a unique contacting point on the master surface. Further, $\gamma_c = \gamma_C^s = \gamma_C^m$, m refer to the master and s to the slave surface. γ_C is however unknown and is determined through introducing a distance function $d = |\boldsymbol{x}^s - \boldsymbol{x}^m|$ which describes the distance between a fixed point on the slave contact surface γ_C^s and an arbitrary point $\boldsymbol{x}^m$ on the master surface contact region γ_C^m. The unique point on the master surface which is in contact with the fixed slave point $\boldsymbol{x}^s$ is the point of minimum distance between $\boldsymbol{x}^s$ and γ_C^m. It is often denoted $\bar{\boldsymbol{x}}^m$ and is computed by finding the closest-point projection of $\boldsymbol{x}^s$ onto γ_C^m. The closest-point projection is equivalent to minimising the distance function, d.

The slave point $\boldsymbol{x}^s$, its closest-projection point $\bar{\boldsymbol{x}}^m$ and the normal $\bar{\boldsymbol{n}}$ at $\bar{\boldsymbol{x}}^m$ are used to express the normal direction gap between the surfaces. Subsequently the bar of $\bar{\boldsymbol{x}}^m$ and $\bar{\boldsymbol{n}}$ is removed for simplicity of notation as the closest-projection point is hereafter the only point relevant on the master surface. The contact integrals are evaluated on the slave contact region of the reference configuration. The reference contact region is denoted Γ_C and hence $\Gamma_C := \Gamma_C^s \neq \Gamma_C^m$. When evaluating contact as a frictionless problem, contact is assumed to be relevant only in the normal direction. For this purpose, the normal directed gap between the two bodies is defined as

$$g_N = (\boldsymbol{x}^s - \boldsymbol{x}^m) \cdot \boldsymbol{n} \tag{4.1}$$

for which $\boldsymbol{n} = \boldsymbol{n}^m$ is the surface normal at $\boldsymbol{x}^m$, pointing towards the slave surface, see Figure (4.1).FE Due to how the gap function is defined, it is negative if the slave is penetrating the master surface and positive if contact is not occur-

ring between the points, i.e. when there is a positive distance between them. The normal traction due to contact, t_N is oppositely directed for the master and slave surfaces. The Piola traction vector is denoted $\boldsymbol{t} = \boldsymbol{t}^m = \boldsymbol{t}^s$ and its normal component is defined,

$$\boldsymbol{t} = t_N \boldsymbol{n}, \quad t_N = \boldsymbol{t} \cdot \boldsymbol{n}. \tag{4.2}$$

The contact normal traction and the gap function defines conditions for impermeability on Γ_C, called The Kuhn-Tucker conditions.

$$g_N \geq 0, \quad t_N \leq 0, \quad g_N t_N = 0 \tag{4.3}$$

4.2 The penalty method to formulate the contact weak form

The Kuhn-Tucker condition for impermeability, Eq. (4.3), is a boundary constraint that have to be fulfilled by the weak form. Contact between two bodies can accordingly be viewed as a constrained minimisation of the potential energy of the system, W. The penalty method is one of the most common methods to formulate the contact constraints and thus the contact contribution to the weak form. The contact constraints formulated using the penalty method are,

$$t_N = \epsilon_N g_N, \quad g_N = \begin{cases} (\boldsymbol{x}^s - \boldsymbol{x}^m) \cdot \boldsymbol{n}^m & \text{if } (\boldsymbol{x}^s - \boldsymbol{x}^m) \cdot \boldsymbol{n}^m < 0 \\ 0 & \text{otherwise.} \end{cases} \tag{4.4}$$

The penalty contact constraint in Eq. (4.4) defines contact as active, i.e. occurring, for points on the surface that has a normal gap function less than zero. That means after penetration of the slave body into the master surface has occured. The penetration g_N is then penalised by a constant penalty parameter $\epsilon_N > 0$. Areas for which $g_N = 0$ are not a part of the active contact region. The penalty formulation is called an active set strategy as it only calculates the contact contribution for parts of the surface that contact occurs.

The frictionless contact contribution to the weak form when applying the penalty method, W_C^P, is formulated in (Fischer, 2005) and (De Lorenzis, Scott, Wriggers,

Taylor and Zavarize, 2014) as

$$W_c^P = \int_{\Gamma_c} t_N \delta g_N dA = \epsilon_N \int_{\Gamma_c} g_N \delta g_N dA \tag{4.5}$$

where the integration is executed on the active contact region, using an active-set strategy as stated earlier. The linearisation of the weak form yields

$$\Delta\delta W_c^P = \epsilon_N \int_{\Gamma_c} \Delta g_N \delta g_N dA + \epsilon_N \int_{\Gamma_c} g_N \Delta(\delta g_N) dA \tag{4.6}$$

for which δ represents a variation and Δ the linearisation. The following equations formulate the variation and linearisation of the contact variables in vector and matrix form. The discretization is retrieved from (De Lorenzis, Scott, Wriggers, Taylor and Zavarize, 2014) though adapted to NURBS instead of T-splines which is use in the article. The quantities needed for the weak form equation can be written in matrix form. The following formulation is obtained,

$$\delta g_N = \delta \boldsymbol{u}^T \boldsymbol{N}_N, \quad \Delta g_N = \boldsymbol{N}_N^T \delta \boldsymbol{u} \tag{4.7}$$

where $\delta \boldsymbol{u}$ is the variation of the displacement vector, $\Delta \boldsymbol{u}$ are the linearized displacements and $\boldsymbol{N}_N$ are the NURBS functions described in a previous section multiplied by the contact normal. These terms are expressed in vector form as

$$\delta \boldsymbol{u} = \begin{bmatrix} \delta \boldsymbol{u}_1^s \\ . \\ . \\ . \\ \delta \boldsymbol{u}_{n^m}^s \\ \delta \boldsymbol{u}_1^m \\ . \\ . \\ . \\ \delta \boldsymbol{u}_{n^m}^m \end{bmatrix}, \quad \Delta \boldsymbol{u} = \begin{bmatrix} \Delta \boldsymbol{u}_1^s \\ . \\ . \\ . \\ \Delta \boldsymbol{u}_{n^m}^s \\ \Delta \boldsymbol{u}_1^m \\ . \\ . \\ . \\ \Delta \boldsymbol{u}_{n^m}^m \end{bmatrix}, \quad \boldsymbol{N}_N = \begin{bmatrix} R_1^s(\xi_s)\boldsymbol{n} \\ . \\ . \\ . \\ R_{n^s}^s(\xi_s)\boldsymbol{n} \\ R_1^m(\xi_m)\boldsymbol{n} \\ . \\ . \\ . \\ R_{n^s}^m(\xi_m)\boldsymbol{n} \end{bmatrix} \tag{4.8}$$

In the contact solution algorithm implemented in this thesis the linearisation of the displacements are not taken into account. It is assumed that it does not largely influence the convergence rate.

The NURBS basis functions for both the slave and the master side of the contact pair in the normal direction are gathered in a vector, $\boldsymbol{N}_N$. The same goes for the linearized and variations of the displacements. The terms related to the slave degrees of freedom are gathered in the upper part of the vector and the lower for the master degrees of freedom. $R_{1,\ldots,n^i}(\xi_i)$ are the basis function values from the basis functions having support on respectively a slave point $\boldsymbol{x}^s$ and its normal projection point $\boldsymbol{x}^m$. ξ_i are the parametric coordinates of the slave or master surface, $i = (s, m)$.

The second part of the weak form expressed in 4.5 includes the linearisation of the variation of the gap g_N,

$$\Delta(\delta g_N) = \delta \boldsymbol{u}^T \boldsymbol{k}_{geo} \Delta \boldsymbol{u} \tag{4.9}$$

The geometric stiffness matrix, $\boldsymbol{k_{geo}}$ contributes to a faster convergence of the code and takes into account the linearisation of the variation of the normal gap.

$$\boldsymbol{k_{geo}} = g_N \bar{\boldsymbol{N}} \boldsymbol{m}^{-1} \bar{\boldsymbol{N}}^T + \boldsymbol{D}\hat{\boldsymbol{N}}^T + \hat{\boldsymbol{N}}\boldsymbol{D}^T - \boldsymbol{D}\boldsymbol{k}\boldsymbol{D}^T \tag{4.10}$$

$\boldsymbol{m}^{-1}$ is the inverse of the metric tensor $m^{\alpha\beta}$ and $\boldsymbol{k}$ is the curvature tensor $k_{\alpha\beta}$. $\alpha = 1,2$ and $\beta = 1,2$ represents the directions of the surface. The following definitions to formulate the geometric stiffness matrix are introduced

$$\boldsymbol{T}_\alpha = \begin{bmatrix} R_1^s(\xi_s)\boldsymbol{\tau}_\alpha \\ \vdots \\ R_{n^s}^s(\xi_s)\boldsymbol{\tau}_\alpha \\ R_1^m(\xi_m)\boldsymbol{\tau}_\alpha \\ \vdots \\ R_{n^s}^m(\xi_m)\boldsymbol{\tau}_\alpha \end{bmatrix}, \quad \boldsymbol{N}_\alpha = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ R_{1,\alpha}^m(\xi_m)\boldsymbol{n}_\alpha \\ \vdots \\ R_{n^s,\alpha}^m(\xi_m)\boldsymbol{n}_\alpha \end{bmatrix} \tag{4.11}$$

$$\hat{\boldsymbol{N}} = [\boldsymbol{N}_1 \quad \boldsymbol{N}_2], \quad \hat{\boldsymbol{T}} = [\boldsymbol{T}_1 \quad \boldsymbol{T}_2] \tag{4.12}$$

$$\boldsymbol{D} = [\hat{\boldsymbol{T}} - g_N \hat{\boldsymbol{N}}]\boldsymbol{A}^{-1}, \quad \bar{\boldsymbol{N}} = \hat{\boldsymbol{N}} - \boldsymbol{D}\boldsymbol{k} \tag{4.13}$$

where the subscript 1,2 refers to surface direction $\alpha = 1,2$ in Eq. (4.11) and $\boldsymbol{A}^{-1}$ is the inverse of $A_{\alpha\beta} = m_{\alpha\beta} - g_N k_{\alpha\beta}$. Moreover, the covariant vectors of the master surface are $\boldsymbol{\tau}_\alpha = \boldsymbol{x}^m_{,\alpha}$. $\boldsymbol{m}^{-1}$ is the inverse metric tensor, $\boldsymbol{k} = k_{\alpha\beta}$ is the curvature

tensor in local directions $\alpha, \beta = 1,2$ on the master surface.

All the metric components and the curvature tensor depends on the master surface and are thus calculated for the master element resulting from the point projection.

General comments to the method

The penalty term ϵ_N can be considered a spring stiffness. It is multiplied by the penetrated distance to achieve the force needed to push the slave body back into a position of perfect contact, $g_N = 0$. It is shown among others in (Luenberger and Ye, 2016) that $\epsilon_N \to \infty$ and $\epsilon_T \to \infty$ corresponds to the exact solution. This is though impossible to achieve as very high penalty parameters ϵ results in ill-conditioned stiffness-matrices. Large penalty parameters are thus subject to a loss in robustness and numerical errors, (De Lorenzis, Hughes and Wriggers, 2014). Typical penalty parameters used in the literature are $\epsilon = 1e3 - 1e5$, see e.g. (De Lorenzis, Scott, Wriggers, Taylor and Zavarize, 2014). The penalty method is advantageous in its simplicity and as stated in (Fischer, 2005), it is purely geometrically based. Therefore no additional degrees of freedom have to be activated or deactivated. Lagrange multiplier method is another of the most common methods to impose contact constraints on a structural problem. It is able to fulfil the condition of impermeability exactly. The disadvantage of this method is that additional unknowns are introduced into the stiffness matrix, which is avoided for the penalty method.

4.3 Contact space discretization with FEM and IGA

This section aims to describe how the contact contribution to the weak form is adapted to a discretized setting. The basic criteria for a contact space discretisation is that it must be able to handle contact between non-matching meshes. As contact typically is related to large deformations and sliding, the discretised setting must not be dependent on a matching mesh between the bodies. There exists several methods varying in complexity, accuracy and robustness. In this chapter the methods directly related to the methods implemented into the proposed contact algorithm are briefly reviewed and the equations needed to setup the discretisation scheme used in the algorithm are provided.

Node-to-segment contact discretization

On of the earliest discretisation schemes for contact problems with non-matching mesh, is the Node-To-Surface, NTS, algorithm. This method is still widely used in commercial finite element codes due to its simplisity and flexibility. The NTS contact discretisation enforce contact constraints between a node on the slave surface and a segment of the master body, see Figure (4.2). The NTS approach needs to be combined with an active-set-strategy, which here implies that only the slave nodes related to a gap $g_N \leq 0$ are included in the contact calculations.

The NTS approach is computationally inexpensive and flexible. Its major drawback stems from the way the contact pressure is transferred from the slave to the master body. The ability of a contact algorithm to transfer stresses is often tested through a patch test, which is a simple indicator of its quality. It is shown in (Zavarise and De Lorenzis, 2009*a*) that the approach transfers stresses as concentrated forces at the slave nodes. This again results in the balance of momentum not being achieved on an element level. Another challenge with the method is its bias between which body is given master and slave status. It have to be kept in mind how interchanging the status of the bodies can interfere with the results.

Knot-to-surface and Gauss-point-to-segment discretization

In IGA, the node-equivalent control points are not positioned at the actual surface of the body. In (Hughes, 2011) it is shown that the control points lay significantly outside of the contact surface during the contact stages. Control points can thus not be directly used as a reference for contact on the slave surface. The gap would not be the actual distance between the surfaces. In order to employ a similar strategy as the NTS contact discretization in IGA, other points needs to be used. In (Hughes, 2011), Gauss-Legendre quadrature points are used to enforce the contact constraints. The algorithm is denoted knot-to-surface, KTS. Compared to the standard C^0 continuous Lagrange finite elements, the procedure proves to provide a more physically acting positive contact pressure, whereas the Lagrange finite elements are prone to produce negative pressures. When two elastic bodies are subjected to large deformations and sliding, the KTS approach shows a greatly improved iterative convergence. The KTS algorithm is overall said to provide satisfactory results. It is however suggested to

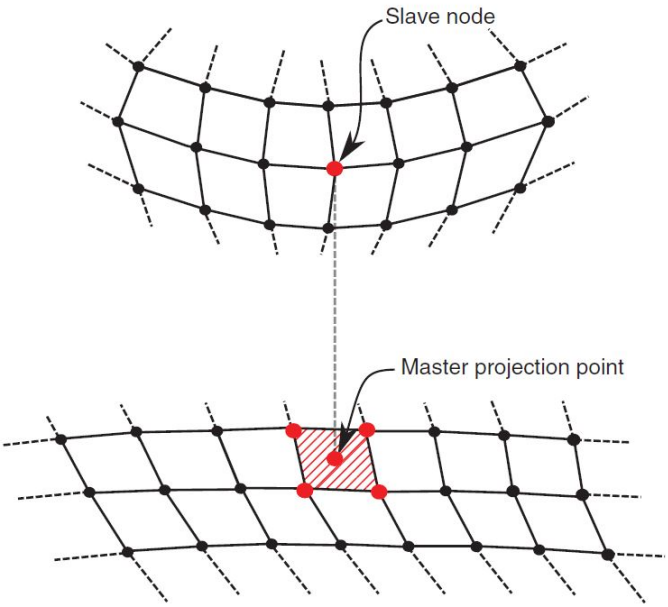


Figure 4.2: NTS discretization, (De Lorenzis et al., 2017)

rather use the more sophisticated Mortar methods than directly applying the constraints at the Gauss-Legendre points. Mortar methods introduce a reference surface and enforce the contact constraints in a weak sense. Mortar methods are more robust than the NTS approach also for common FEM, the methods are described in detail in (Wriggers et al., 2006). Due to its complexity it typically increases the computational cost.

Later the KTS method has been denoted Gauss-point-to-segment, GPTS, as is done in (De Lorenzis, Hughes and Wriggers, 2014). The GPTS discretization scheme is also in (De Lorenzis, Hughes and Wriggers, 2014) described as having the ability to capture the contact surface even for a low number of elements. It is not dependent on the mesh of the structure in the direct sense which the classical NTS approach is.

A drawback of the method is, as stated in Matzen et al. (2013), the non-matching number of virtual knots, here Gauss points, and degrees of freedom at the contact interface makes the system over-constrained which might lead to convergence problems. In (Matzen et al., 2013), Greville and Botella points are used as

collocation points. The number of Greville and Botella points are always equal to the number of control points of a surface and the challenge of having an over-constrained system can be avoided. Moreover, the Gauss points are not able to capture contact at edges as they are situated on the inside of the elements. Using other collocation points or more sophisticated methods might be beneficial for future development of the IGA contact algorithm proposed, but as a first contact implementation to the Marine Department IGA research code, the GPTS discretization is a natural choice. It has an advantage in its simplicity of implementation and it is considered to be sufficiently accurate and robust for a variety of contact problems.

4.4 GPTS discretisation equations

The contact contribution to the weak form is calculated for GPTS contact elements. All basis functions that have weights in the Gauss point from the slave element and in the normal projection point from the master element are included in the contact element. The related stiffness matrix and residual force vector thus includes all degrees of freedom related to the slave element and the master element for which the points $\boldsymbol{x}^s$ and $\boldsymbol{x}^m$ are a part of. For each active Gauss point on the slave surface, a contact element is established and its contribution to the weak form and force residual vector is calculated. The structure of the contact element stiffness matrix K_{gp} and residual force vector $\boldsymbol{R}_{gp}$ is shown in Eq. 4.14. The subscript s, m refers to the degrees of freedom related to the slave and master elements involved and gp indicates that the term belongs to the contact element of a specific Gauss point.

$$K_{gp} = \begin{bmatrix} k_{ss} & k_{sm} \\ k_{ms} & k_{mm} \end{bmatrix}, \quad R_{gp} = \begin{bmatrix} R_s \\ R_m \end{bmatrix} \tag{4.14}$$

The equations related to the GPTS discretization using the penalty method are taken from (De Lorenzis, Scott, Wriggers, Taylor and Zavarize, 2014). The total stiffness matrix and force residual, $\boldsymbol{F}_r$ due to contact is formulated as

$$\boldsymbol{F_r} = \epsilon_N \int_{\Gamma_c} g_N \boldsymbol{N} dA \tag{4.15}$$

with ϵ_N being the penalty parameter to be applied in the normal direction. Discretised with the GPTS method, that yields

$$\boldsymbol{F_r} = \epsilon_N \sum_{activegp} g_{Ngp} \boldsymbol{N}_{gp} z_{gp} J1_{gp} J2_{gp}. \tag{4.16}$$

The summation is executed over all active Gauss points. z_{gp} are the Gauss-Legendre weights associated with the Gauss point, $J1_{gp}$ and $J2_{gp}$ are the Jacobian functions mapping from isoparametric space to physical space and mapping to the reference domain that the Gauss quadrature points and weights are defined for. g_{Ngp} is the normal gap function at a specific Gauss point.

The contact contribution to the stiffness matrix is calculated in two parts related to the two parts of the linearised virtual work, Eq. (4.6). The main contact stiffness contribution comes from the first part and the geometric stiffness contribution from the second part related to the linearisation of the variation of the normal gap function, Eq. (4.9).

$$\boldsymbol{K}_c = \boldsymbol{K}_{c,main} + \boldsymbol{K}_{c,geo} \tag{4.17}$$

$$\boldsymbol{K}_{c,main} = \epsilon_N \int_{\Gamma_c} dA \boldsymbol{N} \boldsymbol{N}^T dA \tag{4.18}$$

$$\boldsymbol{K}_{c,geo} = \epsilon_N \int_{\Gamma_c} g_N \boldsymbol{K}_{geo} \boldsymbol{N}^T dA \tag{4.19}$$

The contribution to the stiffness matrix discretised with the GPTS method yields,

$$\boldsymbol{K}_{c,main} = \epsilon_N \sum_{gp,active} \boldsymbol{N}_{gp} \boldsymbol{N}_{gp}^T z_{gp} J1_{gp} J2_{gp} \tag{4.20}$$

$$\boldsymbol{K}_{c,geo} = \epsilon_N \sum_{gp,active} g_{Ngp} \boldsymbol{k}_{geo,gp} z_{gp} J1_{gp} J2_{gp} \tag{4.21}$$

for which N_{gp} contains all NURBS basis functions R, related to the contact ele-

ment for the specific active Gauss point.

Chapter 5

Implementation of a contact algorithm in MATLAB

The general goal of a structural analysis computational tool is to sufficiently provide accurate results and do so with a minimal use of resources. Applying IGA instead of FEM has shown to both improve efficiency and accuracy. The need for elements in the mesh is reduced and the geometry is described more accurately. As of this moment there are no commercial software that offers contact analysis with IGA using Kirchhoff-Love shell elements. In order to contribute to the research currently being done in this field, a contact algorithm is proposed and implemented into the IGA research code of The Marine Department of NTNU. This chapter presents the proposed contact algorithm. The contact code can be seen in the Appendix of this thesis.

Choosing MATLAB

MATLAB is used for the computational implementation of the contact problems in this thesis. As a high-level programming environment it has the advantage of being an orderly and easy to understand tool for structural analysis. In IGA as well as FEM, the handling and solving of matrices is fundamental. The numerical handling of matrices is straightforwardly done in MATLAB, which is an abbreviation of MATrix LABoratories. Matrices can easily be passed on be-

tween functions and solved directly due to the built-in functions. Moreover, the built-in visualisation scheme comes in handy when plotting displacements and structure geometry. The drawback of using the MATLAB environment is noticeable when the structural problem becomes large with a vast number of degrees of freedom. It is slow compared to low-level programming languages and measures to ensure efficiency should be kept in mind during the implementation. The IGA research code uses *mexfile* subroutines coded in Fortran in order to decrease the computational time.

Choosing methods for contact analysis

The procedure for setting up a contact algorithm in a structural analysis using IGA, as well as FEM, can be roughly considered to consist of two main steps. The first is the enforcement of contact constraints. The second is expressing the contact contribution to the weak form in a discretised setting. There exist several methods that describe how either step can be implemented. The focus of the script proposed in this analysis is to develop a simple to implement contact procedure as a first contact addition to the IGA research code with a sufficient degree of accuracy from an engineering perspective. The computational resources needed for an analysis should be limited and the implementation should be as general as possible so that it is easily adjusted to analyse a number of geometrical examples.

In the proposed contact implementation the penalty method is used for imposing contact constraints and then Gauss-point-to-segment method is used to add the contact contribution to the discretized weak form. These methods are described in the previous chapters. The main disadvantage of using the penalty method as a contact constraint formulation is the nonphysical penetration allowance. On the other hand, its simplicity for which it does not involve including more unknowns to the equations, makes it a natural choice as a part of a first contact implementation. The GPTS formulation is a relatively intuitive discretization scheme that detects contact between Gauss quadrature points on the slave surface and the normal projection point on the master surface. The GPTS approach with the penalty method was implemented in (De Lorenzis, Scott, Wriggers, Taylor and Zavarize, 2014) with T-splines instead of NURBS for which satisfactory results were achieved. For very large penalty parameters, the

GPTS approach showed an oscillatory behaviour in (De Lorenzis et al., 2011). This should not be a problem in the proposed implementation since a high degree of accuracy is not the main objective of the implementation and very high values of penalty is not considered. Keeping the penalty parameter low enough also ensures that the matrices are not ill-conditioned.

5.1 Code overview

The contact code is setup such that one *main function* defines the bodies involved in the analysis and all the prescribed parameters that influence the execution of the analysis. All parameters and specifications needed for the contact problem implementation are defined in the main function, keeping in mind the user should not have to edit other functions than the main function itself to adjust the analysis to a specific contact problem.

The execution and solution of the analysis is controlled by a *solver function* that is called from the *main function*. The solver function applies Newton-Raphson iterations to achieve the solution. Moreover, there are different solver functions implemented to handle some variations in the analysis which are described later in this chapter. The alternative implementations are chosen by specifying in the *main function* which *solver function* that is to execute the analysis. During the solution process the *solver function* call a function that detects contacting surfaces and calculate the contact contributions to the weak form. This function is denoted as a *contact contribution function*. The contact contribution functions are adapted to the specific solver function.

An overview of how the contact code is divided into three main parts can be seen in the table below. The fundamental setup of the main function and solver function are taken from the IGA research code. They are added and adjusted for contact analysis in this thesis for which the features are described in this chapter. The contact contribution calculations are fully created as a part of this thesis.

	MAIN FUNCTION	SOLVER FUNCTION	CONTACT CONTRIBUTION FUNCTION
Main task:	Defining bodies involved in the analysis and all prescribed parameters	Controlling the Newton -Raphson iterational procedure that solves the system.	Detect normal contact between bodies and calculate the contact contribution to the global stiffness matrix and residual force vector.
Interaction with other contact functions:	Calls a solver function →	Calls a contact contribution function →	← Returns results to solver function

5.2 Solution algorithm

To adapt the Newton-Raphson solution procedure for contact problems of varying geometry, some features are included in addition to the standard penalty and GPTS formulation. The solution procedure proposed is described in this section by going through the contact specific features implemented. A stepwise summary of the complete solution algorithm is seen in Figure (5.4).

