Calculation of J-Integral for 3D Cracks Using an Unstructured Tetrahedral Mesh via a Domain Integral Method

A Thesis

Submitted For The Degree Of

Master of Science

In

Lightweight Structures & Impact Engineering

by

Albert Brennan

2247539@brunel.ac.uk

Department of Mechanical and Aerospace Engineering

Brunel University London

March 2025

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This section introduces the project, provides context for why it was undertaken, and presents some important background information. The necessity of crack propagation analysis within the civil aerospace sector is discussed first, followed by an explanation of the potential benefits that the project could deliver within this area. The aims & objectives of the project are then presented, followed by a summary of the methodology used, along with a methodology diagram. The final section of the chapter provides an overview of the structure of the remainder of the thesis.

1.1 Background

Modern civil aircraft must meet multiple – sometimes conflicting – requirements: they must be as light as possible in order to maximise fuel efficiency, durable enough to achieve the long service life expected by customers, and compliant with the stringent safety standards demanded by the aviation authorities. In order to design aircraft that meet these requirements, it is necessary to be able to accurately predict the fatigue lives of structural components. This requires understanding of where cracks are likely to initiate, and how quickly they will grow. Accurate fatigue life predictions provide assurance that an aircraft will be able to achieve its specified service life while maintaining an extremely low risk of fatigue failure, and minimising the amount of unnecessary mass added to the aircraft's structure. The study and analysis of crack initiation and growth within airframe structures falls within the engineering discipline of fatigue and damage tolerance analysis. Fatigue and damage tolerance analysis incorporates many distinct processes – including physical testing, manufacturing inspections, and analysis of in-service data – as shown in Figure 1.1. However, the focus of this thesis is on the process of crack propagation life calculation This process aims to predict the rate of fatigue crack growth within a structure, in order to ensure that it is inspected and retired in a timely manner.

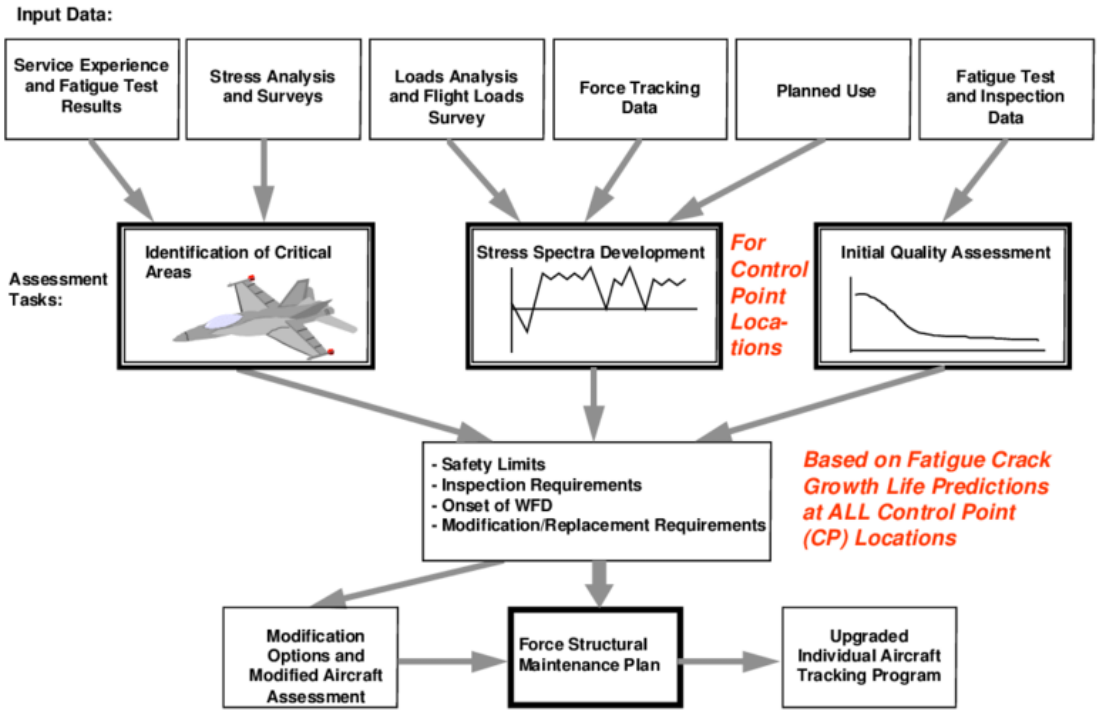

Figure 1.1: An overview of the end-to-end fatigue and damage tolerance analysis process. This thesis focuses solely on the topic of crack propagation analysis. [1]

A variety of methods are available for performing crack propagation analysis, ranging from simple and conservative methods used in general applications, to highly complex methods used in specialised applications, where a high degree of accuracy is required or where the available margin for error is limited. Many aircraft primary structures – such as the wing skins, stringers, and spars – are thin-walled, and can therefore be approximated as

two-dimensional structures under plane-stress conditions. Two-dimensional structures are relatively simple to analyse, because analytical and empirical methods are available which can be used to obtain reasonable and conservative approximations of crack propagation rates. However, some critical primary structures are too thick to be approximated as two-dimensional – these include the reinforcing elements around the main landing gear, for example, which are some of the most highly loaded areas on the entire aircraft. For these areas, more complex approaches are necessary, which normally involve the use of numerical methods – particularly finite element analysis.

Although finite element analysis provides a highly effective method of calculating crack propagation rates, it generally introduces significant complexity when compared to the standard empirical and analytical methods. An engineer with a specialised skill-set is required, and significantly more time is necessary to validate that the finite element model accurately represents the real-world structure. One of the key drivers of this additional complexity is the geometry creation and meshing process. When using simpler analytical methods, a global finite element model (GFEM) may be used. A GFEM is a finite element model which models an entire section of an aircraft's structure – such as a wing, fuselage section