Contact normal sign check

In contact analysis the normal vector of the master surface plays a fundamental role in detecting if contact occur, see Eq. (4.4). If the value of the normal gap function is less than zero, contact is invoked by the penalty method. Accordingly the normal projection point on the master surface have to have a normal pointing towards the slave body. Kirchoff-Love shell elements are used for the implementation. The direction of the normal for this element is either in the outward or inward direction. To ensure the correct direction of the normal, a check is added in the beginning of the solver function.

For the normal check it is assumed that the bodies are not initially in contact, that is before any external load is applied. The normal gap g_N is then calculated for two arbitrary points on the master and slave body. If the gap function is negative, the master surface normal, has initially, $\boldsymbol{n}_{shell}$, the opposite sign than the

contact normal to be used for contact detection, $\boldsymbol{n}$. A constant parameter storing the contact normal sign is created, n_{sign}. It possess the value -1 if the initial master normal points in the wrong direction and 1 if it is initially correct. The shell normal does not change direction in the analysis and it is thus only necessary to calculate the sign parameter once. The master normal sign parameter is passed on into the contact contribution function where the contact detection is implemented. The algorithm is step-wise summarised in text-box 1.

Box 1: *Contact normal check*

$\boldsymbol{p_m}$ = arbitrary master point

$\boldsymbol{p_s}$ = arbitrary slave point

$\boldsymbol{n_{shell}}$ = the shell normal at $\boldsymbol{p_m}$

In solver function:

- Calculate normal gap: $g_N = (\boldsymbol{p_s} - \boldsymbol{p_m}) \cdot \boldsymbol{n_{shell}}$
- Check sign: If $g_N < 0 \rightarrow n_{sign} = -1$, if $g_N > 0 \rightarrow n_{sign} = 1$

In contact contribution function:

- $\boldsymbol{n} = \boldsymbol{n_{shell}} \cdot n_{sign}$
- To detect contact, calculate gap: $g_N = (\boldsymbol{p_s} - \boldsymbol{p_m}) \cdot \boldsymbol{n}$

Multi-patch contact

The solution algorithm is implemented so that it can handle contact between more than two NURBS patches. This is the case when more than two bodies is involved in the contact problem and when a body is defined by multiple NURBS-patches. Multiple NURBS-patches are for example often used to described circular shapes. The multi-patch implementation is based on two main objectives.

1. Have the possibility to define multiple bodies as master or slave.
2. Have the possibility to leave contact irrelevant bodies out of the contact calculations.

The last objective is implemented to restrain the computational cost.

In order to define which patches are masters and which are slaves as well as which patches might come into contact, *master-slave pairs* are defined in the main function. They are then passed on into the solver function. The IGA re-

search code of NTNU assigns a unique number to all the NURBS-patch geometries that are created in the analysis. These patch numbers can further be used to assign slave and master status to a certain patch. This is done through creating an individual vector for each pair of patches that might come into contact, for which the first entry is the patch number to be considered as slave and the last is the patch number to be considered as master in the pair: $[slavenr, masternr]$. One patch can be in contact with multiple other patches by simply including it in several master-slave pairs. The way the master-slave pairs are defined makes it very simple to interchange which body is defined as master and which is slave in order to check the dependence on the choice.

The solver function only calculates the contact contributions for the defined master-slave pairs. This is executed by looping through each pair and passing on the current master and slave patch numbers as arguments into the contact contribution function. Hence, the master-slave pairs are used to fulfil also the second criteria, by limiting which bodies are considered for contact.

The normal vector check has to be adjusted for multi-patch contact. The normal vector check is for the multi-patch implementation executed for each master-slave pair individually and stored so that the correct sign can be passed on to the contact contribution function that uses the sign in the gap function calculation.

The solution algorithm including multi-patch contact i summarised in text-box 3.

Contact by updating displacement vector

A possibility to induce contact by moving rigid bodies onto elastic bodies is implemented. The load steps in the Newton-Raphson iterations for this implementation does not include load increments, but updates the coordinates of a patch an incremental length for each step instead. For this purpose, two parameters are added to the main function. The first parameter contains the patch number for the patch that is to be moved an incremental length for each load step, $d\_rigid.part\_nr$. The second parameter decides the total distance the body is to be moved and in which Cartesian direction, $d\_rigid.d\_step\_dir = [dx, dy, dz]$. Multiple patches can be moved by specifying more patch numbers with individual total displacement vectors.

In the load step, all patches that are to be moved are looped through. The control point coordinates are added a displacement increment in the x, y, and z-direction. The displacement increment is found by dividing the total displacement in a direction by the total number of load steps to be used in the Newton-Raphson procedure. The calculation of contact contributions does not need to be adjusted.

5.3 Calculating the contact contribution

For every iteration in the Newton-Raphson procedure, the contribution to the global stiffness matrix and residual force vector from contact is calculated. The structural and contact contributions are calculated separately and added directly to the global matrix and vector. Separately calculating the structural and contact contributions in the analysis is standard procedure and is described in for example (Wriggers, 2006). An advantage of the separate calculation and addition for the structural and contact contribution tot he weak form is that the discretisations are independent of each other and the contact contribution can thus be discretised without considering the method used for the structural stiffness matrix.

In the proposed contact implementation, contact in the normal direction is considered. The GPTS discretisation is used together with the penalty method. The implementation of the contact contributions calculation implementation is summarised in the text-box below.

Finding the normal projection point

The algorithm that finds the normal projection point related to a point on the slave surface was already a part of the IGA research code of NTNU and is not created for the proposed contact implementation. It is briefly described here as its features influence the point search procedure propose in the next section.

A point on the master surface, $\boldsymbol{x}^m$ is considered a normal projection point to a certain slave point when the dot product of the normal $\boldsymbol{n}$ at the master point and a tangent vector at the same point $\boldsymbol{a}_\alpha^m$ is zero, as described in (Wriggers, 2006). α represent either parametric direction. The normal projection point

Box 2: *Contact contributions calculation implementation*

LOOP over elements on slave body
LOOP over Gauss points on the slave element

- Find normal projection point from the Gauss point onto the master surface
- Check the normal gap function: If $g_N < 0 \rightarrow$ continue. If $g_N \geq 0 \rightarrow$ go to the next Gauss point.
- Calculate the stiffness matrix for the GPTS contact element
- Calculate the residual force for the GPTS contact element
- Add to global stiffness matrix and residual force vector

END GAUSS POINTS LOOP
END SLAVE ELEMENTS LOOP

search algorithm accepts a normal projection point as long as the error is less then a tolerance, see Eq. (5.1). The tolerance of perpendicularity is for this implementation set to $1e-4$.

$$\boldsymbol{n} \cdot \boldsymbol{a}_{\alpha}^{m} \leq tol \tag{5.1}$$

A starting point for the normal projection point algorithm is defined in the main function. This is a constant starting point which implies the starting point for the point projection algorithm is the same for all Gauss points on the slave surface. The algorithm is initialised by first checking Eq. (5.1) for the starting point on the master surface. If it is fulfilled, the function exits and returns the point coordinates. If the calculated value is not below the tolerance, the algorithm moves a small step on the surface and checks another point. It continues the procedure, moving in the direction that shows a reduction in the calculated dot product value, until the equation is fulfilled.

The drawback of using Eq. (5.1) for finding the normal projection point, is that the uniqueness of the normal projection point is not guaranteed. The master surface might have multiple points that results in a zero vector product. A sphere will for example always have two possible points, one on either side of the body.

Another challenge arise for complicated geometries. As the point projection algorithm moves in the direction along the master surface that exhibit a reducing dot product, it is not able to move onto the other side of arched geometries if the starting point is on the wrong side of the master surface. As the starting point for the normal projection point algorithm is constant for all Gauss points on the surface, this is likely to occur for several geometric examples. It is thus greatly dependent on the choice of starting point on the master surface. In the next section, a point search procedure consisting of two more steps is proposed hat might improve the issues stemming from having poorly chosen starting points for the normal point projection algorithm.

5.4 Complete solution algorithm

In text-box 3 below a summary of the complete solution algorithm implemented is displayed. The residual force vector is denoted R, the total number of load steps in the analysis is N, the global stiffness matrix is denoted K. The contributions from contact or structural formulations denoted with a C for contact and S for structural. The displacement vector in the current load step is denoted d^k and the displacement for the next load step is denoted d^{k+1} and Δ is the symbol for an increment. When the norm of the residual gets below a specified tolerance, the iterations in the *solver function* are stopped. The solution procedure can then go on to the next load increment and the procedure is repeated.

Box 3 *Multi-patch contact using GPTS and penalty formulation.*

Check contact normal sign

LOOP over load increments

If contact is induced by external load:

- Calculate current load increment

OR

If contact is induced by moving a rigid body:

LOOP rigid patches to move

- Displace control points one displacement increment

END PATCHES TO MOVE LOOP

LOOP over iterations, $k = 1,...,$convergence

- Calculate structural stiffness matrix and contribution to residual force vector K^S and R^S.
- Add structural contributions to global residual force vector and stiffness matrix: $K = K + K^S$, $R = R + R^S$.

LOOP master-slave pairs

- Check contact condition for all Gauss points: $g_N < 0 \rightarrow$ active contact $\rightarrow$ calculate contribution to stiffness matrix and residual force vector K^C and R^C.
- Add contact contributions to global residual force vector and stiffness matrix: $K = K + K^C$, $R = R + R^C$.

END MASTER-SLAVE PAIRS LOOP

- Solve: $K(d^k)\Delta d^k = -R(d^k)$.
- Check for convergence: $|R| \leq tol \rightarrow$ stop.
- Update solution: $\Delta d^{k+1} = d^k + \Delta d^k$.

END ITERATIONS LOOP

END LOAD INCREMENTS LOOP

It is possible to also update the penalty parameter within the iterations loop. It is suggested in (De Lorenzis et al., 2017) to increase the penalty parameter for each iteration in a specific manner, which gradually drives the solution vector

closer to the converged result. This is not implemented in the current algorithm and remains as a suggestion for further work. Updating the penalty parameters during the iterations can help decrease the penetration error.

5.5 Solution algorithm with two step point search

The normal projection point algorithm in the IGA research code is dependent on the starting point chosen at the master surface for the normal projection point search. As the starting point for the normal projection point search is constant throughout the load steps, it is likely to become less suitable after some deformation. The normal point projection algorithm was first developed for the purpose of coupling patches of the same structure that share a common edge and not for contact analysis.

In the first proposed contact algorithm, a normal projection point onto the master is found for every Gauss point on the slave surface. Commonly there are not all slave elements that are relevant for contact in every load step. Finding the normal projection point for all points on the slave surface is thus unnecessary and inefficient. A two step point search procedure is proposed in this section to improve the performance of the contact point search between slave and master surface. The two step search procedure proposed improves the starting point for the normal projection point algorithm by creating individual starting points for each slave element. The procedure also sorts out slave elements not relevant for contact in order to improve computational cost.

The procedure is denoted a two step search procedure as it adds a step before the normal projection point related to a Gauss point is calculated. This added step, step 1, can be divided into two parts: *a* and *b*. Step *1a* assigns individual starting points for the normal projection algorithm and step *1b* sorts out slave elements that are not contact relevant.

Step 1 a: Assigning individual starting points

In two step point search solver functions, each slave element is assigned an individual starting point on the master surface for the normal projection point search. The starting point is updated for each load step in the Newton-Raphson

load step loop so that it is up to date with the latest displaced configuration of the body. In this manner, the algorithm is less likely to encounter the problem of not finding the normal projection point due to geometry and also the problem of finding the wrong point. Improving the starting point also has the potential in improving the computational speed. The normal projection algorithm needs less iterations to find the normal projection point if the starting point is close to the actual normal projection.

The objective of the two step point search procedure is to both improve robustness and computational effort. It is thus important to find an acceptable compromise between the closeness of the starting point and the speed of the algorithm. An option would be to assign individual starting points for every quadrature point for each slave element. This would though be time consuming and is considered excessive. As a compromise, the centre point of each slave element is calculated and used as the reference position of the element. In this manner, only the elements are looped through and not the individual Gauss points. The centre point of an element is calculated through using the relation between polynomial degree in a parametric direction and entries in the knot vector,

$$u_{centre} = \frac{U(p+1) + U(p+2)}{2}, v_{centre} = \frac{V(q+1) + V(q+2)}{2} \tag{5.2}$$

for which u_{centre}, v_{centre} are the parametric surface coordinates in both direction with the related polynomial degree p, q and knot vector U, V.

The points on the master surface to compare to the slave element are chosen to be the centre points of all master elements. These points serve both as the starting point for the normal projection search and as a reference position for the master element to be used for a distance calculation. The absolute distance between the slave element centre and master element centre is used to find which element on the master is the closest to a specific slave element. Accordingly all master elements are looped through for a specific slave element and the centre point position of the current master element in the loop is calculated. The absolute distance between the centre point of the slave and master elements are calculated and hereafter denoted the reference distance between the slave and master element, L_{ref}. The current reference distance, L_{ref} is compared to the the so far calculated minimum distance, L_{min}. If the distance between the slave

element and the current master element is smaller than L_{min}, it is assigned to L_{min}: $L_{min} = L_{ref}$. The centre point of the respective master element is stored as well, $u_{centre,m}$ and $v_{centre,m}$. When all master elements are looped through, the remaining smallest reference distance between the slave element and the master surface is the actual smallest distance and the related master element centre parametric coordinates is the starting point for the normal projection search.

The minimum distance L_{min} is stored for each slave element in a matrix. $u_{centre,m}$ and $v_{centre,m}$ are stored in the same row in the matrix: $M_{L_{min}}(iel_s) = [u_{centre,m}, v_{centre,m}, iel_m, L_{min}]$. Here iel_m is the element number for the closest master element to the slave element with element number iel_s. The result is a matrix containing all reference closest distances between the slave elements and the master surface together with the related parametric coordinate of the closest master element centre point and the element number of this master element. All Gauss points within the same slave element have the same assigned starting point for the normal projection point search. An overview of the procedure is found in text-box 4. It is implemented into a solver function for two step point search. For multi-patch contact the procedure is executed for all contact pairs.

Step 1 b: Sorting out slave elements not relevant for contact

The second part of the two step point search procedure proposed, address the objective of lowering the computational cost. The first part also contributes by reducing iterations in the normal projection algorithm. The second step sorts out slave elements that are so far away from the master surface that they are not considered relevant for contact. Only elements considered relevant are passed on into the contact contribution calculation loop that checks for contact by calculating the normal gap function. Moreover, avoiding some slave elements from entering the contact contribution calculations further reduce the risk of having poor starting points that results in the normal projection point algorithm being unable to find the correct normal projection point.

The sorting procedure is initialised in the main function. A parameter is defined, containing the absolute acceptable distance between a slave element and master surface, L_{acc} for the element to be considered for contact. This distance needs to be adjusted manually for each geometrical problem. As a part of the

solution procedure, a matrix that is to contain all slave elements considered for contact is created, M_{acc}. It has the same number of rows as the total number of elements in the slave patch. The rows contains the same entries as the $M_{L_{min}}$ except that some rows have zero entries. The zero entry rows represent the slave elements considered irrelevant. They are sorted out in the same loop as when assigning individual starting points. After all master elements have been looped through for a slave element, the reference minimum distance L_{min} is final. If L_{min} is less than the acceptable distance L_{acc}, the row related to the slave element in the matrix $M_{L_{min}}$ is added to the new matrix for contact relevant elements, M_{rel}. Since values are only added to the rows of slave elements that fulfil the acceptable distance for contact consideration, all other rows have zero in all entries.

When contact contributions are calculated, all slave elements are looped through. A check is added at the beginning of the loop to see if the element is to be considered for contact or not. If the entries of the M_{rel} related to the slave element number are zero, the loop is exited and the contact contributions calculation are stopped for this element. The procedure is summarised in text-box 5.

5.6 Chapter summary

The contact algorithm proposed for this thesis is described in this section. It consists mainly of three parts: A main function setting up the NURBS patches and initialising the analysis, a solver function that controls the Newton-Raphson iterations that solves the system and a contact contribution function that creates the contact element using the Gauss point to segment approach together with the penalty method.

The functions are inspired by existing codes in the IGA research code of the Department of marine Technology, NTNU. Contact analysis was not a part of the original research code and a first contact algorithm is proposed in this thesis. A summary of the features added as a part of the proposed contact algorithm are listed below.

- Contact contribution calculation using the GPTS approach and penalty method.
- Multi-patch contact definition
- Moving rigid body in Newton-Raphson procedure instead of applying external load.
- Contact normal sign procedure
- Two step point search procedure

In addition, the geometric modelling of the numerical examples are setup for this thesis. The user input defining the contact analysis which are prescribed in the main function, are listed in Appendix A. They are stored as a part of a MATLAB struct variable *anls.contact.PARAMETER*.

Box 4: *Assigning individual starting points for the normal projection point algorithm.*

Preallocate: Closest distance matrix M_{dist}
LOOP load steps in Solver
LOOP master-slave pairs
LOOP slave elements

- Calculate coordinates of centre point of current element: $\boldsymbol{x}_{center,s}$

Initialize minimum reference distance to a large number: $L_{min} = 1000$
LOOP master elements

- Calculate coordinates of centre point of current element: $\boldsymbol{x}_{center,m}$
- Calculate absolute distance: $|L_{ref,i}| = |\boldsymbol{x}_{center,s} - \boldsymbol{x}_{center,m}|$
- Compare current distance to minimum distance: If $|L_{ref}| < L_{min} \rightarrow$ set $L_{min} = |L_{ref}|$ and add parametric coordinates of $\boldsymbol{x}_{center,m}$ to $M_{L_{min}}$ at the row related to the slave element number in the loop.

END loop master elements

- Compare minimum distance to acceptable contact distance: If $L_{min} < L_{acc} \rightarrow$ add the row in $M_{L_{min}}$ related to the slave element number in the loop to M_{acc}.

END loop slave elements
END loop master-slave pairs
LOOP iterations

- Calculate structural contribution

LOOP master-slave pairs

- Calculate contact contribution: Only calculate contributions for relevant slave elements.

END loop master-slave pairs

- Solve system

END iterations
END load steps

Box 5: *Contact contribution calculation with two step search*

LOOP slave elements

- Check if the row in M_{acc} related to the slave element number in the loop has zero entries. If entries are not zero → continue. if not → break and continue to the next slave element.

LOOP Gauss points on element

- Find normal projection point from the Gauss point to the master surface: Use the parametric coordinates of the master element centre points in M_{acc} for the respective slave element.
- Check the normal gap function: If $g_N < 0$ → continue. If $g_N \geq 0$ → go to the next Gauss point.
- Calculate contact contribution for the GPTS contact element.

END loop Gauss points

END loop slave elements

Chapter 6

Numerical examples and discussion

In the following chapter some numerical contact examples are implemented and discussed. The first example consist of one rigid body that is moved so that it comes into contact with an elastic body. The second example consist of two elastic bodies subjected to external load and lastly a more complex example taken from a PhD thesis, (Matzen, 2015), is implemented. The last example provides a test as to how well all the features of the implemented contact code works together. It includes both multi-patch contact, the 2 step point search algorithm and moving of rigid bodies to induce contact.

6.1 Rigid plate falls down on elastic arch

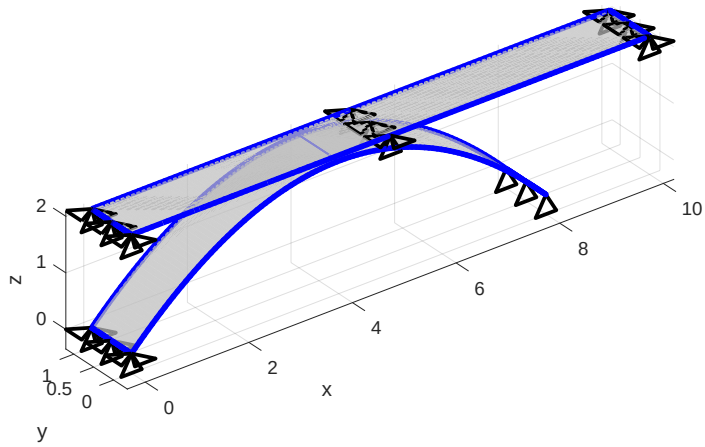


Figure 6.1: Flat rigid and arch: Boundary conditions on initial configuration

Problem description:

The first example implemented consists of an elastic arch being pressed down until it is flattened completely by a rigid plate. The GPTS and penalty solution procedure is used with the rigid given the master status. A Newton-Raphson iterational scheme is used to solve the problem. The calculation of contact contribution is done without including the K_{geo} contribution to the stiffness matrix. A mesh refinement and dependence on the choice of penalty parameter is executed.

The arch is simply supported on the right and left edges. All degrees of freedom of the flat structure are blocked. The boundary conditions are illustrated in Figure (6.1). Since the system is only solved for the unblocked degrees of freedom, the flat structure never actually enters the solution calculations. The plate is displaced an increment in the z-direction for each load step. 5 load steps are used to displace the flat structure a total distance of $2m$ downwards. The arch has its highest point at $z = 2m$ in the initial configuration and the flat rigid body

is situated at $z = 2.1m$. The rigid body is slightly above the arch so that contact is not occurring before the displacements are initiated. The initial configuration, before the solution procedure is executed, is illustrated in Figure (6.1). A polynomial degree of 2 is used in both parametric direction u and v.

Material parameters:

The same material parameters are applied to both the rigid flat plate and the arch. The parameters are irrelevant for the rigid surface as all degrees of freedom are blocked.

Shell thickness $t = 0.05m$, $E = 1e7N/m^2$ and $\nu = 0.0$.

Analysis specifications:

The solution functions used for this analysis are listed below, and can be seen in Appendix B.1 and C.1.

- $solve\_Contact\_DisplacementControlled\_Blocked\_Dofs.m$
- $stiff\_mat\_Contact\_GPTS.m$

The contact user input is displayed in the table below. The arch is denoted $part1$ and the flat rigid body as $part2$ in the analysis files. The full overview of the contact parameters are listed in Appendix A.

Table 6.1: Contact input parameters

Parameter name	Value
d_rigid(1).d_step_dir	[0 0 -2]
d_rigid(1).part_nr	2
ms_pairs(1)	{[1 2]}

6.1.1 Mesh refinement

The rigid body mesh consist of only one element. As all degrees of freedom are blocked, the amount of elements in the mesh does not influence the analysis. A penalty parameter of $1e3$ is used for the analysis.

The arch is meshed into first 2 elements, then 4 elements and then 8. Figures are plotted for the displacement after 3 load steps and after the last load step,

load step 5, has executed.

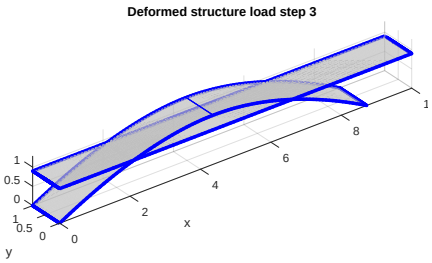


Figure 6.2: 2 elements

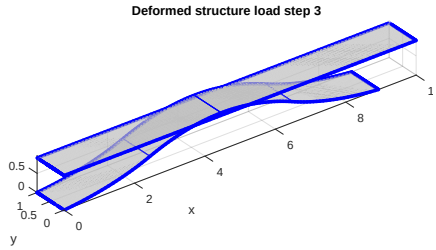


Figure 6.3: 4 elements

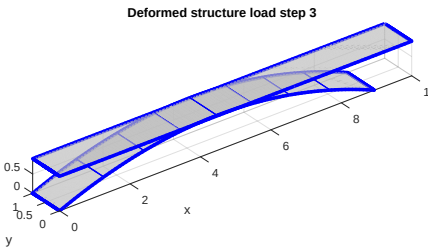


Figure 6.4: 8 elements

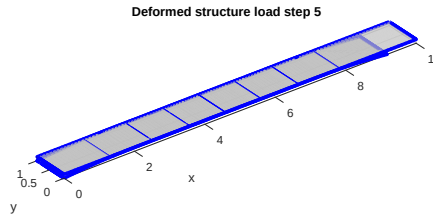


Figure 6.5: 8 elements

When using a mesh of 8 elements for the slave body, the arch is able to follow the displacement of the rigid body. In Figures (6.7) and (6.6) the number of iterations needed per load step is plotted and the load-displacement curve that tracks the displacement of parametric coordinate $[0.5, 0.5]$ of the arch, which is equivalent to its maximal point. That the relationship is linear confirms that the arch follows the displacement of the flat rigid surface, since the rigid body displacement is linear. In the first load step the displacement imposed on the arch from the flat rigid body is slightly less than the other load steps as the starting position of the flat rigid is $0.1mm$ above the arch. The code is accordingly able to give sufficiently accurate results for even a coarse mesh, which can be

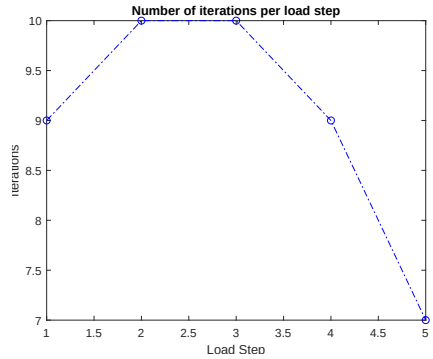


Figure 6.6: 8 elements

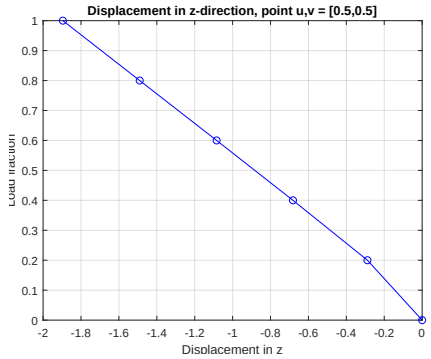


Figure 6.7: 8 elements

expected from a simple numerical problem like the one implemented.

6.1.2 Penalty parameter influence

The influence of changing the penalty parameter is investigated. The penalty parameter typically needs to be chosen carefully. A parameter that is too high results in badly scaled matrices while a too low parameter yields inaccurate results as large penetrations are allowed. A simple example such as this one should though be able to run smoothly for some change in the parameter. The penalty parameter is increased with steps of $1e3$ until the analysis is not able to converge properly anymore. This is achieved for a penalty parameter of $1e8$.

The displacement at load step 3, which is approximately half into the analysis is used as reference to compare the performance of the algorithm for different penalties. The rigid surface is for this step situated at $z = 0.9m$ which is equivalent to a displacement of the arch maximal point of $0.9-2 = -1.1$. This represent the exact solution which the displacement at load step 3 is compared to. The slave body mesh consists of 8 elements.

For this simple example the results in Table (6.2) indicate that the higher the penalty parameter, the more accurate the results. Intuitively the lowest penalty parameters are related to more penetration of the plate which is less physical. The difference between using a penalty parameter of $1e3$ and $1e7$ is though not significant. For a penalty parameter of $1e4$ the difference from the results is less

Table 6.2: Results, flat rigid and arch

Penalty	**Displacement, load step 3**	**Number of iterations, load step 3**	**[%] difference from accurate**
$1e2$	-1.026	7	6.7
$1e3$	-1.086	10	1.3
$1e4$	-1.097	11	0.3
$1e7$	-1.099	15	0.0

than 1%. The higher penalty parameters have a higher demand for iterations in order to achieve a solution. The highest penalty parameter tested has a demand of maximum 34 iterations for the last load step, which is a significant difference from the iterations needed for $1e3$ seen in Figure (6.6). It is thus not recommended to increase the penalty parameter to such numbers when the analysis is to be executed for more complex problems as the solution might not be able to converge.

The objective of the algorithm is to rather have a robust algorithm that can handle variations in the geometries to analyse and variations in the input parameters than an algorithm optimised for accuracy. The GPTS and penalty methods are in general not the most accurate and some minor errors are to be expected. A penalty parameter of $1e3$ or $1e4$ are considered to provide sufficiently accurate results for this analysis, for which the displacement is respectively 1.3% and 0.3% away from the accurate solution, see Table (6.2).

The implemented problem of one rigid patch and an elastic arch indicates that the algorithm is sufficiently robust. It is able to properly run for a very little refined mesh and also for very high penalty parameters. In (De Lorenzis, Scott, Wriggers, Taylor and Zavarize, 2014), the GPTS and penalty algorithm using T-splines was tested for penalty parameters of $1e2$ to $1e5$ for which using $1e3$ had the most accurate results compared to analytic results. A too high penalty parameter lead to an oscillatory behaviour of the solution. It is thus not beneficial to use the highest penalty parameter even if it is closer to accurate contact.

6.1.3 Geometric stiffness K_{geo}

The analysis is run using the same solver function, but changing the contact contribution calculation function to one including the K_{geo} term in the stiffness calculation displayed in Appendix C.2. This is mainly done to verify the accuracy of the implementation as it is significantly more complicated to implement. This algorithm results in the same iterative pattern as without the geometric stiffness contribution due to contact. The results are exactly the same as stated in the Table (6.2). The geometric stiffness includes the linearisation of the variation of the normal gap. It contributes to the convergence rate, but for the simple problem like this it does not show in the results.

6.2 Two elastic arches, edge load

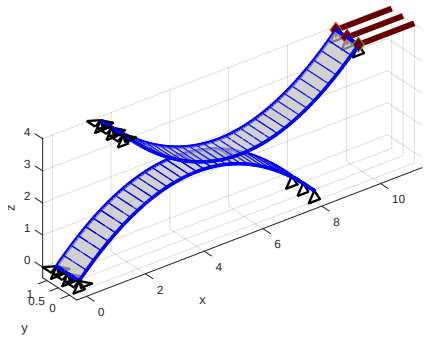


Figure 6.8: Geometry and BCs

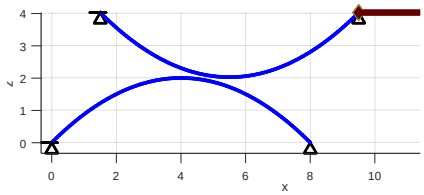


Figure 6.9: Side view of geometry

An example consisting of two arches, one concave and one convex is implemented. The arches are placed just above each other and slightly shifted to the side, see Figure (6.9). The upper arch is applied an edge load on the rightmost edge, which is applied step-wise using Newton-Raphson iterational procedure. During the load steps the upper arch presses down the lower arch. The GPTS and penalty solution procedure is used. Three tests are executed after a mesh refinement is executed. The tests checks the influence of changing the following:

1. Penalty parameter
2. Point search procedure
3. Master/slave status

The arches are simply supported on the right and left edges. The boundary conditions are illustrated in Figure (6.8). 20 load steps are used to impose an edge load of $-10N/m$. Only a small load is applied to stop the displacement before it changes direction which can be a challenge for the Newton-Raphson solution procedure. The lower arch has its highest point at $z = 2m$. A polynomial degree of 2 is used in both parametric direction u and v.

Material parameters:

The same material parameters are applied to both arches: Shell thickness $t = 0.05m$, $E = 1e7N/m^2$ and $\nu = 0.0$.

Analysis specifications:

The solution functions used for this analysis are listed below, and can be seen in Appendix B.2, B.3, C.2 and C.3.

- *solve_Contact_Newton.m*
- *solve_Contact_Newton_2StepSearch.m*
- *stiff_mat_Contact_GPTS_Pen.m*
- *stiff_mat_Contact_GPTS_Pen_2StepSearch.m*

6.2.1 Mesh refinement

The downward displacement of the lower arch, parametric point $u, v = [0.5, 0.5]$ is used as a reference for the mesh refinement. This point correspond to the maximum point of the arch in the initial configuration. The displacement is measured after the last step of the procedure. The results show that there is only 2.4% difference when doubling the mesh size from 32 elements for each body to 64 elements, see Table (6.3). Doubling the mesh also increases the computational effort as stiffness relations needs to be established for more elements. It is thus beneficial to not use a lot more elements than what is sufficient for the

analysis. Using a mesh of 32 elements for each body is here assumed to be sufficient for the tests executed using this geometry. The final deformed shape is plotted in Figure (6.11).

Table 6.3: Relation between displacement and mesh size

Number of elements in mesh	**Displacement in z**	**[%] Difference**
8	-0.20	
16	-0.97	>100
32	-1.23	26.8
64	-1.26	2.4

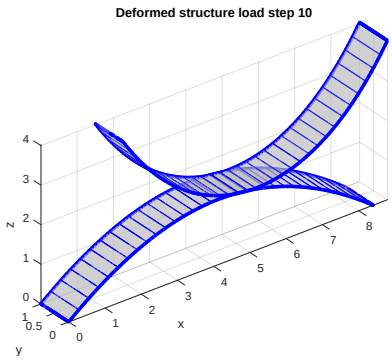


Figure 6.10: Load step 10

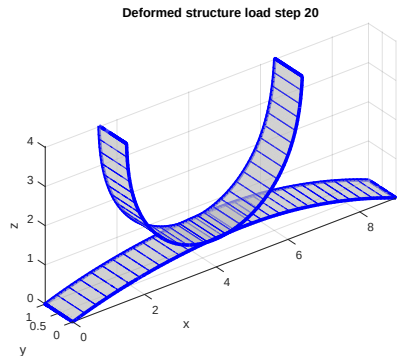


Figure 6.11: Load step 20

Dependence on penalty parameter:

The dependence of the penalty parameter is tested for the final load step, load step 20. Penalty parameters from $1e2$ until the solution can no longer converge are used. The solution is not able to converge for a penalty parameter of $1e5$. Increasing the penalty parameter from $1e3$ to $1e4$, the change in the final displacement in z-direction of point $u, v = [0.5, 0.5]$ is 1.6%. Some change in the results are expected since a large increase in the penalty parameter results in less penetration of the slave into the master body. A penalty parameter of $1e3$ or $1e4$ should be sufficient for the analysis, which was also concluded in the previous example.

Table 6.4: Penalty parameter dependence

Penalty parameter	**Displacement load step 20**	**Number of iterations load step 20**	**[%] Difference in displacement**
$1e2$	-1.18	9	
$1e3$	-1.23	9	4.2
$1e4$	-1.25	9	1.6

6.2.2 Master-slave dependence

Which body is given the master status and which is given the slave status might have an influence on the result. The normal point projection is executed from the slave Gauss point to the master surface. The points that constitute the GPTS contact element will thus not be exactly the same when interchanging the master and slave bodies. Accordingly the contact contribution from the contact element depends on the master-slave definition. An intuitive example of the influence is the normal gap function. The calculated normal gap is not the same when the Gauss point and normal projection is not the same and the results will thus differ. The difference should not be significant for equal, simple geometries. The analysis is for this test run using two different penalty parameters, $1e3$ and $1e4$.

The results are showed in Table (6.5). They indicate that the dependence on which body is chosen as slave and master is smaller for a penalty parameter of $1e3$, though it is not significant for either. This might indicate that $1e3$ is a better choice for the implementation and it is used for the rest of the numerical examples. As the dependence is not significant for either, the algorithm is assumed to be robust enough for change in master-slave status of bodies.

6.3 Dependence on point search procedure

A comparison of the 2 step point search procedure and the procedure only using the already included normal projection point search without adjustments. Up until this point, the numerical examples have been implemented using only the

Table 6.5: Dependence on master-slave status

Penalty parameter	**Master-slave pair**	**Displacement load step 20**	**[%] Difference in displacement**
$1e3$	[1 2]: Lower arch is slave	-1.2375	
	[2 1]: Lower arch is master	-1.2363	<0.1
$1e4$	[1 2]: Lower arch is slave	-1.2545	
	[2 1]: Lower arch is master	-1.2485	0.5

normal projection search. The computational time needed to execute the same analysis is measured. The time measured is for running the complete analysis, from the solver function is called from the main function until it is completed.

The main difference between the two algorithms are summarised here in order to get an understanding of the behaviour.

- The one step search algorithm finds normal projection points for every single Gauss point on the slave surface.
- The two step point search algorithm sorts out the potential contacting elements of the slave and master surface, and only executes the contact stiffness contribution calculation for these elements.
- The one step points search uses the same point as a starting point for each normal projection point search.
- The two step point search has a different starting point for the normal projection point search for each slave element.

The upper arch is given the master status. A penalty parameter of $1e3$ is applied. The number of load steps are 20. The results are shown in Table (6.6). The two step point search is executed for different acceptable contact distances L_{acc}. Only points on the slave surface with a reference distance to the master surface less than L_{acc} enters the contact contribution calculation.

Table 6.6: Point search influence

Point search procedure	**[s] Computational time**	**[%] Difference between one step and two step**
One step	75	
Two step: L_{acc}=1	55	27
Two step: L_{acc}=0.5	52	31
Two step: L_{acc}=0.25	50	33

Each specific problem is run at least three times. If the variation in computational time from one analysis to the other is less than $1e-1$ it is assumed a sufficient reference time for comparison. As the computational time depends on the individual computer capacity and other programs running at the same time, the results only serve as a basis for comparison. The results indicate that the two step point search procedure improves the computational time by approximately 30%, as seen in Table (6.6). The increased computational speed is due to the improved starting points, which results in the normal point projections being found faster, and that the contact contribution is calculated for fewer elements, only those relevant for contact. Decreasing the acceptable distance for contact contribution calculation, L_{acc} from less than 1 does not influence the computational time. The distance needs to be chosen with carefulness. If it is too small, some elements relevant for contact will be left out of the analysis. The L_{acc} parameter needs to be specifically chosen for each geometrically problem as it is defined in the same coordinate system as the geometries.

The two step point search procedure is also tested for updating the contact partner matrix for each iteration instead of for each load step. This procedure takes more time than updating the contacting partners for each load step. It suggests the computing of contacting partners is a time consuming procedure in itself and it is not beneficial to execute it before every contact contribution computation.

6.4 Cylinder squeeze

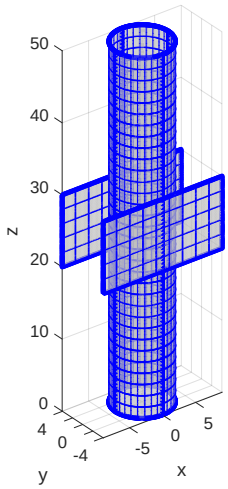


Figure 6.12: Cylinder example, mesh

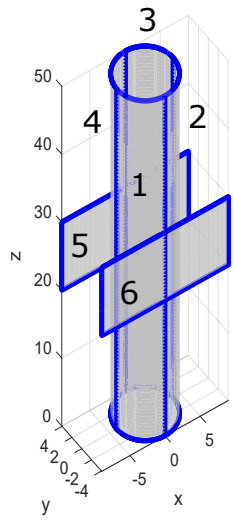


Figure 6.13: Cylinder example, patches

A more complex problem is implemented in order to further test the proposed contact algorithm. The example is taken from (Matzen, 2015) and consists of a cylinder that is *squeezed* by two rigid plates from both sides. The cylinder is subjected to large deformations for which only approximately 10% of the radius is left at the position of the plates on the cylinder. The objective of the example is to test the contact algorithm proposed, including all the implemented features. The problem is analysed by multi-patch contact in two ways. Both by having three bodies involved in the contact and by modelling the cylinder as multiple patches, see Figure (6.13). Moreover, the two step point search algorithm is utilised together with moving two rigid bodies. The example thus serves as a test of how all the features in the code work together. The deformation plots can be roughly compared to the example in (Matzen, 2015), though not accurately as there are no specific results presented in the Phd of Matzen. The deformed shape is plotted and the characteristic shell deformation pattern is looked at.

Material parameters:

The Kirchhoff-Love shell thickness is $t = 0.1\,mm$, Young's modulus $E = 12000\,N/mm^2$,

and $\nu = 0.0$

Geometry modelling:

The diameter of the cylinder is $D = 8mm$ and the length of the cylinder is $L = 50mm$ in the vertical direction. The plate has a width of $w = 10mm$ in the vertical direction. It is given a length of $19mm$. See Figure (6.12). The polynomial degree in both parametric u and v is $p, q = 2$.

The cylinder is modelled by using 4 NURBS patches divided in the longitudinal direction. The patches are connected in the horizontal direction, corresponding to parametric direction v, using penalty symmetry coupling with a penalty parameter of $\alpha = 1e3$. Symmetry coupling is imposed on all cylinder patch edges. This coupling is described in (Herrema et al., 2019). The symmetry coupling functionality is included beforehand in the IGA research code of the marine Department, NTNU. The plates are modelled as one NURBS patch each. In total the analysis consists of six patches that interact. The edges of each patch are visible in the Figure (6.13).

6.5 Analysis specifications

The two rigid plates are moved closer to the cylinder centre for each load step in the Newton-Raphson iterational procedure, pressing it together on the middle. A total displacement of $|y| = 3.61mm$ is imposed on both plates, which is the same displacement as applied in (Matzen, 2015). The displacement is added through 30 load steps. The cylinder is given slave status. All degrees of freedom related to the plates are blocked. The functions used for the analysis solution are listed below and are found in Appendix B.4 and C.4.

- $solve\_Contact\_Cylinder\_Squeeze.m$
- $stiff\_mat\_Contact\_Cylinder\_Squeeze.m$

The solver function and the contact contribution functions are slightly adjusted for the specific problem. The contact contribution calculation of the gap function is adjusted in two ways. Firstly, the shell thickness is accounted for as is done on (Matzen, 2015). Half of the shell thickness is subtracted in order to

avoid the plates penetrating into the cylinder wall, see Eq. (6.1). t^s is the slave shell thickness and t^m is the master shell thickness. As the master surfaces are rigid, t^m is set to zero.

$$g_N^{shell} = g_N - \frac{t^s}{2} - \frac{t^m}{2} \tag{6.1}$$

Additional contact check:

When calculating the normal gap g_N for a Gauss point and its normal projection, an additional check is added. This is due to the possibility of contact registration between cylinder elements outside of the rigid plate region. During testing of the analysis it is discovered normal point projections were found for cylinder elements far away from the plate. This is due to the equation used to check if a point is the normal projection point or not. It only takes into account the tangents at the normal projection point on the surface, and the normal itself. The dot product of Eq. (5.1) can be zero also for points outside of the contact region. As the plate presses into the cylinder, the gap g_N is negative for the unreal contacting points and contact contributions are calculated for them.

In order to come around the unreal contacting points being registered, a maximum tolerated distance in all directions are added. Only if the distance between points $\boldsymbol{x}^s$ and $\boldsymbol{x}^s$ in each direction is less than a tolerance, and if the gap g_N is less than zero, the contact contribution is calculated. The procedure can be seen in text-box 6 in this section. The tolerance in the normal direction is irrelevant as the gap for this direction is correctly calculated. The tolerances set for this example can be seen in Table (6.7) for which tol_z, tol_y, tol_x are the maximum distances in respectively z, y, x direction for the contact contributions to be calculated.

Point search procedure:

For most slave elements outside of the contact region, the normal point projection is not found. This is the correct way for the algorithm to react to these points as they are not within the contact region. The normal projection point search algorithm reaches its maximum number of iterations for these slave elements which results in it returning an arbitrary point at the master surface as the normal projection point. It is a procedure that works well since it is impossible

Box 6. *Additional contact checks*

Check for contact:

- Calculate $g_N = (\boldsymbol{x}^s - \boldsymbol{x}^m) \cdot \boldsymbol{n} - \frac{t^s}{2}$
- If $g_N < 0$ and
- if $|(z^s - z^m)| < tol_z$ and
- if $|(y^s - y^m)| < tol_z$ and
- if $|(x^s - x^m)| < tol_z$
- → calculate contact contribution for the Gauss point.

for the arbitrarily located normal projection to result in $g_N < 0$ and thus these points do not enter the contact contribution calculation. On the other hand it is not ideal to allow the point projection algorithm to go through many iterations unnecessary. It is time consuming and increases the computational effort, especially when most of the slave elements are not within a contact region. The two step point search procedure can limit the number of slave elements considered for the analysis which again reduces the number of unnecessary point projection iterations. It does though not completely leave out irrelevant slave elements since the acceptable contact distance, L_{acc} needs to allow some extra elements to enter the analysis in order to be sure all relevant elements are included in the load step. A check specifically related to the normal projection point search can be implemented in the future to further reduce time consumption.

Table 6.7: Input parameters, cylinder squeeze numerical example

Contact input parameter	**Value**
alf	$1e3$
tol_z	0.1
tol_y	100
tol_x	0.1
L_{acc}	5
ms_pair(1)	{[3 5]}
ms_pair(2)	{[4 5]}
ms_pair(3)	{[1 6]}
ms_pair(4)	{[2 6]}

Master-slave pair definition:

Multiple master-slave pairs are defined to couple the patches that are relevant for contact. The patch numbering is illustrated in Figure (6.13). The master-slave pairs seen in Table (6.7) defines which patch numbers are to enter the contact contribution calculation and which of them is the slave and the master in the formulation. The first entry in the master-slave pair vector is the patch number of the patch to be given slave status and the second entry the patch o be given master status.

6.6 Deformation of the cylinder

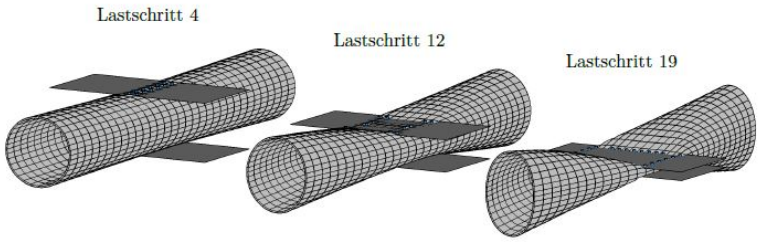


Figure 6.14: Cylinder and rigid plates deformation (Matzen, 2015)

The deformed shape for load step 3, 9, 24 and 30 are plotted in Figures (6.15) to (6.22). The deformation shape achieved in (Matzen, 2015) is illustrated in Figure (6.14). The deformed shape resulting from the proposed algorithm is comparable to what is seen in Figure (6.14). During the first load steps there is an ovalisation in the direction perpendicular to the plate deformation, here direction x. This can be seen i Figures (6.15) to (6.18). In later load steps the the ovalisation changes direction. In Figure (6.19) to (6.22 there is an obvious ovalisation in the direction of contact, along the y-axis. For Patch 1 and 2, the ovalisation is in negative y-direction. For Patch 3 and 4 the ovalisation is in positive y-direction. The stiffness is originally low and increases after some deformation has occured. This can be explained by ring-stress that is activated which results in an increased stiffness in the structure, as is also described in (Matzen,

2015). This is a typical shell-deformation pattern and the figures thus confirm a physically-acting contact algorithm.

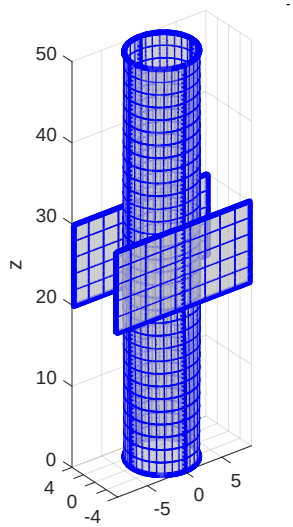


Figure 6.15: Load step 3

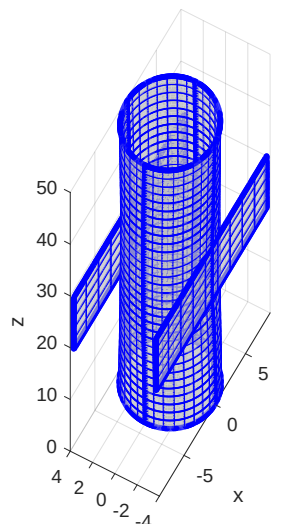


Figure 6.16: Load step 3

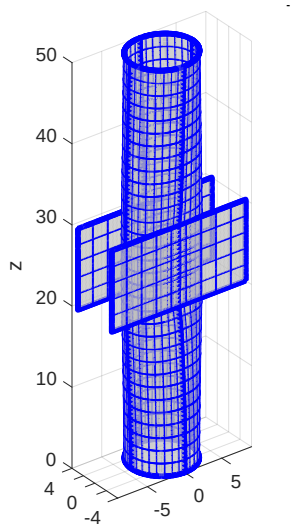


Figure 6.17: Load step 9

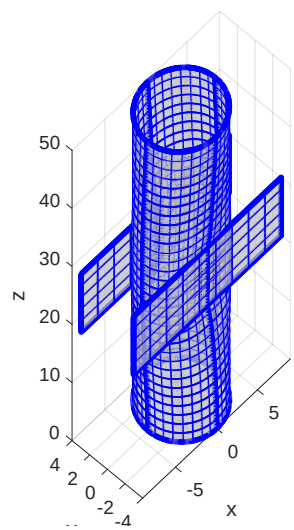


Figure 6.18: Load step 9

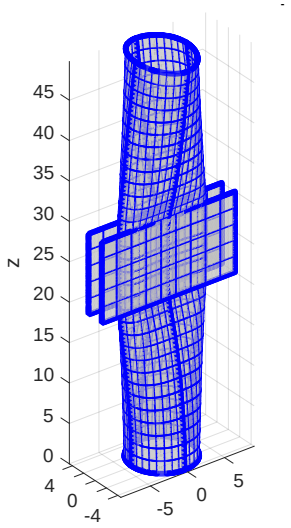


Figure 6.19: Load step 24

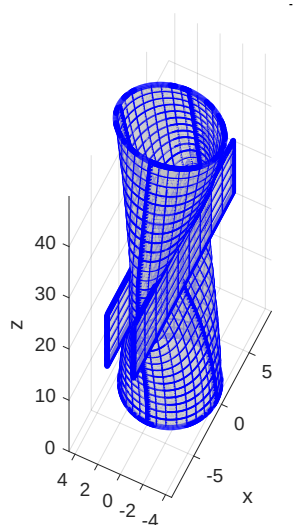


Figure 6.20: Load step 24

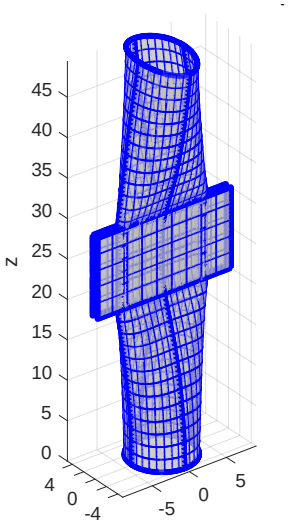


Figure 6.21: Load step 30

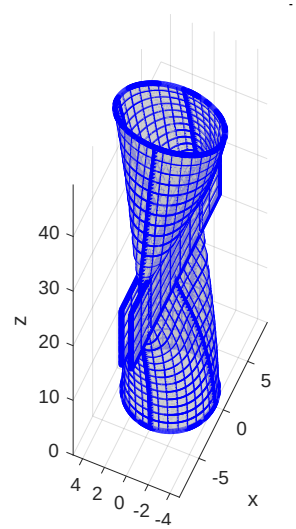


Figure 6.22: Load step 30

The deformation in the y-direction is plotted for points situated on Patch 1 in Figure (6.23). The points are uniformly distributed on the plane $z = 25$. The first point is situated at the centre of the plate corresponding to $(u, v) = (0.5, 1)$, and the rest are distributed going in the clockwise direction along the circle peripheral until $(u, v) = (0.5, 0.4)$.

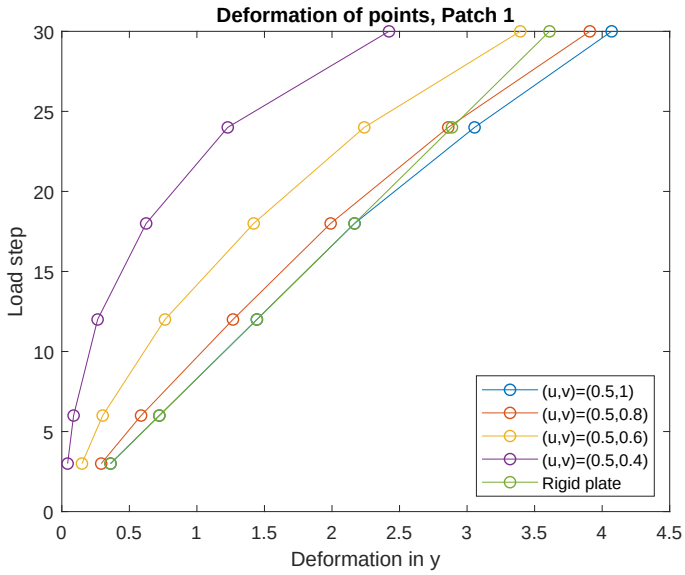


Figure 6.23: Cylinder example, deformation of points along z=25

The total displacement of the plate is $3.61mm$. The deformation of points plot, Figure (6.23), show that the cylinder positioned at the middle of the plate deforms more than the plate after load step 18. This indicates that the cylinder wall at this position changes from convex to concave. This shape is expected from a cylindrical shape which is also described in (Matzen, 2015). A nonphysical behaviour is though illustrated in the plot. Some points deform even more than the cylinder radius which indicates self-contact has occured. As the implementation i not defined to search for contact between two cylinder patches, the self contact is not detected.

In Figure (6.24) a point situated at $(u,v) = (1,1)$ is plotted. That corresponds to $(x,y,z) = (0,-4,50)$ in Cartesian coordinates. This figure further illustrates the ovalisation in the different directions. It can be seen that the deformation changes direction from a displacement in positive y-direction to negative after load step 7.

An issue is discovered while running the analysis. The algorithm is not able to converge to a solution for one of the final load steps, load step 25. The algorithm continues to the next load step and the solution is again able to converge for the

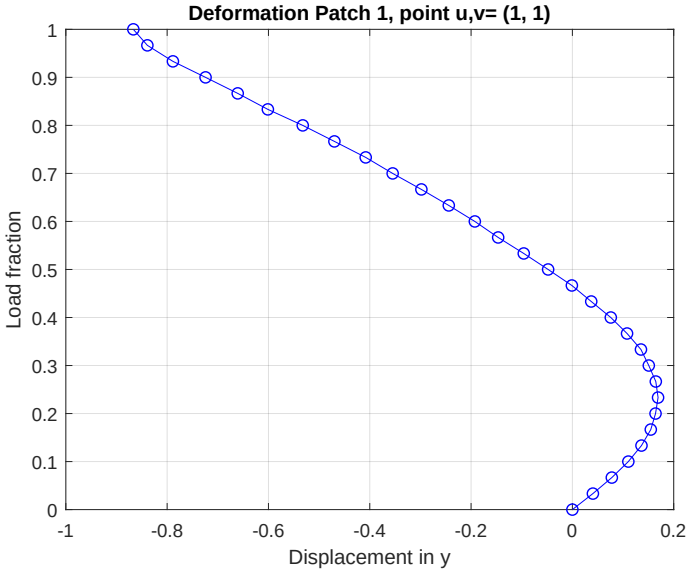


Figure 6.24: Cylinder example, deformation of Patch 1, (u,v)=(1,1)

rest of the load steps. This behaviour is seen in Figure (6.25). For cylindrical shells the deformation pattern is complicated and the Newton-Raphson procedure might not be able to converge if the geometry snaps through. At load step 25 the change to concave has occured. An arc-length solver should be implemented in order to handle highly geometrically nonlinear problems.

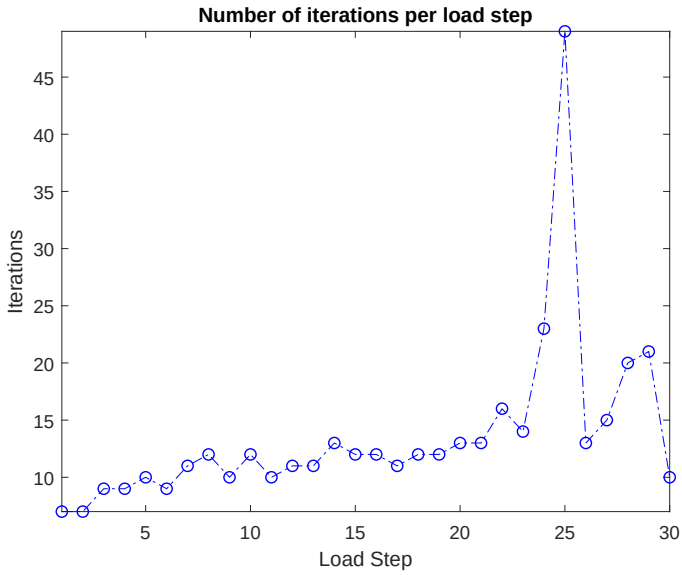


Figure 6.25: Cylinder example, iterations per load step

Concluding remarks for the cylinder example:

The cylinder example is quite complex. It includes both multi-patch penalty coupling and multi-patch contact. The multi-patch symmetry coupling introduces two more penalties as the patch boundaries are coupled both in the rotational and transnational degree of freedom, see (Herrema et al., 2019). There are consequently in total 3 different penalty parameters involved in the analysis. As errors and convergence issues are related to penalty parameters, the demand for a robust algorithm is accordingly increased. That the proposed contact algorithm is able to solve the system thus is an indication of its robustness.

The displacement pattern looks physically feasible. The drawback is the indicated penetration of the cylinder wall into itself which is not taken into consideration in the implemented algorithm.

Chapter 7

Conclusions and further work

A frictionless contact algorithm based on a combination between Gauss-point-to-segment, GPTS contact discretization and the penalty method to impose the contact constraints has been proposed. The code includes a two step point search procedure and possibilities to impose contact between multiple bodies or NURBS patches. Contact can be introduced by adding external forces or displacement of rigid bodies.

The GPTS formulation is simple to implement and has together with the penalty method previously showed a sufficient accuracy and robustness for engineering problems, in (De Lorenzis, Scott, Wriggers, Taylor and Zavarize, 2014), though it was related to convergence issues for high penalty parameters. It has in this thesis been showed that the proposed algorithm using these methods is able to execute analysis on simple as well as complex problems. A study of convergence, dependence on the penalty parameter and dependence of master-slave choice was conducted. For the simplest problem including one rigid and one elastic body, the algorithm was able to provide satisfactory accurate results and robustness for even a coarse mesh of the slave body. It was able to converge for even a large penalty parameter of $1e7$. Moreover, the results were deemed sufficiently accurate for a penalty parameter of $1e3$. For contact between two elastic bodies including sliding, the convergence was more dependent on the penalty parameter. The solution was unable to converge for high penalty parameters. It was on the other hand shown that a penalty parameter of $1e3$ or

$1e4$ provided sufficiently accurate results. The example showed little to no dependence on which body is given slave status and which is given master status for both penalty parameters $1e3$ and $1e4$.

A study of the influence of the proposed two step point search algorithm was also conducted. It improved the analysis speed by approximately 30% for an example of two elastic bodies. Contact problems are complicated and often related to a high demand for computational resources. It is thus relevant to find methods to improve the efficiency such as the proposed search algorithm.

A complex problem consisting of two rigid plates pressing together a cylinder was implemented. For the implementation both multi-patch contact, moving rigid bodies and the two step point search procedure are included. The complexity of the problem increases the demand for robustness of the algorithm. Furthermore there are three separate penalty parameters involved in the analysis. Two from symmetry coupling of the patches that the cylinder geometry is modelled by and one due to the contact formulation. There is consequently a very high demand of robustness for the algorithm. The code was able to converge and provide physically viable results compared to the same numerical problem in (Matzen, 2015). Conclusively the algorithm can be regarded a sufficient first implementation of contact, though the accuracy of the code should be further investigated.

Further work

The proposed contact algorithm has not been applied to a contact problem for which the accurate results are known in advance. In order to prove its sufficiency it is necessary to compare it more accurately to earlier work. The first recommendation for further work is thus to test the algorithm for some benchmark examples.

Some improvements might impact the convergence and robustness of the code. Using Gauss points can lead to an over-constrained nature of the problem which can be avoided by using Greville or Botella points as contact collocation points instead. In the future it is feasible to investigate the performance of the code with these points instead of the Gauss points. Moreover, a linearisation of the displacement is not executed in this implementation. It might interfere with the convergence rate and can be later be included. An arc-length solver should be

applied in order to improve the convergence of problems related to complicated deformations.

In order to make the code more realistic, contact in the tangential direction should be added. Frictional forces are for this implementation completely disregarded. The accuracy can be improved by introducing more sophisticated contact discretisation schemes, such as Mortar methods in combination with Lagrange multiplier method for imposing the contact constraints. These methods are on the other hand more demanding in regards to computational effort and implementation.

Bibliography

Breitenberger, M. (2016), CAD-integrated design and analysis of shell structures, PhD thesis, Technische Universität München.

Carnes, K. (2005), 'The ten greatest events in tribology history', *Tribology and Lubrication Technology* **61**(6), 36.47.

De Lorenzis, L Dimitri, R., Scott, M. A., Wriggers, P., Taylor, R. and Zavarize, G. (2014), 'Isogeometric large deformation frictionless contact using t-spline', *Computer Methods in Applied Mechanics and Engineering* **269**, 394–414. Web.

De Lorenzis, L., Hughes, T. J. R. and Wriggers, P. (2014), 'Isogeometric contact: a review', *GAMM-Mitteilungen* **37**(1), 85–123.

De Lorenzis, L., Temizer, L., Wriggers, P. and Zavarise, G. (2011), 'A large deformation frictional contact formulation using nurbs-based isogeometric analysis', *International Journal for Numerical Methods in Engineering* **87**(13), 1278–1300.

De Lorenzis, L., Wriggers, P. and Weissenfels, C. (2017), 'Computational contact mechanics with the finite element method', *Encoclypedia of Computational Mechanics Second Edition* **2: Solids and Structures**.

Fischer, K. A. (2005), Mortar type methods applied to nonlinear contact mechanics, PhD thesis, Universität Hannover.

Fischer, K. and Wriggers, A. (2005), 'Frictionless 2d contact formulations for fi-

nite deformations based on the mortar method', *International Journal for Numerical Methods in Engineering* **36**(3), 226–244.

Frankie, D., Düster, A., Nübel, V. and Rank, E. (2010), 'A comparison of the h-, p-, hp-, and rp-version of the fem for the solution of the 2d hertzian contact problem', *Computational Mechanics* **45**(5), 513–522.

Grossmann, D., Jüttler, B., Schlusnus, H., Barner, J. and Vuong, A. (2012), 'Isogeometric simulation of turbine blades for aircraft engines', *Computer Aided Geometric Design* **29**(7), 519–531.

Herrema, A. J., Johnson, E. L., Proserpio, D., Wu, M. C., Kiendl, J. and Hsu, M.-C. (2019), 'Penalty coupling of non-matching isogeometric kirchhoff–love shell patches with application to composite wind turbine blades', *Computer Methods in Applied Mechanics and Engineering* **346**, 810–840.

Hughes, T., Cottrell, J. and Bazilevs, Y. (2005), 'Isogeometric analysis: Cad, finite elements, nurbs, exact geometry and mesh refinement', *Computer methods in applied mechanics and engineering* **194**, 4135–4195.

Hughes, T.J.R, T. I. W. P. (2011), 'Contact treatment in isogeometric analysis with nurbs', *Computer Methods in Applied Mechanics and Engineering* **200**(9), 1100–1112.

Kiendl, J. M. (2011), Isogeometric Analysis and Shape Optimal Design of Shell Structures, PhD thesis, Technische Universität München.

Luenberger, D. and Ye, Y. (2016), *Linear and Nonlinear Programming*, Vol. 228, 4 edn, Springer, Cham.

Matzen, Cichosz and Bischoff (2013), 'A point to segment contact formulation for isogeometric, nurbs based finite elements', *Computer Methods in Applied Mechanics and Engineering* **255**, 27–39.

Matzen, M. E. (2015), Isogeometrische Modellierung und Diskretisierung von Kontaktproblemen; Isogeometric modeling and discretization of contact problems, PhD thesis, Universität Stuttgart.

Parsons, B. and Wilson, E. A. (1970), 'Finite element analysis of elastic contact problems using differential displacements', *International journal for numerical methods in engineering* **2**, 387–395.

Rogers, D. (2001), *An Introduction to NURBS: With Historical Perspective*, Elsevier Science.

Wriggers, P. (2006), *Computational contact mechanics*, Vol. 498, 2nd edn, Springer Verlag, Berlin Heidelberg, Germany.

Wriggers, P., Nackenhorst, U. and Symposium, C. M. I. (2006), *Analysis and Simulation of Contact Problems*, Springer, Berlin, Heidelberg.

Zavarise, G. and De Lorenzis, L. (2009*a*), 'A modified node-to-segment algorithm passing the contact patch test', *International Journal for Numerical Methods in Engineering* **79**(4), 379–416.

Zavarise, G. and De Lorenzis, L. (2009*b*), 'The node-to-segment algorithm for 2d frictionless contact: Classical formulation and special cases', *Computer Methods in Applied Mechanics and Engineering* **198**(41), 3428–3451.

Appendices

Appendix A

Parameter description

This appendix section describes the parameters defined for contact analysis defined in accordingly the main functions, solver functions and contact contribution functions. The Main functions are not added in the appendix as the setup is taken from other codes already a part of the IGA research code. The parameters defined in main are passed on to the solver functions.

****Parameters defined in the main function****

All of these parameters are stored in the struct: *anls.contact.XX*.

alf: Penalty parameter for the contact Penalty method.

tol_z/y/x: Acceptable distance in xyz-dir for contact consideration.

potential_contact_d: Maximum distance between master and slave points for them to be considered as potential contact partners for the 2StepSearch. Denoted as L_{acc} in the text.

search_par{**1**}: For the one step point search: Start point for normal projection algorithm.

search_par{**2**}: Step size for the normal projection algorithm.

search_par{**3**}: Tolerance on perpendicularity for the normal projection point search.

***search_par*{4}**: Tolerance on distance between points for the normal projection point search.

***ms_pairs*(1...*n*)**: [slave patch nr, master patch nr] = bodies to be considered for contact between each other. N = number of master-slave pairs.

d_rigid*(1).*d_step_dir: Rigid body to move, total distance in each direction [x y z] to move all control points.

d_rigid*(1).*part_nr: The number of the patch to be moved.

*****Parameters defined in the solver functions*****

u_active: Displacements of active degrees of freedom.

potential_contact_d: See MAIN functions

u_cm, v_cm: Parametric coordinate of centre point of a master element.

u_cs, v_cs: Parametric coordinate of centre point of a slave element.

d_center_elem: Normal distance between a point u_cm, v_cm and u_cs, v_cs.

d_center_elem_min: The smallest out of all ***d_center_elem*** for a slave element.

M_min_dist_sm: Matrix storing for each row = all slave elements and 4 cols = info about the closest master element: $[u\_cm, v\_cm, iel\_m, normal\_d\_center\_elem]$, for which iel_m is the element number of the closest master element.

M_contact_partners: Same structure as ***M_min_dist_sm***, except: all entries = 0 for a row(slave elem) that is not considered for contact, which is the case if ($d\_center\_elem\_min > potential\_contact\_d$).

*****Parameters defined in the contact contribution functions*****

***part*1, *part*2**: Slave patch, master patch.

el: Current slave element in the slave element loop.

***el*2**: Master element containing the normal projection point.

xyz_CP: Coordinates of CPs of an element: .._1 for slave, .._2 on master.

BF1, BF2: Shape functions and deriv at a point in element el, el2

R1, R2: Shape functions at a point in element el, el2.

ke_c: Main contact element stiffness contribution to global matrix.

k_geo: Geometric stiffness contact contribution to the element stiffness matrix.

ke_gp: Total contact stiffness for the element including k_geo.

fie_c: Element internal forces contribution (residual force contribution) to global internal force and residual force vector due to contact.

Appendix B

Solver functions

B.1 Solver function: Move rigid body

```
%-------------------CYLINDER SQUEEZE SOLVER--------------
% ****DESCRIPTION****
% Solves nonlinear problem by Newton-Raphson iterations with moving rigid
% bodies for every load step. Single point search procedure.

% Created by Embla L. Holten , 2019.
% General setup from Davide Proserpios codes


%****STIFF MAT FUNCTION CALLED****
%   stiff_mat_Contact_GPTS_Pen.m           Inclides 1 Step Search and kgeo
%OR stiff_mat_Contact_GPTS_Pen_Simple.m    Excludes kgeo

%-----------------------------------------------------
function [solution, anls] = ...
  solve_Contact_Newton_DisplacementControlled_Blocked_Dofs(anls, resolution)

% -----------------------------------------------------
% SOLVER INPUT AND INITIALISATION
```

```
20
21 %***DEFINE PARAMETERS FOR NONLINEAR SOLVING***
22 n_step = anls.par_solv.nstep;   % number of load steps
23 n_iter = anls.par_solv.niter;   % max number of iterations for a
       load step
24 tol    = anls.par_solv.tol;     % tolerance on residual for exit
       iterations
25 ndof = anls.ndof;
26
27 %***DEFINE PARAMETERS FOR PLOTTING***
28 patch_plot = anls.par_plot.patch;    % patch to plot
29 plot_coord = anls.par_plot.coord;    % point to plot the
       displacement
30 plot_dir   = anls.par_plot.dir;      % direction of displacement
       to plot
31 plot_sign  = anls.par_plot.sign;     % sign of the displacement
32
33 iters = zeros(n_step,1);
34
35 %***INITIALIZE DISPL***
36 u_active = zeros(ndof,1);
37 u_active_step = zeros(ndof,n_step+1); % initialize vector
       collecting displ
38
39 %***INITIALIZE PLOTTING OF LOAD-DISPL***
40 lambda_plot = zeros(n_step+1,1);
41 u_plot = zeros(n_step+1,1);
42
43 figure()
44 xlabel('displacement')
45 ylabel('load fraction')
46 title(['point par. coord. (', ...
47     num2str(plot_coord), '), displacement dir=', num2str(plot_dir)
           ])
48 grid on
49 h = animatedline('Marker','o');
50 addpoints(h,u_plot,lambda_plot);
51 drawnow
52
53
54 % -------------------------------------------------------
55 % DEFINE AND INITIALISE CONTACT PARAMETERS
56
```

```
57 % Preallocate vector of contact normal signs:
58 n_sign_ms_pairs = zeros(length(anls.contact.ms_pairs),1);
59
60 % Loop master-slave pairs:
61 for i = 1:length(anls.contact.ms_pairs)
62     % Calculate contact normal sign at the master surface:
63     us = 0.5; vs = 0.5; %Choose an arbitrary point us and vs on
           slave
64     [XYZ_s] = get_point_coord([us, vs],...              %Get
           cartesian coor
65         anls.parts(anls.contact.ms_pairs{i}(1)).patch); %of point
               (us,vs)
66
67     % Get normal projection point onto master surface from [XYZ_s]
68     search_par = anls.contact.search_par; %Retrieve analysis data
69     search_start = search_par{1}; %Start of normal proj point
           search
70
71     [um,vm,~] = ...
72         point_project_surf(anls.parts(anls.contact.ms_pairs{i}(2))
               .patch,...
73         search_start,XYZ_s,search_par); % Normal projection
               algorithm
74
75     iel = get_point_element([um, vm], ...
76         anls.parts(anls.contact.ms_pairs{i}(2))); %Fin element
               number of
77                                                   %um,vm
78
79     % Retrieve the element of the normal proj point um,vm:
80     elm = anls.parts(anls.contact.ms_pairs{i}(2)).els(iel);
81     ncp_em = anls.parts(anls.contact.ms_pairs{i}(2)).els.ncp_e;
82
83     %Calculate current xyz coordinate of control points:
84     xyz_CPm = reshape(elm.CP(:,:,1:3),[ncp_em,3]);
85
86     %Compute base functions to get drduv:
87     [BFm] = compute_BF_elementbased([um,vm],elm,1);
88     dRduvm = BFm(2:3,:)';
89
90     %Get base functions:
91     [~,g3,lg3] = get_base_func(dRduvm,xyz_CPm);
92
```

```
93      %Calculate normal vector n (normalized):
94      n_shell = [0; 0; 0];
95      n_shell(1) = g3(1)/lg3;
96      n_shell(2) = g3(2)/lg3;
97      n_shell(3) = g3(3)/lg3; %normalization of g3 by components
98
99
100     % Calculate normal gap and check sign:
101     [XYZ_m] = get_point_coord([um,vm],...
102         anls.parts(anls.contact.ms_pairs{i}(2)).patch);
103     gap = (XYZ_s-XYZ_m)*n_shell;
104     if gap < 0
105         n_sign = -1;
106     end
107
108     if gap > 0
109         n_sign = 1;
110     end
111     n_sign_ms_pairs(i) = n_sign;
112 end % Loop through master-slave pairs
113
114
115 % -------------------------------------------------------
116 % LOAD STEPS / NEWTON-LOOP
117
118 for il = 1:n_step   % starting step(il step counter)
119
120     lambda_step = il/n_step;
121     lambda_plot(il+1) = lambda_step; %for plotting
122
123     disp('**************')
124     disp([' Step: ', num2str(il)]);
125
126 %-------------- Move rigid bodies:
127     % Add displacement to CPs of all rigid patches to move:
128     for i = 1:length(anls.contact.d_rigid) % Loop rigid patches
129         %Retrieve control points of rigid patch:
130         CP_p = anls.parts( anls.contact.d_rigid(i).part_nr ).
                patch_def.CP;
131         for iv = 1:length(CP_p(1,:,1)) %Loop CPs in parametric v-
                direction
132             for iu = 1:length(CP_p(:,1,1)) %Loop CPs in param. u-
                    dir.
```

```
133                 % Add prescribed displacement:
134                 CP_p(iu,iv,3) = CP_p(iu,iv,3) + ...
135                     anls.contact.d_rigid(i).d_step_dir(3)/n_step;
                            %z-dir
136                 CP_p(iu,iv,2) = CP_p(iu,iv,2) + ...
137                     anls.contact.d_rigid(i).d_step_dir(2)/n_step;
                            %y-dir
138                 CP_p(iu,iv,1) = CP_p(iu,iv,1) + ...
139                     anls.contact.d_rigid(i).d_step_dir(1)/n_step;
                            %x-dir
140             end
141         end
142         anls.parts( anls.contact.d_rigid(i).part_nr ).patch_def.CP
                = CP_p;
143         anls.parts( anls.contact.d_rigid(i).part_nr ).patch.CP =
                CP_p;
144
145         % dd displacement to CPs of all rigid elements to move:
146         for iel = 1:...
147                 length(anls.parts( anls.contact.d_rigid(i).part_nr
                        ).els)
148             CP_elem = ...
149                 anls.parts( anls.contact.d_rigid(i).part_nr ).els(
                        iel).CP;
150             for iv = 1:length(CP_elem(1,:,1))
151                 for iu = 1:length(CP_elem(:,1,1))
152                         CP_elem(iu,iv,3) = CP_elem(iu,iv,3) + ...
153                             anls.contact.d_rigid(i).d_step_dir(3)/
                                    n_step;
154                         CP_elem(iu,iv,2) = CP_elem(iu,iv,2) + ...
155                             anls.contact.d_rigid(i).d_step_dir(2)/
                                    n_step;
156                         CP_elem(iu,iv,1) = CP_elem(iu,iv,1) + ...
157                             anls.contact.d_rigid(i).d_step_dir(1)/
                                    n_step;
158                 end
159             end
160             anls.parts( anls.contact.d_rigid(i).part_nr ).els(iel)
                    .CP = ...
161                 CP_elem;
162         end
163     end % Loop all rigid parts
164
```

```
165 %-------------- Iterations loop :
166     for it = 1:n_iter % (it is counter of iterations for the step)
167         if (it == (n_iter-1) ) %check if convergence occurred or
                max iterss
168             disp('WARNING: Reached max iterations for the load
                    step.')
169             break;    %exit iteration loop -> next load step
170         end
171
172         add_stiff_var.u_active_prev = u_active;  % update
                displacement vec
173
174         if anls.NL_geo == 0 % check that nonlinear analysis is
                specified,
175                             % error if linear
176             disp('***ERROR calculating stiffness matrices:
                    Analysis is ')
177             disp('specified as linear in anls.NL, have to be
                    nonlin!***')
178         end
179
180         [stiffness] = stiff_mat(anls,add_stiff_var);
181         [stiffness] = stiff_mat_coupling(anls,stiffness,
                add_stiff_var);
182
183         % Calculate contact contributions:
184        for i = 1:length(anls.contact.ms_pairs) %Loop master-slave
               pairs
185             %Pass on current master-slave pair and contact normal
                    sign
186             add_stiff_var_contact.n_sign =  n_sign_ms_pairs(i);
187             add_stiff_var_contact.ms_pairs = anls.contact.ms_pairs
                    {i};
188
189             % Calculate contact stiffness contribution:
190             %[stiffness, anls] = stiff_mat_Contact_GPTS_Pen...
191             %    (anls,stiffness,add_stiff_var,
                    add_stiff_var_contact);
192
193             [stiffness, anls] = stiff_mat_Contact_GPTS_Pen_Simple
                    ...
194                 (anls,stiffness,add_stiff_var,
                        add_stiff_var_contact);
```

```
195         end
196
197         % Retrieve stiffness and force residual:
198         K = stiffness.K;
199         Fi_active = stiffness.Fi_active;
200         Fr = Fi_active;   %No external load, residual = internal
                forces
201
202         % Calculate residual for convergence:
203         residual = sqrt(Fr'*Fr);     %calculate residual for conv.
                check
204
205         disp(['     iteration: ', num2str(it), ', res: ', num2str(
                residual)]);
206
207         % Check convergence:
208         if (residual < tol)
209             break;
210         end
211         disp(['Residual: ', num2str(residual)])
212
213         % Solve system linearly and update displacements:
214         du_it_active = -K\Fr;
215
216         u_active = u_active+du_it_active; % update displ of active
                dof
217         % Compute displaced CPS of the patches:
218         [anls] = compute_deformed_CP(anls,u_active);
219     end % Loop iterations
220
221     iters(il) = it; %store number of iterations for the load step
222     solution.iters = iters;
223
224     u_active_step(:,il+1) = u_active; % vector collecting displ at
            each step
225
226     % Update global displacement vector u:
227     u = cell(length(anls.parts),1); % initialize
228     for ip = 1:length(anls.parts) %loop over patches
229
230         ncp_p = anls.parts(ip).patch.ncp; % number of CP for the
                patch
231         ndof_cp = anls.parts(ip).ndof_cp;
```

```
232
233         u{ip} = zeros(ncp_p*ndof_cp,1); % initialize
234         for icp = 1:ncp_p %loop over CP of the patch
235             for dir = 1:ndof_cp % loop over dof for each cp
236                 %If the dof is free, update:
237                 if ( anls.parts(ip).connectivity.ID(dir,icp)~=0 )
                        % if the dof is free
238                     u{ip}( ndof_cp*(icp-1)+dir ) = u_active( anls.
                            parts(ip).connectivity.ID(dir,icp) ); %u(a
                             control point) = u_active
239                 end
240             end
241
242         end %loop over CP of the patch
243     end % loop over patches
244
245
246     [displ] = get_point_displ(plot_coord,anls.parts(patch_plot),u{
            patch_plot});
247     u_plot(il+1) = plot_sign*displ(plot_dir);
248
249     %live plotting of load-displ
250     addpoints(h,u_plot(il+1),lambda_plot(il+1));
251     drawnow
252
253     % Plot current configuration:
254     if rem(il,1)==0 %Plot every 5 load step
255         solution.d = u;
256         solution.d_active = u_active;
257
258         plot_factor = 1;        %magnification factor
259         figure
260         plot_structure_deformed(anls,solution,plot_factor,
                resolution,'mesh')   %'mesh','num_elem','basis','
                num_basis','thickness'
261         xlabel('x')
262         ylabel('y')
263         zlabel('z')
264         title(['Deformed structure load step ', num2str(il), ''])
265     end
266
267 end % Loop load steps: Newton-Raphson solution loop
268
```

```
lambda_plot(il+2:end) =[];
u_plot(il+2:end) =[];

% -------------------------------------------------------
% SOLUTION, OUTPUT

solution.d = u;
solution.d_active = u_active;

u_active_step(:,il+2:end) = [];
solution.d_active_step = u_active_step; % vector collecting displ
    at each step


solution.Fi_active = Fi_active;

% output nonlinear load-displ curve
solution.u_plot = u_plot;
solution.lambda_plot = lambda_plot;

end %function
```

B.2 Solver function: External load

```
%-------------------SIMPLE CONTACT SOLVER-------------
% ****DESCRIPTION****
% Solves nonlinear problem by Newton-Raphson iterations. Single
    point step
% procedure.

% Created by Embla L. Holten , 2019.
% General setup from Davide Proserpios codes


%****CONTACT STIFF MAT FUNCTION CALLED****
%   stiff_mat_Contact_GPTS_Pen.m    Includes geometric stiffness
    kgeo
%OR stiff_mat_Contact_GPTS_Pen_Simple.m  Excludes geometric
    stiffness kgeo

%-------------------------------------------------------
```

```
15
16 function [solution, anls] = solve_Contact_Newton(anls, resolution)
17
18 % -----------------------------------------------------
19 % SOLVER INPUT AND INITIALISATION
20
21 %***DEFINE PARAMETERS FOR NONLINEAR SOLVING***
22 n_step = anls.par_solv.nstep;   % number of load steps
23 n_iter = anls.par_solv.niter;   % max number of iterations for a
       load step
24 tol    = anls.par_solv.tol;     % tolerance on residual for exit
       iterations
25 ndof = anls.ndof;
26 it = 0;                         %Initialize iterations for storing
27
28 %***DEFINE PARAMETERS FOR PLOTTING***
29 patch_plot = anls.par_plot.patch;    % patch to plot
30 plot_coord = anls.par_plot.coord;    % point to plot the
       displacement
31 plot_dir   = anls.par_plot.dir;      % direction of displacement
       to plot
32 plot_sign  = anls.par_plot.sign;     % sign of the displacement
33 iters = zeros(n_step,1);
34
35 F = anls.F; %Extract force vector
36
37 %***INITIALIZE DISPL***
38 u_active = zeros(ndof,1);
39 u_active_step = zeros(ndof,n_step+1); % initialize vector
       collecting displ at each step
40
41 %***INITIALIZE LIVE PLOTTING OF LOAD-DISPL***
42 lambda_plot = zeros(n_step+1,1);
43 u_plot = zeros(n_step+1,1);
44
45 figure()
46 xlabel('displacement')
47 ylabel('load fraction')
48 title(['point par. coord. (', num2str(plot_coord), '),
       displacement dir=', num2str(plot_dir)])
49 grid on
50 h = animatedline('Marker','o');
51 addpoints(h,u_plot,lambda_plot);
```

```
52 drawnow
53
54 % ------------------------------------------------------
55 % DEFINE AND INITIALISE CONTACT PARAMETERS
56 % Preallocate vector of contact normal signs:
57 n_sign_ms_pairs = zeros(length(anls.contact.ms_pairs),1);
58
59 % Loop master-slave pairs:
60 for i = 1:length(anls.contact.ms_pairs)
61     % Calculate contact normal sign at the master surface:
62     us = 0.5; vs = 0.5; %Choose an arbitrary point us and vs on
           slave
63     [XYZ_s] = get_point_coord([us, vs],...              %Get
           cartesian coor
64         anls.parts(anls.contact.ms_pairs{i}(1)).patch); %of point
               (us,vs)
65
66     % Get normal projection point onto master surface from [XYZ_s]
67     search_par = anls.contact.search_par; %Retrieve analysis data
68     search_start = search_par{1}; %Start of normal proj point
           search
69
70     [um,vm,~] = ...
71         point_project_surf(anls.parts(anls.contact.ms_pairs{i}(2))
               .patch,...
72         search_start,XYZ_s,search_par); % Normal projection
               algorithm
73
74     iel = get_point_element([um, vm], ...
75         anls.parts(anls.contact.ms_pairs{i}(2))); %Fin element
               number of
76                                                   %um,vm
77
78     % Retrieve the element of the normal proj point um,vm:
79     elm = anls.parts(anls.contact.ms_pairs{i}(2)).els(iel);
80     ncp_em = anls.parts(anls.contact.ms_pairs{i}(2)).els.ncp_e;
81
82     %Calculate current xyz coordinate of control points:
83     xyz_CPm = reshape(elm.CP(:,:,1:3),[ncp_em,3]);
84
85     %Compute base functions to get drduv:
86     [BFm] = compute_BF_elementbased([um,vm],elm,1);
87     dRduvm = BFm(2:3,:)';
```

```
88
89      %Get base functions:
90      [~,g3,lg3] = get_base_func(dRduvm,xyz_CPm);
91
92      %Calculate normal vector n (normalized):
93      n_shell = [0; 0; 0];
94      n_shell(1) = g3(1)/lg3;
95      n_shell(2) = g3(2)/lg3;
96      n_shell(3) = g3(3)/lg3; %normalization of g3 by components
97
98
99      % Calculate normal gap and check sign:
100     [XYZ_m] = get_point_coord([um,vm],...
101         anls.parts(anls.contact.ms_pairs{i}(2)).patch);
102     gap = (XYZ_s-XYZ_m)*n_shell;
103     if gap < 0
104         n_sign = -1;
105     end
106
107     if gap > 0
108         n_sign = 1;
109     end
110     n_sign_ms_pairs(i) = n_sign;
111 end % Loop through master-slave pairs
112
113
114 % ---------------------------------------------------------
115 % LOAD STEPS / NEWTON-LOOP
116
117 for il = 1:n_step   % % starting step(is step counter)
118
119     lambda_step = il/n_step;
120     Fe_active = lambda_step*F; %Define external load level
121                                %at the present step
122     lambda_plot(il+1) = lambda_step; %for plotting
123
124     disp('***************')
125     disp([' Step: ', num2str(il)]);
126
127 %-------------- Iterations loop :
128     for it = 1:n_iter % (it is counter of iterations for the step)
129
130         %***CALC STIFFNESS AND INTERNAL FORCE***
```

```
131         add_stiff_var.u_active_prev = u_active;    % update
                displacement vec
132
133         if anls.NL_geo == 0 % check that nonlinear analysis is
                specified,
134                         % error if nlinear
135             disp('***ERROR calculating stiffness matrices:
                    Analysis is ')
136             disp('specified as linear in anls.NL, have to be
                    nonlin!***')
137         end
138
139         [stiffness] = stiff_mat(anls,add_stiff_var);
140         [stiffness] = stiff_mat_coupling(anls,stiffness,
                add_stiff_var); %add coupling
141
142
143         % Calculate contact contributions:
144        for i = 1:length(anls.contact.ms_pairs) %Loop master-slave
               pairs
145            %Pass on current master-slave pair and contact normal
                   sign
146             add_stiff_var_contact.n_sign =  n_sign_ms_pairs(i);
147             add_stiff_var_contact.ms_pairs = anls.contact.ms_pairs
                    {i};
148
149             % Calculate contact stiffness and force contribution:
150             [stiffness, anls] = stiff_mat_Contact_GPTS_Pen...
151                 (anls,stiffness,add_stiff_var,
                        add_stiff_var_contact);
152
153             % without k_geo implemented:
154             %[stiffness, anls] = stiff_mat_Contact_GPTS_Pen_Simple
                    ...
155             %(anls,stiffness,add_stiff_var,add_stiff_var_contact);
156         end
157
158          % Retrieve stiffness and force residual:
159         K = stiffness.K;
160         Fi_active = stiffness.Fi_active;
161
162         Fr = Fi_active-Fe_active; % Reasidual force vectoer
163
```

```
164         % Calculate residual for convergence:
165         % relative criterion wrt to external forces magnitude:
166         residual = sqrt(Fr'*Fr)/sqrt(Fe_active'*Fe_active);
167         % norm of the energy increment:
168         %residual = abs(Fr'*u_active)/abs(Fe_active'*u_active);
169 
170         disp(['     iteration: ', num2str(it), ', res: ', num2str(
                residual)]);
171 
172         % Check convergence:
173         if (residual < tol) %check if convergence occurred
174             break;          %exit iteration loop
175         end
176         disp(['Residual: ', num2str(residual)])
177 
178         % Solve system and update displacements:
179         du_it_active = -K\Fr; %CHANGED SIGNsolving the iteration
                linearly,
180                               %get increment of the displacment
                                    for the
181                               %iteration step
182 
183         u_active = u_active+du_it_active; % update displ of active
                dof
184         % Update deformed CPs of the patches
185         [anls] = compute_deformed_CP(anls,u_active);
186     end % Loop iterations
187 
188     iters(il) = it; %store number of iterations for the load step
189     solution.iters = iters;
190 
191     u_active_step(:,il+1) = u_active; % vector collecting displ at
            each step
192 
193 
194     % Update global displacement vector u:
195     u = cell(length(anls.parts),1); % initialize
196     for ip = 1:length(anls.parts) %loop over patches
197 
198         ncp_p = anls.parts(ip).patch.ncp; % number of CP for the
                patch
199         ndof_cp = anls.parts(ip).ndof_cp;
200 
```

```
201         u{ip} = zeros(ncp_p*ndof_cp,1); % initialize
202         for icp = 1:ncp_p %loop over CP of the patch
203             for dir = 1:ndof_cp % loop over dof for each cp
204
205                 if ( anls.parts(ip).connectivity.ID(dir,icp)~=0 )
                        % if the dof is free
206                     u{ip}( ndof_cp*(icp-1)+dir ) = u_active( anls.
                            parts(ip).connectivity.ID(dir,icp) ); %u(a
                             control point) = u_active
207                 end
208             end
209
210         end %loop over CP of the patch
211     end % loop over patches
212
213
214     [displ] = get_point_displ(plot_coord,anls.parts(patch_plot),u{
            patch_plot});
215     u_plot(il+1) = plot_sign*displ(plot_dir);
216
217     %live plotting of load-displ
218     addpoints(h,u_plot(il+1),lambda_plot(il+1));
219     drawnow
220
221     %***PLOT CURRENT CONFIGURATION***
222     if rem(il,1)==0 %Plot every x load step
223         solution.d = u;
224         solution.d_active = u_active;
225         plot_factor = 1;        %magnification factor
226         figure
227         plot_structure_deformed(anls,solution,plot_factor,
                resolution,'mesh')
228         %'mesh','num_elem','basis','num_basis','thickness'
229         xlabel('x')
230         ylabel('y')
231         zlabel('z')
232         title(['Deformed structure load step ', num2str(il), ''])
233     end
234
235
236 end % Loop load steps: Newton-Raphson solution loop
237
238 lambda_plot(il+2:end) =[];
```

```
239 u_plot(il+2:end) =[];
240
241 % ------------------------------------------------------
242 % SOLUTION, OUTPUT
243 solution.d = u;
244 solution.d_active = u_active;
245
246 u_active_step(:,il+2:end) = [];
247 solution.d_active_step = u_active_step; % vector collecting displ
        at each step
248
249 solution.Fi_active = Fi_active;
250
251 % output nonlinear load-displ curve
252 solution.u_plot = u_plot;
253 solution.lambda_plot = lambda_plot;
254
255 end %function
```

B.3 Solver function: 2 Step Point Search and external load

```
1  %-------------------2 STEP SEARCH SOLVER-----------------
2  % ****DESCRIPTION****
3  % Solves nonlinear problem by Newton-Raphson iterations.
4
5  % 2 step search:
6  %   Creates matrix storing for each slave element a closest master
       element.
7  %   The matrix is passed on to the stiff_mat function.
8
9  % Created by Embla L. Holten , 2019.
10 % General setup from Davide Proserpios codes
11
12
13 %****CONTACT STIFF MAT FUNCTION CALLED****
14 %   stiff_mat_Contact_GPTS_Pen_2StepSearch.m    Inclides 2 Step
       Search and
15 %                                               geometric
       stiffness kgeo
16
17 %------------------------------------------------------
```

```
18 function [solution, anls] = solve_Contact_Newton_2StepSearch...
19     (anls, resolution)
20
21 % ------------------------------------------------------
22 % SOLVER INPUT AND INITIALISATION
23
24 %***DEFINE PARAMETERS FOR NONLINEAR SOLVING***
25 n_step = anls.par_solv.nstep;   % number of load steps
26 n_iter = anls.par_solv.niter;   % max number of iterations for a
       load step
27 tol    = anls.par_solv.tol;     % tolerance on residual for exit
       iterations
28 ndof = anls.ndof;
29 it = 0;                         %Initialize iterations for storing
30
31 %***DEFINE PARAMETERS FOR PLOTTING***
32 patch_plot = anls.par_plot.patch;    % patch to plot
33 plot_coord = anls.par_plot.coord;    % point to plot the
       displacement
34 plot_dir   = anls.par_plot.dir;      % direction of displacement
       to plot
35 plot_sign  = anls.par_plot.sign;     % sign of the displacement
36 iters = zeros(n_step,1);
37
38 F = anls.F; %Extract force vector
39
40 %***INITIALIZE DISPL***
41 u_active = zeros(ndof,1);
42 u_active_step = zeros(ndof,n_step+1); %initialize vector
       collecting displ
43                                       %at each step
44
45 %***INITIALIZE LIVE PLOTTING OF LOAD-DISPL***
46 lambda_plot = zeros(n_step+1,1);
47 u_plot = zeros(n_step+1,1);
48
49 figure()
50 xlabel('displacement')
51 ylabel('load fraction')
52 title(['point par. coord. (', num2str(plot_coord), '),
       displacement dir=', num2str(plot_dir)])
53 grid on
54 h = animatedline('Marker','o');
```

```
addpoints(h,u_plot,lambda_plot);
drawnow

% ------------------------------------------------------
% DEFINE AND INITIALISE CONTACT PARAMETERS

% Preallocate vector of contact normal signs:
n_sign_ms_pairs = zeros(length(anls.contact.ms_pairs),1);
% Preallocate struct of master and slave closest element pairs:
contact_pairs = struct;

% Loop master-slave pairs:
for i = 1:length(anls.contact.ms_pairs)
        % Preallocate master-slave distances matrix for the 2 step
            point
        % search
    contact_pairs(i).M_min_dist_sm ...
        = zeros(length(anls.parts(anls.contact.ms_pairs{i}(1)).els
            ), 4);
        % rows = slave elements 1,2,3,...number of slave elem
        % cols = info about closest master element to the slave
        % element el_s:
        % [u_cm, v_cm, iel_m, normal_d_center_elem]
    contact_pairs(i).M_contact_partners ...
        = zeros(length(anls.parts(anls.contact.ms_pairs{i}(1)).els
            ), 4);
        % same structure as M_min_dist_sm except:
        % cols = 0 if not potential contact

    % Calculate contact normal sign at the master surface:
    us = 0.5; vs = 0.5; %Choose an arbitrary point us and vs on
        slave
    [XYZ_s] = get_point_coord([us, vs],...              %Get
        cartesian coor
        anls.parts(anls.contact.ms_pairs{i}(1)).patch); %of point
            (us,vs)

    % Get normal projection point onto master surface from [XYZ_s]
    search_par = anls.contact.search_par; %Retrieve analysis data
    search_start = search_par{1}; %Start of normal proj point
        search

    [um,vm,~] = ...
```

```
91          point_project_surf(anls.parts(anls.contact.ms_pairs{i}(2))
                .patch,...
92          search_start,XYZ_s,search_par); % Normal projection
                algorithm
93
94      iel = get_point_element([um, vm], ...
95          anls.parts(anls.contact.ms_pairs{i}(2))); %Fin element
                number of
96                                                    %um,vm
97
98      % Retrieve the element of the normal proj point um,vm:
99      elm = anls.parts(anls.contact.ms_pairs{i}(2)).els(iel);
100     ncp_em = anls.parts(anls.contact.ms_pairs{i}(2)).els.ncp_e;
101
102     %Calculate current xyz coordinate of control points:
103     xyz_CPm = reshape(elm.CP(:,:,1:3),[ncp_em,3]);
104
105     %Compute base functions to get drduv:
106     [BFm] = compute_BF_elementbased([um,vm],elm,1);
107     dRduvm = BFm(2:3,:)';
108
109     %Get base functions:
110     [~,g3,lg3] = get_base_func(dRduvm,xyz_CPm);
111
112     %Calculate normal vector n (normalized):
113     n_shell = [0; 0; 0];
114     n_shell(1) = g3(1)/lg3;
115     n_shell(2) = g3(2)/lg3;
116     n_shell(3) = g3(3)/lg3; %normalization of g3 by components
117
118
119     % Calculate normal gap and check sign:
120     [XYZ_m] = get_point_coord([um,vm],...
121         anls.parts(anls.contact.ms_pairs{i}(2)).patch);
122     gap = (XYZ_s-XYZ_m)*n_shell;
123     if gap < 0
124         n_sign = -1;
125     end
126
127     if gap > 0
128         n_sign = 1;
129     end
130     n_sign_ms_pairs(i) = n_sign;
```

```
131 end % Loop through master-slave pairs
132
133
134 % ------------------------------------------------------
135 % LOAD STEPS / NEWTON-LOOP
136
137 for il = 1:n_step   % starting step(il step counter)
138
139     lambda_step = il/n_step;
140     Fe_active = lambda_step*F; %Define external load level
141                                %at the present step
142     lambda_plot(il+1) = lambda_step; %for plotting
143
144     disp('**************')
145     disp([' Step: ', num2str(il)]);
146
147 %-------------- Find potential contact partners (2 step point
        search):
148         for i = 1:length(anls.contact.ms_pairs)
149             m_els = anls.parts(anls.contact.ms_pairs{i}(2)).els;
150             s_els = anls.parts(anls.contact.ms_pairs{i}(1)).els;
151
152             % Loop slave elements:
153             for iel_s = 1:length(s_els)
154
155                 % Initialize matrices for 2 step point search:
156                 contact_pairs(i).M_contact_partners(iel_s,:) = 0;
157                 contact_pairs(i).M_min_dist_sm(iel_s,:) = 0;
158
159                 % Calculate center point of slave element, (u_cs,
                        v_cs):
160                 el_s = s_els(iel_s);
161                 p_s = el_s.p;
162                 U_s = el_s.U{1};
163                 V_s = el_s.U{2};
164                 u_cs = ( U_s(p_s(1)+1)+U_s(p_s(1)+2) )/2;
165                 v_cs = ( V_s(p_s(2)+1)+V_s(p_s(2)+2) )/2;
166
167                 %Find cartesian coordinate of (u_cs,v_cs):
168                 [XYZ_cs] = get_point_coord([u_cs,v_cs],...
169                     anls.parts(anls.contact.ms_pairs{i}(1)).
                            patch_def);
170
```

```
171                 % Loop through master elements:
172                 d_center_elem_min = 1000; % Initialize minimum
                        dist
173                 for iel_m = 1:length(m_els)
174                     % Calculate center point of master element:
175                     el_m = m_els(iel_m);
176                     p_m = el_m.p;
177                     U_m = el_m.U{1};
178                     V_m = el_m.U{2};
179                     u_cm = ( U_m(p_m(1)+1)+U_m(p_m(1)+2) )/2;
180                     v_cm = ( V_m(p_m(2)+1)+V_m(p_m(2)+2) )/2;
181                     %Find cartesian coordinate of (u_cm,v_cm):
182                     [XYZ_cm] = get_point_coord([u_cm,v_cm],...
183                         anls.parts(anls.contact.ms_pairs{i}(2)).
                            patch_def);
184
185                     % Calculate absolute distance between master
                            and slave:
186                     d_center_elem = (XYZ_cs - XYZ_cm);
187                     abs_d_center_elem...
188                         = sqrt(d_center_elem(1)^2 + ...
189                         d_center_elem(2)^2 + d_center_elem(3)^2);
190                     d_center_elem = abs_d_center_elem;
191                     % Check if absolute distance is less than
                            minimum:
192                     if d_center_elem < d_center_elem_min
193                         contact_pairs(i).M_min_dist_sm(iel_s,:)...
194                             =[u_cm,v_cm,iel_m,d_center_elem];
195                         % Update current minimum:
196                         d_center_elem_min = d_center_elem;
197                     end
198                 end  %Loop master elements
199
200                 % Check if current distance is less than
                        acceptable for
201                 % including in contact calculations:
202                 if d_center_elem_min < ...
203                         anls.contact.potential_contact_d
204                     contact_pairs(i).M_contact_partners(iel_s,:)
                            ...
205                         = contact_pairs(i).M_min_dist_sm(iel_s,:);
206                 end
207             end  % Loop slave elements
```

```
208         end %Loop master-slave pairs
209
210 %-------------- Iterations loop :
211     for it = 1:n_iter % (it is counter of iterations for the step)
212         if (it == (n_iter-1) ) %check if convergence occurred or
                max iters
213             disp('WARNING: Reached max iterations for the load
                    step.')
214             break;
215         end
216         add_stiff_var.u_active_prev = u_active;    % update
                displacement vec
217
218         if anls.NL_geo == 0 % check that nonlinear analysis is
                specified,
219                             % error if nlinear
220             disp('***ERROR calculating stiffness matrices:
                    Analysis is ')
221             disp('specified as linear in anls.NL, have to be
                    nonlin!***')
222         end
223
224         [stiffness] = stiff_mat(anls,add_stiff_var);
225         [stiffness] = stiff_mat_coupling(anls,stiffness,
                add_stiff_var); %add coupling
226
227        % Calculate contact contributions:
228        for i = 1:length(anls.contact.ms_pairs) %Loop master-slave
               pairs
229             %Pass on current master-slave pair, contact partner
                    matrix
230             %for 2 step point search and contact normal sign
231             add_stiff_var_contact.M_contact_partners = ...
232                 contact_pairs(i).M_contact_partners;
233             add_stiff_var_contact.n_sign =  n_sign_ms_pairs(i);
234             add_stiff_var_contact.ms_pairs = anls.contact.ms_pairs
                    {i};
235
236             % Calculate contact stiffness and force contribution:
237             [stiffness, anls] =
                    stiff_mat_Contact_GPTS_Pen_2StepSearch...
238                 (anls,stiffness,add_stiff_var,
                        add_stiff_var_contact);
```

```
239         end
240
241         % Retrieve stiffness and force residual:
242         K = stiffness.K;
243         Fi_active = stiffness.Fi_active;
244
245         Fr = Fi_active-Fe_active; % Reasidual force vectoer
246
247         % Calculate residual for convergence:
248         % relative criterion wrt to external forces magnitude:
249         residual = sqrt(Fr'*Fr)/sqrt(Fe_active'*Fe_active);
250         % norm of the energy increment:
251         %residual = abs(Fr'*u_active)/abs(Fe_active'*u_active);
252
253         disp(['     iteration: ', num2str(it), ', res: ', num2str(
                residual)]);
254
255         % Check convergence:
256         if (residual < tol) %check if convergence occurred
257             break;          %exit iteration loop
258         end
259         disp(['Residual: ', num2str(residual)])
260
261         % Solve system and update displacements:
262         du_it_active = -K\Fr; %CHANGED SIGNsolving the iteration
                linearly,
263                               %get increment of the displacment
                                    for the
264                               %iteration step
265
266         u_active = u_active+du_it_active; % update displ of active
                dof
267         % Update deformed CPs of the patches
268         [anls] = compute_deformed_CP(anls,u_active);
269     end % Loop iterations
270
271     iters(il) = it; %store number of iterations for the load step
272     solution.iters = iters;
273
274     u_active_step(:,il+1) = u_active; % vector collecting displ at
            steps
275
276
```

```
277     % Update global displacement vector u:
278     u = cell(length(anls.parts),1); % initialize
279     for ip = 1:length(anls.parts) %loop over patches
280
281         ncp_p = anls.parts(ip).patch.ncp; % number of CP for the
                patch
282         ndof_cp = anls.parts(ip).ndof_cp;
283
284         u{ip} = zeros(ncp_p*ndof_cp,1); % initialize
285         for icp = 1:ncp_p %loop over CP of the patch
286             for dir = 1:ndof_cp % loop over dof for each cp
287                 %If the dof is free, update:
288                 if ( anls.parts(ip).connectivity.ID(dir,icp)~=0 )
289                     u{ip}( ndof_cp*(icp-1)+dir ) = u_active...
290                         ( anls.parts(ip).connectivity.ID(dir,icp)
                            );
291                 end
292             end
293
294         end %loop over CP of the patch
295     end % loop over patches
296
297
298
299
300
301     [displ] = get_point_displ(plot_coord,anls.parts(patch_plot),u{
            patch_plot});
302     u_plot(il+1) = plot_sign*displ(plot_dir);
303
304     %live plotting of load-displ
305     addpoints(h,u_plot(il+1),lambda_plot(il+1));
306     drawnow
307
308     % Plot current configuration:
309     if rem(il,3)==0 %Plot every x load step
310         solution.d = u;
311         solution.d_active = u_active;
312
313         plot_factor = 1;        %magnification factor
314         figure
315         plot_structure_deformed(anls,solution,plot_factor,
                resolution,'mesh')   %'mesh','num_elem','basis','
```

```
            num_basis','thickness'
316         xlabel('x')
317         ylabel('y')
318         zlabel('z')
319         title(['Deformed structure load step ', num2str(il), ''])
320     end
321
322
323 end % Loop load steps: Newton-Raphson solution loop
324
325 lambda_plot(il+2:end) =[];
326 u_plot(il+2:end) =[];
327
328 % ------------------------------------------------------
329 % SOLUTION, OUTPUT
330
331 solution.d = u;
332 solution.d_active = u_active;
333
334 u_active_step(:,il+2:end) = [];
335 solution.d_active_step = u_active_step; % vector collecting displ
        at each step
336
337 solution.Fi_active = Fi_active;
338
339 % output nonlinear load-displ curve
340 solution.u_plot = u_plot;
341 solution.lambda_plot = lambda_plot;
342
343 end %function
```

B.4 Solver function: Cylinder example with 2 step point search

```
1 %-------------------CYLINDER SQUEEZE SOLVER----------
2 % ****DESCRIPTION****
3 % Solves nonlinear problem by Newton-Raphson iterations wuth
      moving rigid
4 % bodies for every load step. The code is adapted to the cylinder
      example.
```

```
5
6  % Created by Embla L. Holten , 2019.
7  % General setup from Davide Proserpios codes
8
9
10 %****STIFF MAT FUNCTION CALLED****
11 %    stiff_mat_Contact_Cylinder_Squeeze.m        Inclides 2 Step
       Search and
12 %                                                geometric
       stiffness kgeo
13
14 %-------------------------------------------------------
15
16
17 function [solution, anls] = solve_Contact_Cylinder_Squeeze...
18     (anls, resolution)
19
20 % -------------------------------------------------------
21 % SOLVER INPUT AND INITIALISATION
22
23 %***DEFINE PARAMETERS FOR NONLINEAR SOLVING***
24 n_step = anls.par_solv.nstep;   % number of load steps
25 n_iter = anls.par_solv.niter;   % max number of iterations for a
       load step
26 tol    = anls.par_solv.tol;     % tolerance on residual for exit
       iterations
27 ndof = anls.ndof;
28
29 %***DEFINE PARAMETERS FOR PLOTTING***
30 patch_plot = anls.par_plot.patch;    % patch to plot
31 plot_coord = anls.par_plot.coord;    % point to plot the
       displacement
32 plot_dir   = anls.par_plot.dir;      % direction of displacement
       to plot
33 plot_sign  = anls.par_plot.sign;     % sign of the displacement
34
35 iters = zeros(n_step,1);
36
37 %***INITIALIZE DISPL***
38 u_active = zeros(ndof,1);
39 u_active_step = zeros(ndof,n_step+1); %initialize vector
       collecting displ
40                                       %at each step
```

```
41
42 %***INITIALIZE LIVE PLOTTING OF LOAD-DISPL***
43 lambda_plot = zeros(n_step+1,1);
44 u_plot = zeros(n_step+1,1);
45 figure()
46 xlabel('displacement')
47 ylabel('load fraction')
48 title(['point par. coord. (', ...
49     num2str(plot_coord(1,:)), '), displacement dir=', num2str(
       plot_dir)])
50 grid on
51 h = animatedline('Marker','o');
52 addpoints(h,u_plot,lambda_plot);
53 drawnow
54
55
56 % -------------------------------------------------------
57 % DEFINE AND INITIALISE CONTACT PARAMETERS
58
59 % Preallocate vector of contact normal signs:
60 n_sign_ms_pairs = zeros(length(anls.contact.ms_pairs),1);
61 % Preallocate struct of master and slave closest element pairs:
62 contact_pairs = struct;
63
64 % Loop master-slave pairs:
65 for i = 1:length(anls.contact.ms_pairs)
66         % Preallocate master-slave distances matrix for the 2 step
              point
67         % search
68     contact_pairs(i).M_min_dist_sm ...
69         = zeros(length(anls.parts(anls.contact.ms_pairs{i}(1)).els
              ), 4);
70         % rows = slave elements 1,2,3,...number of slave elem
71         % cols = info about closest master element to the slave
72         % element el_s:
73         % [u_cm, v_cm, iel_m, normal_d_center_elem]
74     contact_pairs(i).M_contact_partners ...
75         = zeros(length(anls.parts(anls.contact.ms_pairs{i}(1)).els
              ), 4);
76         % same structure as M_min_dist_sm except:
77         % cols = 0 if not potential contact
78
79     % Calculate contact normal sign at the master surface:
```

```
80      us = 0.5; vs = 0.5; %Choose an arbitrary point us and vs on
            slave
81      [XYZ_s] = get_point_coord([us, vs],...              %Get
            cartesian coor
82          anls.parts(anls.contact.ms_pairs{i}(1)).patch); %of point
                (us,vs)
83
84      % Get normal projection point onto master surface from [XYZ_s]
85      search_par = anls.contact.search_par; %Retrieve analysis data
86      search_start = search_par{1}; %Start of normal proj point
            search
87
88      [um,vm,~] = ...
89          point_project_surf(anls.parts(anls.contact.ms_pairs{i}(2))
                .patch,...
90          search_start,XYZ_s,search_par); % Normal projection
                algorithm
91
92      iel = get_point_element([um, vm], ...
93          anls.parts(anls.contact.ms_pairs{i}(2))); %Fin element
                number of
94                                                    %um,vm
95
96      % Retrieve the element of the normal proj point um,vm:
97      elm = anls.parts(anls.contact.ms_pairs{i}(2)).els(iel);
98      ncp_em = anls.parts(anls.contact.ms_pairs{i}(2)).els.ncp_e;
99
100     %Calculate current xyz coordinate of control points:
101     xyz_CPm = reshape(elm.CP(:,:,1:3),[ncp_em,3]);
102
103     %Compute base functions to get drduv:
104     [BFm] = compute_BF_elementbased([um,vm],elm,1);
105     dRduvm = BFm(2:3,:)';
106
107     %Get base functions:
108     [~,g3,lg3] = get_base_func(dRduvm,xyz_CPm);
109
110     %Calculate normal vector n (normalized):
111     n_shell = [0; 0; 0];
112     n_shell(1) = g3(1)/lg3;
113     n_shell(2) = g3(2)/lg3;
114     n_shell(3) = g3(3)/lg3; %normalization of g3 by components
115
```

```
116
117     % Calculate normal gap and check sign:
118     [XYZ_m] = get_point_coord([um,vm],...
119         anls.parts(anls.contact.ms_pairs{i}(2)).patch);
120     gap = (XYZ_s-XYZ_m)*n_shell;
121     if gap < 0
122         n_sign = -1;
123     end
124
125     if gap > 0
126         n_sign = 1;
127     end
128     n_sign_ms_pairs(i) = n_sign;
129 end % Loop through master-slave pairs
130
131 % -------------------------------------------------------
132 % LOAD STEPS / NEWTON-LOOP
133
134 for il = 1:n_step   % starting step(il step counter)
135
136     lambda_step = il/n_step;
137     lambda_plot(il+1) = lambda_step; %for plotting
138
139     disp('**************')
140     disp([' Step: ', num2str(il)]);
141 %-------------- Move rigid bodies:
142     % Add displacement to CPs of all rigid patches to move:
143     for i = 1:length(anls.contact.d_rigid) % Loop rigid patches
144         %Retrieve control points of rigid patch:
145         CP_p = anls.parts( anls.contact.d_rigid(i).part_nr ).
                patch_def.CP;
146         for iv = 1:length(CP_p(1,:,1)) %Loop CPs in parametric v-
                direction
147             for iu = 1:length(CP_p(:,1,1)) %Loop CPs in param. u-
                    dir.
148                 % Add prescribed displacement:
149                 CP_p(iu,iv,3) = CP_p(iu,iv,3) + ...
150                     anls.contact.d_rigid(i).d_step_dir(3)/n_step;
                        %z-dir
151                 CP_p(iu,iv,2) = CP_p(iu,iv,2) + ...
152                     anls.contact.d_rigid(i).d_step_dir(2)/n_step;
                        %y-dir
153                 CP_p(iu,iv,1) = CP_p(iu,iv,1) + ...
```

```
154                     anls.contact.d_rigid(i).d_step_dir(1)/n_step;
                            %x-dir
155             end
156         end
157         anls.parts( anls.contact.d_rigid(i).part_nr ).patch_def.CP
                 = CP_p;
158         anls.parts( anls.contact.d_rigid(i).part_nr ).patch.CP =
                CP_p;
159
160         % dd displacement to CPs of all rigid elements to move:
161         for iel = 1:...
162                 length(anls.parts( anls.contact.d_rigid(i).part_nr
                         ).els)
163             CP_elem = ...
164                 anls.parts( anls.contact.d_rigid(i).part_nr ).els(
                    iel).CP;
165             for iv = 1:length(CP_elem(1,:,1))
166                 for iu = 1:length(CP_elem(:,1,1))
167                         CP_elem(iu,iv,3) = CP_elem(iu,iv,3) + ...
168                             anls.contact.d_rigid(i).d_step_dir(3)/
                                n_step;
169                         CP_elem(iu,iv,2) = CP_elem(iu,iv,2) + ...
170                             anls.contact.d_rigid(i).d_step_dir(2)/
                                n_step;
171                         CP_elem(iu,iv,1) = CP_elem(iu,iv,1) + ...
172                             anls.contact.d_rigid(i).d_step_dir(1)/
                                n_step;
173                 end
174             end
175             anls.parts( anls.contact.d_rigid(i).part_nr ).els(iel)
                    .CP = ...
176                 CP_elem;
177         end
178     end % Loop all rigid parts
179
180 %-------------- Find potential contact partners (2 step point
        search):
181         for i = 1:length(anls.contact.ms_pairs)
182             m_els = anls.parts(anls.contact.ms_pairs{i}(2)).els;
183             s_els = anls.parts(anls.contact.ms_pairs{i}(1)).els;
184
185             % Loop slave elements:
186             for iel_s = 1:length(s_els)
```

```
187
188                 % Initialize matrices for 2 step point search:
189                 contact_pairs(i).M_contact_partners(iel_s,:) = 0;
190                 contact_pairs(i).M_min_dist_sm(iel_s,:) = 0;
191
192                 % Calculate center point of slave element, (u_cs,
                        v_cs):
193                 el_s = s_els(iel_s);
194                 p_s = el_s.p;
195                 U_s = el_s.U{1};
196                 V_s = el_s.U{2};
197                 u_cs = ( U_s(p_s(1)+1)+U_s(p_s(1)+2) )/2;
198                 v_cs = ( V_s(p_s(2)+1)+V_s(p_s(2)+2) )/2;
199
200                 %Find cartesian coordinate of (u_cs,v_cs):
201                 [XYZ_cs] = get_point_coord([u_cs,v_cs],...
202                     anls.parts(anls.contact.ms_pairs{i}(1)).
                            patch_def);
203
204                 % Loop through master elements:
205                 d_center_elem_min = 1000; % Initialize minimum
                        dist
206                 for iel_m = 1:length(m_els)
207                     % Calculate center point of master element:
208                     el_m = m_els(iel_m);
209                     p_m = el_m.p;
210                     U_m = el_m.U{1};
211                     V_m = el_m.U{2};
212                     u_cm = ( U_m(p_m(1)+1)+U_m(p_m(1)+2) )/2;
213                     v_cm = ( V_m(p_m(2)+1)+V_m(p_m(2)+2) )/2;
214                     %Find cartesian coordinate of (u_cm,v_cm):
215                     [XYZ_cm] = get_point_coord([u_cm,v_cm],...
216                         anls.parts(anls.contact.ms_pairs{i}(2)).
                                patch_def);
217
218                     % Calculate absolute distance between master
                            and slave:
219                     d_center_elem = (XYZ_cs - XYZ_cm);
220                     abs_d_center_elem...
221                         = sqrt(d_center_elem(1)^2 + ...
222                         d_center_elem(2)^2 + d_center_elem(3)^2);
223                     d_center_elem = abs_d_center_elem;
224                     % Check if absolute distance is less than
```

```
                        minimum:
225                     if d_center_elem < d_center_elem_min
226                         contact_pairs(i).M_min_dist_sm(iel_s,:)...
227                             =[u_cm,v_cm,iel_m,d_center_elem];
228                         % Update current minimum:
229                         d_center_elem_min = d_center_elem;
230                     end
231                 end  %Loop master elements
232
233                 % Check if current distance is less than
                        acceptable for
234                 % including in contact calculations:
235                 if d_center_elem_min < ...
236                         anls.contact.potential_contact_d
237                     contact_pairs(i).M_contact_partners(iel_s,:)
                            ...
238                         = contact_pairs(i).M_min_dist_sm(iel_s,:);
239                 end
240             end  % Loop slave elements
241         end %Loop master-slave pairs
242
243 %-------------- Iterations loop :
244     for it = 1:n_iter % (it is counter of iterations for the step)
245         if (it == (n_iter-1) ) %check if convergence occurred or
                max iters
246             disp('WARNING: Reached max iterations for the load
                    step.')
247             break;   %exit iteration loop -> next load step
248         end
249
250         add_stiff_var.u_active_prev = u_active; % update
                displacement vec
251
252         if anls.NL_geo == 0 % check that nonlinear analysis is
                specified,
253                         % error if linear
254             disp('***ERROR calculating stiffness matrices:
                    Analysis is ')
255             disp('specified as linear in anls.NL, have to be
                    nonlin!***')
256         end
257
258         [stiffness] = stiff_mat(anls,add_stiff_var);
```

```
259          [stiffness] = stiff_mat_coupling(anls,stiffness,
                 add_stiff_var);
260
261         % Calculate contact contributions:
262         for i = 1:length(anls.contact.ms_pairs) %Loop master-slave
                pairs
263              %Pass on current master-slave pair, contact partner
                     matrix
264              %for 2 step point search and contact normal sign
265              add_stiff_var_contact.M_contact_partners = ...
266                  contact_pairs(i).M_contact_partners;
267              add_stiff_var_contact.n_sign =  n_sign_ms_pairs(i);
268              add_stiff_var_contact.ms_pairs = anls.contact.ms_pairs
                     {i};
269
270              % Calculate contact stiffness and force contribution:
271              [stiffness, anls] = ...
272                  stiff_mat_Contact_Cylinder_Squeeze...
273                  (anls,stiffness,add_stiff_var,
                         add_stiff_var_contact);
274          end
275
276          % Retrieve stiffness and force residual:
277          K = stiffness.K;
278          Fi_active = stiffness.Fi_active;
279
280          Fr = Fi_active; %No external load, residual = internal
                 forces
281
282          % Calculate residual for convergence:
283          residual = sqrt(Fr'*Fr);
284
285          disp(['     iteration: ', num2str(it), ', res: ',...
286              num2str(residual)]);
287
288          % Check convergence:
289          if (residual < tol )
290              break;
291          end
292          disp(['Residual: ', num2str(residual)])
293
294          % Solve system and update displacements:
295          du_it_active = -K\Fr;
```

```
296
297         u_active = u_active+du_it_active; % update displ of active
                dof
298         % Compute displaced CPS of the patches:
299         [anls] = compute_deformed_CP(anls,u_active);
300     end % Loop iterations
301
302     iters(il) = it; %store number of iterations for the load step
303     solution.iters = iters;
304
305     u_active_step(:,il+1) = u_active; % collect displ at each step
306
307
308     % Update global displacement vector u:
309     u = cell(length(anls.parts),1); % initialize
310     for ip = 1:length(anls.parts) %loop over patches
311
312         ncp_p = anls.parts(ip).patch.ncp; % number of CP for the
                patch
313         ndof_cp = anls.parts(ip).ndof_cp;
314
315         u{ip} = zeros(ncp_p*ndof_cp,1); % initialize
316         for icp = 1:ncp_p %loop over CP of the patch
317             for dir = 1:ndof_cp % loop over dof for each cp
318                 %If the dof is free, update:
319                 if ( anls.parts(ip).connectivity.ID(dir,icp)~=0 )
320                     u{ip}( ndof_cp*(icp-1)+dir ) = ...
321                         u_active...
322                         ( anls.parts(ip).connectivity.ID(dir,icp)
                            );
323                 end
324             end
325
326         end %Loop over CP of the patch
327     end % Loop over patches
328
329     % Store displacement for plotting:
330     [displ] = get_point_displ(plot_coord(1,:),anls.parts(
            patch_plot),...
331         u{patch_plot});
332     u_plot(il+1) = plot_sign*displ(plot_dir);
333
334     % Live plotting of load-displ
```

```
335     addpoints(h,u_plot(il+1),lambda_plot(il+1));
336     drawnow
337
338
339     % Plot current configuration:
340     if rem(il,3)==0 %Plot every x load step
341         solution.d = u;
342         solution.d_active = u_active;
343
344         plot_factor = 1;        %magnification factor
345         figure
346         plot_structure_deformed...
347             (anls,solution,plot_factor,resolution,'mesh')
348             %'mesh','num_elem','basis','num_basis','thickness'
349         xlabel('x')
350         ylabel('y')
351         zlabel('z')
352         title(['Deformed structure load step ', num2str(il), ''])
353     end
354
355 end % Loop load steps: Newton-Raphson solution loop
356
357 lambda_plot(il+2:end) =[];
358 u_plot(il+2:end) =[];
359
360
361 % ---------------------------------------------------------
362 % SOLUTION, OUTPUT
363
364 solution.d = u;
365 solution.d_active = u_active;
366
367 u_active_step(:,il+2:end) = [];
368 solution.d_active_step = u_active_step; % vector collecting displ
        at each step
369
370 solution.Fi_active = Fi_active;
371
372 % output nonlinear load-displ curve
373 solution.u_plot = u_plot;
374 solution.lambda_plot = lambda_plot;
375
376 end % Function
```

Appendix C

Contact contribution functions

The equation numbers in the functions refer to (De Lorenzis, Scott, Wriggers, Taylor and Zavarize, 2014).

C.1 Contact contribution function: Simplified GPTS and penalty without geometric stiffness

```
%------CONTACT CONTRIBUTIONS CALC. FUNCTION-------------
% ****DESCRIPTION****
% Calculates the contact contribution to the stiffness matrix and
    force
% residual. Gauss-point-to-segment discretization with the penalty
     method.
% Excluding geometric contact stiffness. Single point search step.

% Created by Embla L. Holten , 2019.



%-----------------------------------------------------

function [stiffness, anls] = stiff_mat_Contact_GPTS_Simple...
    (anls,stiffness,add_stiff_var,add_stiff_var_contact)
```

```
15
16 % -------------------------------------------------------
17 % EXTRACT ANALYSIS INPUT
18 alf = anls.contact.alf; %Penalty parameter
19 search_par = anls.contact.search_par; %Normal point projection
       parameters
20 search_start = search_par{1}; %Starting point for normal
       projection point
21                                %search.
22 K = stiffness.K; %Stiffness matrix from before
23 Fi_active = stiffness.Fi_active; %Internal forces fom before
24 u_active_prev = add_stiff_var.u_active_prev; %Displacements prev
       load step
25
26 % Extract info slave part:
27 part1 = anls.parts(add_stiff_var_contact.ms_pairs(1)); %Slave part
28 nel1 = length(part1.els);  %Number of elemnts in the patch
29
30 % Extract infor master part:
31 part2 = anls.parts(add_stiff_var_contact.ms_pairs(2)); %Master
       part
32 nel2 = length(part2.els);
33 spline_type2 = part2.spltyp;
34
35 % Calculate deformed control points master elements:
36 for iel=1:nel2
37     el2 = part2.els(iel);
38     [~, el2, ~] = compute_deformed_CP_elem(el2, part2.connectivity
           ,...
39         part2.ndof_cp, u_active_prev); %Assign to current element
               el2
40     part2.els(iel) = el2; %Assign to part
41
42 end
43
44 % -------------------------------------------------------
45 % DETECT CONTACT
46
47 %--------------- Loop slave elements:
48 for iel=1:nel1
49      % Retrieve information from slave element:
50     el = part1.els(iel); % Current slave element in slave element
           loop
```

```
51      integ = el.integ;    %Integration rule for the element
52      p = el.p;            %Polynomial degrees
53      J2 = el.J2;          %Jacobian
54      U = el.U{1};         %Knot vector
55      V = el.U{2};         %Knot vector
56
57      % Retrieve the displaced control point coordinates and add to
            element:
58      [xyz_CP_d, el, ~] = compute_deformed_CP_elem(el, part1.
            connectivity,...
59          part1.ndof_cp, u_active_prev); %Assign to current master
                elem el2
60      part1.els(iel) = el; %Assign to part
61
62      % Find gauss point positions in an element in u and v
            direction
63      [GP,GW] = gauss_point_weights(p,integ);
64
65  %-------------- Loop over Gauss Points in slave element el:
66      for igp = 1:length(GW)
67          %Find NURBS coordinate of gpoints:
68          ugp = ( U(p(1)+2)+U(p(1)+1) + GP(igp,1)*...
69              (U(p(1)+2)-U(p(1)+1)) )/2;
70          vgp = ( V(p(2)+2)+V(p(2)+1) + GP(igp,2)*...
71              (V(p(2)+2)-V(p(2)+1)) )/2;
72          gw = GW(igp);
73
74          %Find xyz coordinate of gp:
75          [XYZ_s] = get_point_coord([ugp,vgp],part1.patch_def);
76
77          % Calculate normal projection point on master from [XYZ_s
                ]:
78          [u2,v2,~] = point_project_surf(part2.patch_def,
                search_start,...
79              XYZ_s,search_par);
80          if (spline_type2==0) %NURBS
81              % Fin master element number related to projected point
                    :
82              iel2 = get_point_element([u2,v2],part2);
83          else
84              disp('Not NURBS! Not yet implemented in code')
85          end
86          el2 = part2.els(iel2);  %Assign to master part
```

```
87
88          % Calculate base functions at projection point:
89          [BF2] = compute_BF_patchbased([u2,v2],part2.patch_def,2);
90          dRduv2 = BF2(2:3,:)';
91
92          % Deformed CP of el2:
93          xyz_CP_d2 = reshape(el2.CP(:,:,1:3),[el2.ncp_e,3]);
94
95          % Basis vector 1, 2 and 3=jacobian for master elemente:
96          [g_2 ,g3_2,lg3_2] = get_base_func(dRduv2,xyz_CP_d2);
97
98          % Normal vector n (normalized) at projection point:
99          n = [0; 0; 0];
100         n(1) = g3_2(1)/lg3_2;
101         n(2) = g3_2(2)/lg3_2;
102         n(3) = g3_2(3)/lg3_2; %normalization of g3 by components
103
104
105         % Calculate normal gap function:
106         [XYZ_m] = get_point_coord([u2,v2],part2.patch_def);
107         n = n*add_stiff_var_contact.n_sign; %Correct sign of
                normal
108         gn = (XYZ_s - XYZ_m)*n; %Calculate normal gap
109
110        % Check for contact between Gauss point and projection
              point:
111         if gn<0
112 % -------------------------------------------------------
113 % IF CONTACT: CALCULATE CONTACT CONTRIBUTIONS
114             % Calculate base functions slave element:
115             [BF1] = compute_BF_patchbased([ugp,vgp],part1.
                    patch_def,1);
116             R1 = BF1(1,:)';          % Base functions (not
                    derivatives)
117             N1 = reshape(n*R1',3*length(R1),1);
118             % Derivatives of bf w.r.t. u and v in
119             % the considered GP (two columns [dN/du, dN/dv]):
120             dRduv = BF1(2:3,:)';
121
122             % Calculate Jacobian, slave element:
123             [~,~,J1] = get_base_func(dRduv,xyz_CP_d);
124
125             % Calculate base functions slave element:
```

```
126             R2 = BF2(1,:)';
127             N2 = reshape(n*R2',length(n)*length(R2),1);
128 
129             % Coupled contact element base functions:
130             N = zeros(length(N1)+length(N2),1); %Vertical vector
131             N( 1 : length(N1) ) = N1;
132             N( (length(N1)+1) : (length(N1)+length(N2)) ) = -N2; %
                    From Laura article T-splines
133 
134 %--------------- Final calculation of contact element contributions
    :
135             % Stiffness ke_c:
136             kea_c = alf*(N*N');
137             ke_c = kea_c*gw*J2*J1;
138 
139             % Force contribution:
140             fiea_c = alf*N*gn;
141             fie_c = fiea_c*gw*J2*J1;
142  %--------------- Assembly:
143             % Assemble into global K:
144             K = Kassembly_contact_fourLoops(el,el2,K,ke_c);
145 
146             %Assemble into global F:
147             for i = 1:el.ndof_e
148                 if (el.LM(i)~=0)
149                 globi = el.LM(i);
150                 Fi_active(globi) = Fi_active(globi) + fie_c(i);
151                 end
152             end
153 
154             for i = 1:el2.ndof_e
155                 if (el2.LM(i)~=0)
156                 globi = el2.LM(i);
157                 Fi_active(globi) = Fi_active(globi) + fie_c(i+el.
                        ndof_e);
158                 end
159             end
160         end % Contact condition: if gn < 0
161     end % Loop Gauss points
162 end % Loop slave elements
163 
164 % -------------------------------------------------------
165 % OUTPUT / RESULTS
```

```
stiffness.K = K;
stiffness.Fi_active = Fi_active;

end %function
```

C.2 Contact contribution function: GPTS and penalty with geometric stiffness

```
%-------------CONTACT CONTRIBUTIONS CALC. FUNCTION-----
% ****DESCRIPTION****
% Calculates the contact contribution to the stiffness matrix and force
% residual. Gauss-point-to-segment discretization with the penalty method.
% includes geometric contact stiffness. Single point search step.

% Created by Embla L. Holten , 2019.




%-----------------------------------------------------------



function [stiffness, anls] = stiff_mat_Contact_GPTS_Pen...
    (anls,stiffness,add_stiff_var,add_stiff_var_contact)

% -----------------------------------------------------------
% EXTRACT ANALYSIS INPUT
alf = anls.contact.alf; %Penalty parameter
search_par = anls.contact.search_par; %Normal point projection parameters
search_start = search_par{1}; %Starting point for normal projection point
                              %search.
K = stiffness.K; %Stiffness matrix from before
Fi_active = stiffness.Fi_active; %Internal forces fom before
u_active_prev = add_stiff_var.u_active_prev; %Displacements prev load step

```

```
28 % Extract info slave part:
29 part1 = anls.parts(add_stiff_var_contact.ms_pairs(1)); %Slave part
30 nel1 = length(part1.els);  %Number of elemnts in the patch
31
32 % Extract infor master part:
33 part2 = anls.parts(add_stiff_var_contact.ms_pairs(2)); %Master
       part
34 nel2 = length(part2.els);
35 spline_type2 = part2.spltyp;
36
37 % Calculate deformed control points master elements:
38 for iel=1:nel2
39     el2 = part2.els(iel);
40     [~, el2, ~] = compute_deformed_CP_elem(el2, part2.connectivity
           ,...
41         part2.ndof_cp, u_active_prev); %Assign to current element
               el2
42     part2.els(iel) = el2; %Assign to part
43
44 end
45
46 % -------------------------------------------------------
47 % DETECT CONTACT
48
49 %-------------- Loop slave elements:
50 for iel=1:nel1
51      % Retrieve information from slave element:
52     el = part1.els(iel); % Current slave element in slave element
           loop
53     integ = el.integ;   %Integration rule for the element
54     p = el.p;           %Polynomial degrees
55     J2 = el.J2;         %Jacobian
56     U = el.U{1};        %Knot vector
57     V = el.U{2};        %Knot vector
58
59     % Retrieve the displaced control point coordinates and add to
           element:
60     [xyz_CP_d, el, ~] = compute_deformed_CP_elem(el, part1.
           connectivity,...
61         part1.ndof_cp, u_active_prev); %Assign to current master
               elem el2
62     part1.els(iel) = el; %Assign to part
63
```

```
64      % Find gauss point positions in an element in u and v
            direction
65      [GP,GW] = gauss_point_weights(p,integ);
66
67  %-------------- Loop over Gauss Points in slave element el:
68      for igp = 1:length(GW)
69          %Find NURBS coordinate of gpoints:
70          ugp = ( U(p(1)+2)+U(p(1)+1) + GP(igp,1)*...
71              (U(p(1)+2)-U(p(1)+1)) )/2;
72          vgp = ( V(p(2)+2)+V(p(2)+1) + GP(igp,2)*...
73              (V(p(2)+2)-V(p(2)+1)) )/2;
74          gw = GW(igp);
75
76          %Find xyz coordinate of gp:
77          [XYZ_s] = get_point_coord([ugp,vgp],part1.patch_def);
78
79          % Calculate normal projection point on master from [XYZ_s
                ]:
80          [u2,v2,~] = point_project_surf(part2.patch_def,
                search_start,...
81              XYZ_s,search_par);
82          if (spline_type2==0) %NURBS
83              % Fin master element number related to projected point
                    :
84              iel2 = get_point_element([u2,v2],part2);
85          else
86              disp('Not NURBS! Not yet implemented in code')
87          end
88          el2 = part2.els(iel2);  %Assign to master part
89
90          % Calculate base functions at projection point:
91          [BF2] = compute_BF_patchbased([u2,v2],part2.patch_def,2);
92          dRduv2 = BF2(2:3,:)';
93
94          % Deformed CP of el2:
95          xyz_CP_d2 = reshape(el2.CP(:,:,1:3),[el2.ncp_e,3]);
96
97          % Basis vector 1, 2 and 3=jacobian for master elemente:
98          [g_2 ,g3_2,lg3_2] = get_base_func(dRduv2,xyz_CP_d2);
99
100         % Normal vector n (normalized) at projection point:
101         n = [0; 0; 0];
102         n(1) = g3_2(1)/lg3_2;
```

```
103         n(2) = g3_2(2)/lg3_2;
104         n(3) = g3_2(3)/lg3_2; %normalization of g3 by components
105
106
107         % Calculate normal gap function:
108         [XYZ_m] = get_point_coord([u2,v2],part2.patch_def);
109         n = n*add_stiff_var_contact.n_sign; %Correct sign of
                normal
110         gn = (XYZ_s - XYZ_m)*n; %Calculate normal gap
111
112        % Check for contact between Gauss point and projection
              point:
113         if gn<0
114 % ---------------------------------------------------------
115 % IF CONTACT: CALCULATE CONTACT CONTRIBUTIONS
116             % Calculate base functions slave element:
117             [BF1] = compute_BF_patchbased([ugp,vgp],part1.
                    patch_def,1);
118             R1 = BF1(1,:)';          % Base functions (not
                    derivatives)
119             N1 = reshape(n*R1',3*length(R1),1);
120             % Derivatives of bf w.r.t. u and v in
121             % the considered GP (two columns [dN/du, dN/dv]):
122             dRduv = BF1(2:3,:)';
123
124             % Calculate Jacobian, slave element:
125             [~,~,J1] = get_base_func(dRduv,xyz_CP_d);
126
127             % Calculate base functions slave element:
128             R2 = BF2(1,:)';
129             N2 = reshape(n*R2',length(n)*length(R2),1);
130             ddRduv2(:,1) = BF2(4,:);
131             ddRduv2(:,2) = BF2(6,:);
132             ddRduv2(:,3) = BF2(5,:);
133
134             % Coupled contact element base functions:
135             N = zeros(length(N1)+length(N2),1); %Vertical vector
136             N( 1 : length(N1) ) = N1;
137             N( (length(N1)+1) : (length(N1)+length(N2)) ) = -N2;
138
139
140 %-------------- Geometric contact stiffness, ke_geo, calculation:
141
```

```
142             %**Calculate metrics on master element**
143
144             % Covariant base vectors g:
145             tau_cov = g_2;                  %Cov vectors = g = [g1
                    g2],
146                                             %param dir 1 and 2 on
                                                    surface
147             tau_cov_dir1 = tau_cov(:,1);
148             tau_cov_dir2 = tau_cov(:,2);
149
150             % Hessian of the surface (based on 2nd derivatives):
151             h(:,1)=(ddRduv2(:,1)'*xyz_CP_d2)'; % column vector, as
                    all
152             h(:,2)=(ddRduv2(:,2)'*xyz_CP_d2)';
153             h(:,3)=(ddRduv2(:,3)'*xyz_CP_d2)';
154
155             % Covariant metric tensor gab as a vector
156             %(gab33=1, the others in 3rd line/column are null):
157             gab = [0;0;0];
158             gab(1) = g_2(1,1)*g_2(1,1) + g_2(2,1)*g_2(2,1) + ...
159                 g_2(3,1)*g_2(3,1);
160             gab(2) = g_2(1,2)*g_2(1,2) + g_2(2,2)*g_2(2,2) + ...
161                 g_2(3,2)*g_2(3,2);
162             gab(3) = g_2(1,1)*g_2(1,2) + g_2(2,1)*g_2(2,2) + ...
163                 g_2(3,1)*g_2(3,2);
164
165             m_cov = [gab(1), gab(3);
166                      gab(3), gab(2)];
167
168             % Curvature coefficients (second fund. form) as vector
169             %(bv33=1, the others in 3rd line/column are null)
170             bv=[0;0;0];
171             bv(1) = h(1,1)*n(1) + h(2,1)*n(2) + h(3,1)*n(3);
172             bv(2) = h(1,2)*n(1) + h(2,2)*n(2) + h(3,2)*n(3);
173             bv(3) = h(1,3)*n(1) + h(2,3)*n(2) + h(3,3)*n(3);
174
175             curvature_cov = [bv(1), bv(3);
176                              bv(3), bv(2)]; %the components are:
177                                             %[bv11, bv12; bv12,
                                                    bv22];
178
179             % Contravariant metric tensor:
180             invdetgab = 1/(gab(1)*gab(2)-gab(3)*gab(3));
```

```
181
182             gab_con11 =  invdetgab*gab(2);
183             gab_con12 = -invdetgab*gab(3);
184             gab_con22 =  invdetgab*gab(1);
185
186             m_con = [gab_con11, gab_con12;
187                      gab_con12, gab_con22];
188
189
190             %**k_geo TERMS**
191             %From article (De Lorenzis et.al, 2014), see top.
192
193             % --Inverse of Eq. (18)--
194             A_cov = m_cov-gn*curvature_cov;
195             % Inverse of determinant of covarian curvature bv:
196             invdetA = 1/(A_cov(1,1)*A_cov(2,2)-A_cov(1,2)*A_cov
                    (2,1));
197             A_con11 =  invdetA*A_cov(2,2);
198             A_con12 = -invdetA*A_cov(1,2);
199             A_con21 = -invdetA*A_cov(2,1);
200             A_con22 =  invdetA*A_cov(1,1);
201
202             A_con = [A_con11, A_con12;
203                      A_con21, A_con22];
204
205             % --Eq. (25), "N_alpha" and "T_alpha"--
206             dRduv2_dir1 = dRduv2(:,1);
207             dRduv2_dir2 = dRduv2(:,2);
208
209             % Multiply all terms in dRduv2_dir1 with n and gather
                    in vector
210             % N_local_dir_m
211             N_local_dir1 = zeros(length(N1)+length(N2),1);
212             N_local_dir1_m = reshape(n*dRduv2_dir1',length(n)*...
213                 length(dRduv2_dir1),1);   % Master components
214             % Add to global vector and change sign
215             N_local_dir1((length(N1)+1):(length(N2)+length(N1)))
                    ...
216                 = - N_local_dir1_m;
217
218             % Same procedure direction 2:
219             N_local_dir2 = zeros(length(N1)+length(N2),1);
220             N_local_dir2_m = reshape(n*dRduv2_dir2',length(n)*...
```

```
221                 length(dRduv2_dir2),1);    % Master components
222             N_local_dir2((length(N1)+1):(length(N2)+length(N1)))
                    ...
223                 = - N_local_dir2_m;
224
225             % Multiply all terms in R2 and R1 with tau_cov_dir1,
226             % gather in separate vectors for R2 and R1 (master and
                     slave)
227             T_local_dir1 = zeros(length(N1)+length(N2),1);
228             T_local_dir1_m = reshape(tau_cov_dir1*R2',...
229                 length(tau_cov_dir1)*length(R2),1);    % Master
                        components
230             T_local_dir1_s = reshape(tau_cov_dir1*R1',...
231                 length(tau_cov_dir1)*length(R1),1);    % Slave
                        components
232             % Add to global vector and change sign master part:
233             %Top part of vector = slave components
234             T_local_dir1( 1:length(N1) ) = T_local_dir1_s;
235             %Last part of vector = master components
236             T_local_dir1( (length(N1)+1):(length(N1)+length(N2)) )
                    ...
237                 = - T_local_dir1_m;
238
239
240             % Same procedure direction 2:
241             T_local_dir2 = zeros(length(N1)+length(N2),1);
242             T_local_dir2_m = reshape(tau_cov_dir2*R2',...
243                 length(tau_cov_dir2)*length(R2),1);    % Master
                        components
244             T_local_dir2_s = reshape(tau_cov_dir2*R1',...
245                 length(tau_cov_dir2)*length(R1),1);    % Slave
                        components
246             T_local_dir2( 1:length(N1) ) = T_local_dir2_s;
247             T_local_dir2( (length(N1)+1):(length(N1)+length(N2)) )
                     ...
248                 = - T_local_dir2_m;  %Master components
249
250             % ---Eq. (26): Gather both directions into vectors---
251             N_hat = [N_local_dir1, N_local_dir2];
252             T_hat = [T_local_dir1, T_local_dir2];
253
254             % ---Eq. (27)---
255             D = (T_hat - gn*N_hat)*A_con;
```

```
256             N_bar = N_hat - D*curvature_cov;
257
258
259             % ---Eq. (24)---
260             kea_geo = gn*N_bar*m_con*N_bar' + D*N_hat' + N_hat*D'
                    - ...
261                 D*curvature_cov*D';
262
263             % ---Eq. (34): Calculate ke_geo---
264             ke_geo = alf*gn*kea_geo*gw*J2*J1;
265
266 %-------------- Final calculation of contact element contributions
    :
267             % Stiffness ke_c:
268             kea_c = alf*(N*N');
269             ke_c = kea_c*gw*J2*J1;
270
271             % Total stiffness:
272             ke_gp = ke_c + ke_geo;
273
274             % Force contribution:
275             fiea_c = alf*N*gn;
276             fie_c = fiea_c*gw*J2*J1;
277
278 %-------------- Assembly:
279             % Assemble into global K:
280             K = Kassembly_contact_fourLoops(el,el2,K,ke_gp);
281
282             %Assemble into global F:
283             for i = 1:el.ndof_e
284                 if (el.LM(i)~=0)
285                 globi = el.LM(i);
286                 Fi_active(globi) = Fi_active(globi) + fie_c(i);
287                 end
288             end
289
290             for i = 1:el2.ndof_e
291                 if (el2.LM(i)~=0)
292                 globi = el2.LM(i);
293                 Fi_active(globi) = Fi_active(globi) + fie_c(i+el.
                        ndof_e);
294                 end
295             end
```

```
296         end % Contact condition: if gn < 0
297     end % Loop Gauss points
298 end % Loop slave elements
299 
300 % ------------------------------------------------------
301 % OUTPUT / RESULTS
302 stiffness.K = K;
303 stiffness.Fi_active = Fi_active;
304 
305 end %function
```

C.3 Contact contribution function: GPTS, penalty and 2 step point search

```
1  %-------------CONTACT CONTRIBUTIONS CALC. FUNCTION----
2  % ****DESCRIPTION****
3  % Calculates the contact contribution to the stiffness matrix and
       force
4  % residual. Gauss-point-to-segment discretization with the penalty
        method.
5  % includes geometric contact stiffness.
6  
7  % Created by Embla L. Holten , 2019.
8  
9  % 2 Step point Search:
10 % First step:
11 %  Only elements that have nonzero entries in M_contact_partners
       matrix
12 %  are included in the contact contribution calculations.
13 % Second step:
14 %  Normal projection point search function is called, using the
       statring
15 %  point from M_contact_partners.
16 
17 
18 
19 
20 %-------------------------------------------------------
21 
22 function [stiffness, anls] =
```

```
    stiff_mat_Contact_GPTS_Pen_2StepSearch...
23      (anls,stiffness,add_stiff_var,add_stiff_var_contact)
24
25
26  % ------------------------------------------------------
27  % EXTRACT ANALYSIS INPUT
28  alf = anls.contact.alf; %Penalty parameter
29  search_par = anls.contact.search_par; %Normal point projection
        parameters
30  K = stiffness.K; %Stiffness matrix from before
31  Fi_active = stiffness.Fi_active; %Internal forces fom before
32  u_active_prev = add_stiff_var.u_active_prev; %Displacements prev
        load step
33
34  % Extract info slave part:
35  part1 = anls.parts(add_stiff_var_contact.ms_pairs(1)); %Slave part
36  nel1 = length(part1.els);  %Number of elemnts in the patch
37
38  % Extract infor master part:
39  part2 = anls.parts(add_stiff_var_contact.ms_pairs(2)); %Master
        part
40  nel2 = length(part2.els);
41  spline_type2 = part2.spltyp;
42
43  % Calculate deformed control points master elements:
44  for iel=1:nel2
45      el2 = part2.els(iel);
46      [~, el2, ~] = compute_deformed_CP_elem(el2, part2.connectivity
            ,...
47          part2.ndof_cp, u_active_prev); %Assign to current element
                el2
48      part2.els(iel) = el2; %Assign to part
49
50  end
51
52  % ------------------------------------------------------
53  % DETECT CONTACT
54
55  %-------------- Loop slave elements:
56  for iel=1:nel1
57      % 2 step point search check: If element has a contact partner
58      if add_stiff_var_contact.M_contact_partners(iel,3) == 0
59          continue % Continue to next slave element if no contact
```

```
            partner
60      end
61      % Retrieve starting point for normal projection point search:
62      search_start = [add_stiff_var_contact.M_contact_partners(iel
            ,1),...
63          add_stiff_var_contact.M_contact_partners(iel,2)];
64
65      % Retrieve information from slave element:
66      el = part1.els(iel); % Current slave element in slave element
            loop
67      integ = el.integ;    %Integration rule for the element
68      p = el.p;            %Polynomial degrees
69      J2 = el.J2;          %Jacobian
70      U = el.U{1};         %Knot vector
71      V = el.U{2};         %Knot vector
72
73      % Retrieve the displaced control point coordinates and add to
            element:
74      [xyz_CP_d, el, ~] = compute_deformed_CP_elem(el, part1.
            connectivity,...
75          part1.ndof_cp, u_active_prev); %Assign to current master
                elem el2
76      part1.els(iel) = el; %Assign to part
77
78      % Find gauss point positions in an element in u and v
            direction
79      [GP,GW] = gauss_point_weights(p,integ);
80
81  %--------------- Loop over Gauss Points in slave element el:
82      for igp = 1:length(GW)
83          %Find NURBS coordinate of gpoints:
84          ugp = ( U(p(1)+2)+U(p(1)+1) + GP(igp,1)*...
85              (U(p(1)+2)-U(p(1)+1)) )/2;
86          vgp = ( V(p(2)+2)+V(p(2)+1) + GP(igp,2)*...
87              (V(p(2)+2)-V(p(2)+1)) )/2;
88          gw = GW(igp);
89
90          %Find xyz coordinate of gp:
91          [XYZ_s] = get_point_coord([ugp,vgp],part1.patch_def);
92
93          % Calculate normal projection point on master from [XYZ_s
                ]:
94          [u2,v2,~] = point_project_surf(part2.patch_def,
```

```
            search_start,...
95              XYZ_s,search_par);
96          if (spline_type2==0) %NURBS
97              % Fin master element number related to projected point
                    :
98              iel2 = get_point_element([u2,v2],part2);
99          else
100             disp('Not NURBS! Not yet implemented in code')
101         end
102         el2 = part2.els(iel2);  %Assign to master part
103
104         % Calculate base functions at projection point:
105         [BF2] = compute_BF_patchbased([u2,v2],part2.patch_def,2);
106         dRduv2 = BF2(2:3,:)';
107
108         % Deformed CP of el2:
109         xyz_CP_d2 = reshape(el2.CP(:,:,1:3),[el2.ncp_e,3]);
110
111         % Basis vector 1, 2 and 3=jacobian for master elemente:
112         [g_2 ,g3_2,lg3_2] = get_base_func(dRduv2,xyz_CP_d2);
113
114         % Normal vector n (normalized) at projection point:
115         n = [0; 0; 0];
116         n(1) = g3_2(1)/lg3_2;
117         n(2) = g3_2(2)/lg3_2;
118         n(3) = g3_2(3)/lg3_2; %normalization of g3 by components
119
120
121         % Calculate normal gap function:
122         [XYZ_m] = get_point_coord([u2,v2],part2.patch_def);
123         n = n*add_stiff_var_contact.n_sign; %Correct sign of
                normal
124         gn = (XYZ_s - XYZ_m)*n; %Calculate normal gap
125
126        % Check for contact between Gauss point and projection
              point:
127         if gn<0
128 % -------------------------------------------------------
129 % IF CONTACT: CALCULATE CONTACT CONTRIBUTIONS
130
131             % Calculate base functions slave element:
132             [BF1] = compute_BF_patchbased([ugp,vgp],part1.
                    patch_def,1);
```

```
133             R1 = BF1(1,:)';          % Base functions (not
                    derivatives)
134             N1 = reshape(n*R1',3*length(R1),1);
135             % Derivatives of bf w.r.t. u and v in
136             % the considered GP (two columns [dN/du, dN/dv]):
137             dRduv = BF1(2:3,:)';
138
139             % Calculate Jacobian, slave element:
140             [~,~,J1] = get_base_func(dRduv,xyz_CP_d);
141
142             % Calculate base functions slave element:
143             R2 = BF2(1,:)';
144             N2 = reshape(n*R2',length(n)*length(R2),1);
145             ddRduv2(:,1) = BF2(4,:);
146             ddRduv2(:,2) = BF2(6,:);
147             ddRduv2(:,3) = BF2(5,:);
148
149             % Coupled contact element base functions:
150             N = zeros(length(N1)+length(N2),1); %Vertical vector
151             N( 1 : length(N1) ) = N1;
152             N( (length(N1)+1) : (length(N1)+length(N2)) ) = -N2;
153
154
155 %--------------- Geometric contact stiffness, ke_geo, calculation:
156
157             %**Calculate metrics on master element**
158
159             % Covariant base vectors g:
160             tau_cov = g_2;                  %Cov vectors = g = [g1
                     g2],
161                                             %param dir 1 and 2 on
                                                    surface
162             tau_cov_dir1 = tau_cov(:,1);
163             tau_cov_dir2 = tau_cov(:,2);
164
165             % Hessian of the surface (based on 2nd derivatives):
166             h(:,1)=(ddRduv2(:,1)'*xyz_CP_d2)'; % column vector, as
                     all
167             h(:,2)=(ddRduv2(:,2)'*xyz_CP_d2)';
168             h(:,3)=(ddRduv2(:,3)'*xyz_CP_d2)';
169
170             % Covariant metric tensor gab as a vector
171             %(gab33=1, the others in 3rd line/column are null):
```

```
172             gab = [0;0;0];
173             gab(1) = g_2(1,1)*g_2(1,1) + g_2(2,1)*g_2(2,1) + ...
174                 g_2(3,1)*g_2(3,1);
175             gab(2) = g_2(1,2)*g_2(1,2) + g_2(2,2)*g_2(2,2) + ...
176                 g_2(3,2)*g_2(3,2);
177             gab(3) = g_2(1,1)*g_2(1,2) + g_2(2,1)*g_2(2,2) + ...
178                 g_2(3,1)*g_2(3,2);
179
180             m_cov = [gab(1), gab(3);
181                      gab(3), gab(2)];
182
183             % Curvature coefficients (second fund. form) as vector
184             %(bv33=1, the others in 3rd line/column are null)
185             bv=[0;0;0];
186             bv(1) = h(1,1)*n(1) + h(2,1)*n(2) + h(3,1)*n(3);
187             bv(2) = h(1,2)*n(1) + h(2,2)*n(2) + h(3,2)*n(3);
188             bv(3) = h(1,3)*n(1) + h(2,3)*n(2) + h(3,3)*n(3);
189
190             curvature_cov = [bv(1), bv(3);
191                              bv(3), bv(2)]; %the components are:
192                                             %[bv11, bv12; bv12,
                                                    bv22];
193
194             % Contravariant metric tensor:
195             invdetgab = 1/(gab(1)*gab(2)-gab(3)*gab(3));
196
197             gab_con11 =  invdetgab*gab(2);
198             gab_con12 = -invdetgab*gab(3);
199             gab_con22 =  invdetgab*gab(1);
200
201             m_con = [gab_con11, gab_con12;
202                      gab_con12, gab_con22];
203
204
205             %**k_geo TERMS**
206             %From article (De Lorenzis et.al, 2014), see top.
207
208             % --Inverse of Eq. (18)--
209             A_cov = m_cov-gn*curvature_cov;
210             % Inverse of determinant of covarian curvature bv:
211             invdetA = 1/(A_cov(1,1)*A_cov(2,2)-A_cov(1,2)*A_cov
                    (2,1));
212             A_con11 =  invdetA*A_cov(2,2);
```

```
213             A_con12 = -invdetA*A_cov(1,2);
214             A_con21 = -invdetA*A_cov(2,1);
215             A_con22 =  invdetA*A_cov(1,1);
216
217             A_con = [A_con11, A_con12;
218                      A_con21, A_con22];
219
220             % --Eq. (25), "N_alpha" and "T_alpha"--
221             dRduv2_dir1 = dRduv2(:,1);
222             dRduv2_dir2 = dRduv2(:,2);
223
224             % Multiply all terms in dRduv2_dir1 with n and gather
                    in vector
225             % N_local_dir_m
226             N_local_dir1 = zeros(length(N1)+length(N2),1);
227             N_local_dir1_m = reshape(n*dRduv2_dir1',length(n)*...
228                 length(dRduv2_dir1),1);   % Master components
229             % Add to global vector and change sign
230             N_local_dir1((length(N1)+1):(length(N2)+length(N1)))
                    ...
231                 = - N_local_dir1_m;
232
233             % Same procedure direction 2:
234             N_local_dir2 = zeros(length(N1)+length(N2),1);
235             N_local_dir2_m = reshape(n*dRduv2_dir2',length(n)*...
236                 length(dRduv2_dir2),1);   % Master components
237             N_local_dir2((length(N1)+1):(length(N2)+length(N1)))
                    ...
238                 = - N_local_dir2_m;
239
240             % Multiply all terms in R2 and R1 with tau_cov_dir1,
241             % gather in separate vectors for R2 and R1 (master and
                     slave)
242             T_local_dir1 = zeros(length(N1)+length(N2),1);
243             T_local_dir1_m = reshape(tau_cov_dir1*R2',...
244                 length(tau_cov_dir1)*length(R2),1);   % Master
                        components
245             T_local_dir1_s = reshape(tau_cov_dir1*R1',...
246                 length(tau_cov_dir1)*length(R1),1);   % Slave
                        components
247             % Add to global vector and change sign master part:
248             %Top part of vector = slave components
249             T_local_dir1( 1:length(N1) ) = T_local_dir1_s;
```

```
250             %Last part of vector = master components
251             T_local_dir1( (length(N1)+1):(length(N1)+length(N2)) )
                    ...
252                 = - T_local_dir1_m;
253
254
255             % Same procedure direction 2:
256             T_local_dir2 = zeros(length(N1)+length(N2),1);
257             T_local_dir2_m = reshape(tau_cov_dir2*R2',...
258                 length(tau_cov_dir2)*length(R2),1);   % Master
                        components
259             T_local_dir2_s = reshape(tau_cov_dir2*R1',...
260                 length(tau_cov_dir2)*length(R1),1);   % Slave
                        components
261             T_local_dir2( 1:length(N1) ) = T_local_dir2_s;
262             T_local_dir2( (length(N1)+1):(length(N1)+length(N2)) )
                     ...
263                 = - T_local_dir2_m;  %Master components
264
265             % ---Eq. (26): Gather both directions into vectors---
266             N_hat = [N_local_dir1, N_local_dir2];
267             T_hat = [T_local_dir1, T_local_dir2];
268
269             % ---Eq. (27)---
270             D = (T_hat - gn*N_hat)*A_con;
271             N_bar = N_hat - D*curvature_cov;
272
273
274             % ---Eq. (24)---
275             kea_geo = gn*N_bar*m_con*N_bar' + D*N_hat' + N_hat*D'
                    - ...
276                 D*curvature_cov*D';
277
278             % ---Eq. (34): Calculate ke_geo---
279             ke_geo = alf*gn*kea_geo*gw*J2*J1;
280
281 %--------------- Final calculation of contact element contributions
        :
282             % Stiffness ke_c:
283             kea_c = alf*(N*N');
284             ke_c = kea_c*gw*J2*J1;
285
286             % Total stiffness:
```

```
287             ke_gp = ke_c + ke_geo;
288
289             % Force contribution:
290             fiea_c = alf*N*gn;
291             fie_c = fiea_c*gw*J2*J1;
292
293 %--------------- Assembly:
294             % Assemble into global K:
295             K = Kassembly_contact_fourLoops(el,el2,K,ke_gp);
296
297             %Assemble into global F:
298             for i = 1:el.ndof_e
299                 if (el.LM(i)~=0)
300                 globi = el.LM(i);
301                 Fi_active(globi) = Fi_active(globi) + fie_c(i);
302                 end
303             end
304
305             for i = 1:el2.ndof_e
306                 if (el2.LM(i)~=0)
307                 globi = el2.LM(i);
308                 Fi_active(globi) = Fi_active(globi) + fie_c(i+el.
                        ndof_e);
309                 end
310             end
311         end % Contact condition: if gn < 0
312     end % Loop Gauss points
313 end % Loop slave elements
314
315 % ---------------------------------------------------------
316 % OUTPUT / RESULTS
317 stiffness.K = K;
318 stiffness.Fi_active = Fi_active;
319
320 end %function
```

C.4 Contact contribution function: Cylinder Sqeeze

```
1 %-------------------CYLINDER SQUEEZE STIFF MAT-----------
2 % ****DESCRIPTION****
3 % Calculates the contact contribution to the stiffness matrix and
```

```
    force
% residual. Gauss-point-to-segment discretization with the penalty
    method.
% This function is specialized for the cylinder example.

% Created by Embla L. Holten , 2019.



%-----------------------------------------------------------

function [stiffness, anls] = stiff_mat_Contact_Cylinder_Squeeze...
    (anls,stiffness,add_stiff_var,add_stiff_var_contact)




% -----------------------------------------------------------
% EXTRACT ANALYSIS INPUT
alf = anls.contact.alf; %Penalty parameter
search_par = anls.contact.search_par; %Normal point projection
    parameters
K = stiffness.K; %Stiffness matrix from before
Fi_active = stiffness.Fi_active; %Internal forces fom before
u_active_prev = add_stiff_var.u_active_prev; %Displacements prev
    load step

% Extract info slave part:
part1 = anls.parts(add_stiff_var_contact.ms_pairs(1)); %Slave part
nel1 = length(part1.els);  %Number of elemnts in the patch

% Extract infor master part:
part2 = anls.parts(add_stiff_var_contact.ms_pairs(2)); %Master
    part
nel2 = length(part2.els);
spline_type2 = part2.spltyp;

% Calculate deformed control points master elements:
for iel=1:nel2
    el2 = part2.els(iel);
    [~, el2, ~] = compute_deformed_CP_elem(el2, part2.connectivity
        ,...
        part2.ndof_cp, u_active_prev); %Assign to current element
```

```
            el2
41      part2.els(iel) = el2; %Assign to part
42
43  end
44
45  % ------------------------------------------------------
46  % DETECT CONTACT
47
48  %-------------- Loop slave elements:
49  for iel=1:nel1
50      % 2 step point search check: If element has a contact partner
51      if add_stiff_var_contact.M_contact_partners(iel,3) == 0
52          continue % Continue to next slave element if no contact
                partner
53      end
54      % Retrieve starting point for normal projection point search:
55      search_start = [add_stiff_var_contact.M_contact_partners(iel
            ,1),...
56          add_stiff_var_contact.M_contact_partners(iel,2)];
57
58      % Retrieve information from slave element:
59      el = part1.els(iel); % Current slave element in slave element
            loop
60      integ = el.integ;   %Integration rule for the element
61      p = el.p;           %Polynomial degrees
62      J2 = el.J2;         %Jacobian
63      U = el.U{1};        %Knot vector
64      V = el.U{2};        %Knot vector
65
66      % Retrieve the displaced control point coordinates and add to
            element:
67      [xyz_CP_d, el, ~] = compute_deformed_CP_elem(el, part1.
            connectivity,...
68          part1.ndof_cp, u_active_prev); %Assign to current master
                elem el2
69      part1.els(iel) = el; %Assign to part
70
71      % Find gauss point positions in an element in u and v
            direction
72      [GP,GW] = gauss_point_weights(p,integ);
73
74  %-------------- Loop over Gauss Points in slave element el:
75      for igp = 1:length(GW)
```

```
76          %Find NURBS coordinate of gpoints:
77          ugp = ( U(p(1)+2)+U(p(1)+1) + GP(igp,1)*...
78              (U(p(1)+2)-U(p(1)+1)) )/2;
79          vgp = ( V(p(2)+2)+V(p(2)+1) + GP(igp,2)*...
80              (V(p(2)+2)-V(p(2)+1)) )/2;
81          gw = GW(igp);
82
83          %Find xyz coordinate of gp:
84          [XYZ_s] = get_point_coord([ugp,vgp],part1.patch_def);
85
86          % Calculate normal projection point on master from [XYZ_s
                ]:
87          [u2,v2,~] = point_project_surf(part2.patch_def,
                search_start,...
88              XYZ_s,search_par);
89          if (spline_type2==0) %NURBS
90              % Fin master element number related to projected point
                    :
91              iel2 = get_point_element([u2,v2],part2);
92          else
93              disp('Not NURBS! Not yet implemented in code')
94          end
95          el2 = part2.els(iel2);  %Assign to master part
96
97
98          % Calculate base functions at projection point:
99          [BF2] = compute_BF_patchbased([u2,v2],part2.patch_def,2);
100         dRduv2 = BF2(2:3,:)';
101
102         % Deformed CP of el2:
103         xyz_CP_d2 = reshape(el2.CP(:,:,1:3),[el2.ncp_e,3]);
104
105         % Basis vector 1, 2 and 3=jacobian for master elemente:
106         [g_2 ,g3_2,lg3_2] = get_base_func(dRduv2,xyz_CP_d2);
107
108         % Normal vector n (normalized) at projection point:
109         n = [0; 0; 0];
110         n(1) = g3_2(1)/lg3_2;
111         n(2) = g3_2(2)/lg3_2;
112         n(3) = g3_2(3)/lg3_2; %normalization of g3 by components
113
114
115         % Calculate normal gap function:
```

```
116         [XYZ_m] = get_point_coord([u2,v2],part2.patch_def);
117         n = n*add_stiff_var_contact.n_sign; %Correct sign of
                normal
118         gn = (XYZ_s - XYZ_m)*n; %Calculate normal gap
119         gn = gn-part1.thick/2; %Subtract half shell thickness
120
121         % Check for contact between Gauss point and projection
                point:
122         if ( gn<0 && abs(XYZ_s(3) - XYZ_m(3))<anls.contact.tol_z
                && ...
123                 abs(XYZ_s(2) - XYZ_m(2))<anls.contact.tol_y && ...
124                 abs(XYZ_s(1) - XYZ_m(1))<anls.contact.tol_x )
125
126 % -------------------------------------------------------
127 % CALCULATE CONTACT CONTRIBUTIONS
128
129             % Calculate base functions slave element:
130             [BF1] = compute_BF_patchbased([ugp,vgp],part1.
                    patch_def,1);
131             R1 = BF1(1,:)';          % Base functions (not
                    derivatives)
132             N1 = reshape(n*R1',3*length(R1),1);
133             % Derivatives of bf w.r.t. u and v in
134             % the considered GP (two columns [dN/du, dN/dv]):
135             dRduv = BF1(2:3,:)';
136
137             % Calculate Jacobian, slave element:
138             [~,~,J1] = get_base_func(dRduv,xyz_CP_d);
139
140             % Calculate base functions slave element:
141             R2 = BF2(1,:)';
142             N2 = reshape(n*R2',length(n)*length(R2),1);
143             ddRduv2(:,1) = BF2(4,:);
144             ddRduv2(:,2) = BF2(6,:);
145             ddRduv2(:,3) = BF2(5,:);
146
147             % Coupled contact element base functions:
148             N = zeros(length(N1)+length(N2),1); %Vertical vector
149             N( 1 : length(N1) ) = N1;
150             N( (length(N1)+1) : (length(N1)+length(N2)) ) = -N2;
151
152
153 %-------------- Geometric contact stiffness, ke_geo, calculation:
```

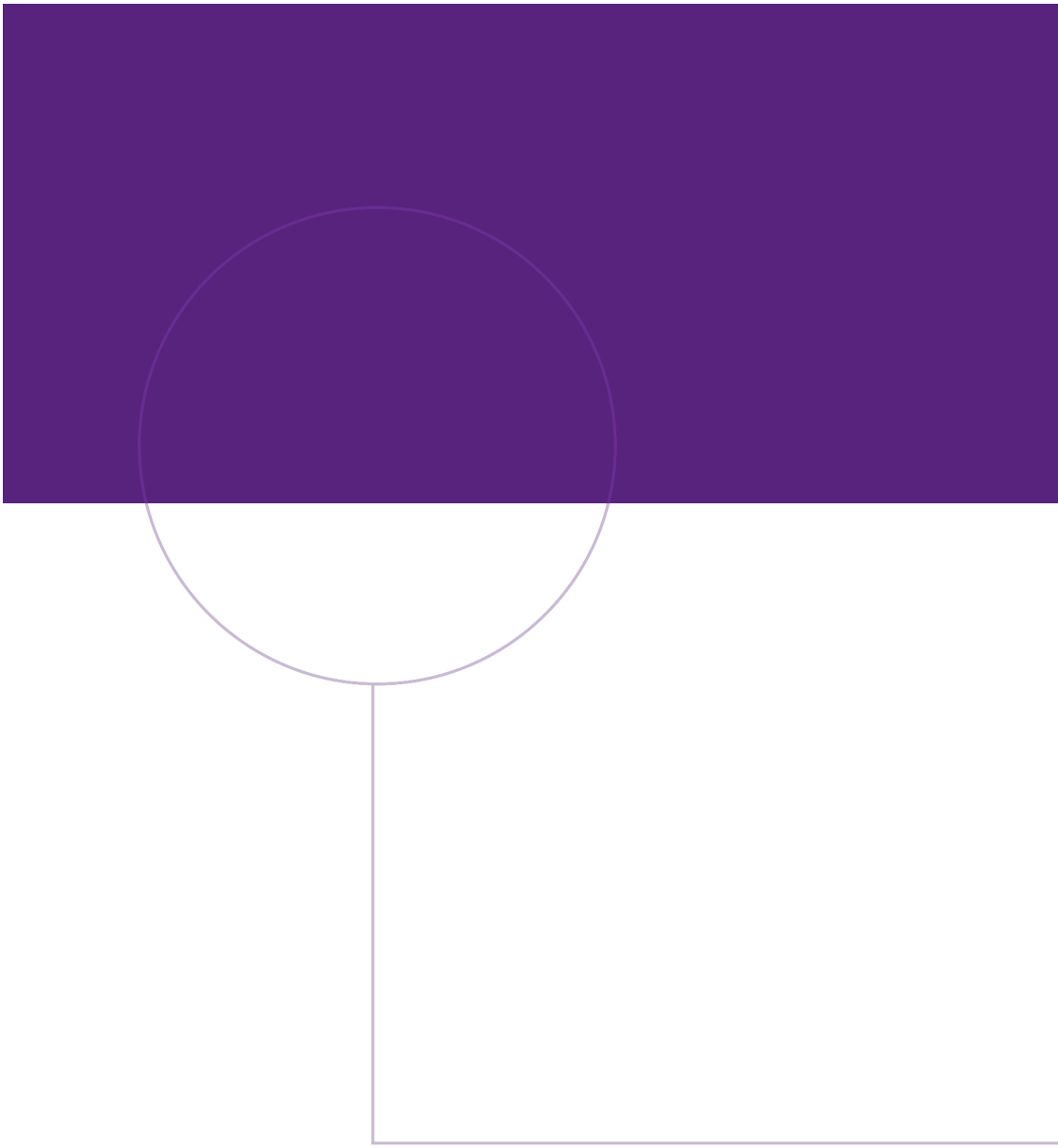
```
154
155             %**Calculate metrics on master element**
156
157             % Covariant base vectors g:
158             tau_cov = g_2;                  %Cov vectors = g = [g1
                     g2],
159                                             %param dir 1 and 2 on
                        surface
160             tau_cov_dir1 = tau_cov(:,1);
161             tau_cov_dir2 = tau_cov(:,2);
162
163             % Hessian of the surface (based on 2nd derivatives):
164             h(:,1)=(ddRduv2(:,1)'*xyz_CP_d2)'; % column vector, as
                     all
165             h(:,2)=(ddRduv2(:,2)'*xyz_CP_d2)';
166             h(:,3)=(ddRduv2(:,3)'*xyz_CP_d2)';
167
168             % Covariant metric tensor gab as a vector
169             %(gab33=1, the others in 3rd line/column are null):
170             gab = [0;0;0];
171             gab(1) = g_2(1,1)*g_2(1,1) + g_2(2,1)*g_2(2,1) + ...
172                 g_2(3,1)*g_2(3,1);
173             gab(2) = g_2(1,2)*g_2(1,2) + g_2(2,2)*g_2(2,2) + ...
174                 g_2(3,2)*g_2(3,2);
175             gab(3) = g_2(1,1)*g_2(1,2) + g_2(2,1)*g_2(2,2) + ...
176                 g_2(3,1)*g_2(3,2);
177
178             m_cov = [gab(1), gab(3);
179                      gab(3), gab(2)];
180
181             % Curvature coefficients (second fund. form) as vector
182             %(bv33=1, the others in 3rd line/column are null)
183             bv=[0;0;0];
184             bv(1) = h(1,1)*n(1) + h(2,1)*n(2) + h(3,1)*n(3);
185             bv(2) = h(1,2)*n(1) + h(2,2)*n(2) + h(3,2)*n(3);
186             bv(3) = h(1,3)*n(1) + h(2,3)*n(2) + h(3,3)*n(3);
187
188             curvature_cov = [bv(1), bv(3);
189                              bv(3), bv(2)]; %the components are:
190                                             %[bv11, bv12; bv12,
                                                bv22];
191
192             % Contravariant metric tensor:
```

```
193             invdetgab = 1/(gab(1)*gab(2)-gab(3)*gab(3));
194
195             gab_con11 =  invdetgab*gab(2);
196             gab_con12 = -invdetgab*gab(3);
197             gab_con22 =  invdetgab*gab(1);
198
199             m_con = [gab_con11, gab_con12;
200                      gab_con12, gab_con22];
201
202
203             %**k_geo TERMS**
204             %From article (De Lorenzis et.al, 2014), see top.
205
206             % --Inverse of Eq. (18)--
207             A_cov = m_cov-gn*curvature_cov;
208             % Inverse of determinant of covarian curvature bv:
209             invdetA = 1/(A_cov(1,1)*A_cov(2,2)-A_cov(1,2)*A_cov
                    (2,1));
210             A_con11 =  invdetA*A_cov(2,2);
211             A_con12 = -invdetA*A_cov(1,2);
212             A_con21 = -invdetA*A_cov(2,1);
213             A_con22 =  invdetA*A_cov(1,1);
214
215             A_con = [A_con11, A_con12;
216                      A_con21, A_con22];
217
218             % --Eq. (25), "N_alpha" and "T_alpha"--
219             dRduv2_dir1 = dRduv2(:,1);
220             dRduv2_dir2 = dRduv2(:,2);
221
222             % Multiply all terms in dRduv2_dir1 with n and gather
                    in vector
223             % N_local_dir_m
224             N_local_dir1 = zeros(length(N1)+length(N2),1);
225             N_local_dir1_m = reshape(n*dRduv2_dir1',length(n)*...
226                 length(dRduv2_dir1),1);   % Master components
227             % Add to global vector and change sign
228             N_local_dir1((length(N1)+1):(length(N2)+length(N1)))
                    ...
229                 = - N_local_dir1_m;
230
231             % Same procedure direction 2:
232             N_local_dir2 = zeros(length(N1)+length(N2),1);
```

```
233             N_local_dir2_m = reshape(n*dRduv2_dir2',length(n)*...
234                 length(dRduv2_dir2),1);   % Master components
235             N_local_dir2((length(N1)+1):(length(N2)+length(N1)))
                    ...
236                 = - N_local_dir2_m;
237
238             % Multiply all terms in R2 and R1 with tau_cov_dir1,
239             % gather in separate vectors for R2 and R1 (master and
                    slave)
240             T_local_dir1 = zeros(length(N1)+length(N2),1);
241             T_local_dir1_m = reshape(tau_cov_dir1*R2',...
242                 length(tau_cov_dir1)*length(R2),1);   % Master
                        components
243             T_local_dir1_s = reshape(tau_cov_dir1*R1',...
244                 length(tau_cov_dir1)*length(R1),1);   % Slave
                        components
245             % Add to global vector and change sign master part:
246             %Top part of vector = slave components
247             T_local_dir1( 1:length(N1) ) = T_local_dir1_s;
248             %Last part of vector = master components
249             T_local_dir1( (length(N1)+1):(length(N1)+length(N2)) )
                    ...
250                 = - T_local_dir1_m;
251
252
253             % Same procedure direction 2:
254             T_local_dir2 = zeros(length(N1)+length(N2),1);
255             T_local_dir2_m = reshape(tau_cov_dir2*R2',...
256                 length(tau_cov_dir2)*length(R2),1);   % Master
                        components
257             T_local_dir2_s = reshape(tau_cov_dir2*R1',...
258                 length(tau_cov_dir2)*length(R1),1);   % Slave
                        components
259             T_local_dir2( 1:length(N1) ) = T_local_dir2_s;
260             T_local_dir2( (length(N1)+1):(length(N1)+length(N2)) )
                    ...
261                 = - T_local_dir2_m;  %Master components
262
263             % ---Eq. (26): Gather both directions into vectors---
264             N_hat = [N_local_dir1, N_local_dir2];
265             T_hat = [T_local_dir1, T_local_dir2];
266
267             % ---Eq. (27)---
```

```
268             D = (T_hat - gn*N_hat)*A_con;
269             N_bar = N_hat - D*curvature_cov;
270
271
272             % ---Eq. (24)---
273             kea_geo = gn*N_bar*m_con*N_bar' + D*N_hat' + N_hat*D'
                    - ...
274                 D*curvature_cov*D';
275
276             % ---Eq. (34): Calculate ke_geo---
277             ke_geo = alf*gn*kea_geo*gw*J2*J1;
278
279 %-------------- Final calculation of contact element contributions
    :
280             % Stiffness ke_c:
281             kea_c = alf*(N*N');
282             ke_c = kea_c*gw*J2*J1;
283
284             % Total stiffness:
285             ke_gp = ke_c + ke_geo;
286
287             % Force contribution:
288             fiea_c = alf*N*gn;
289             fie_c = fiea_c*gw*J2*J1;
290
291 %-------------- Assembly:
292             % Assemble into global K:
293             K = Kassembly_contact_fourLoops(el,el2,K,ke_gp);
294
295             %Assemble into global F:
296             for i = 1:el.ndof_e
297                 if (el.LM(i)~=0)
298                 globi = el.LM(i);
299                 Fi_active(globi) = Fi_active(globi) + fie_c(i);
300                 end
301             end
302
303             for i = 1:el2.ndof_e
304                 if (el2.LM(i)~=0)
305                 globi = el2.LM(i);
306                 Fi_active(globi) = Fi_active(globi) + fie_c(i+el.
                        ndof_e);
307                 end
```

```
308             end
309         end % Contact condition: if gn < 0
310     end % Loop Gauss points
311 end % Loop slave elements
312 
313 % -------------------------------------------------------
314 % OUTPUT / RESULTS
315 stiffness.K = K;
316 stiffness.Fi_active = Fi_active;
317 
318 end %function
```



NTNU
Norwegian University of
Science and Technology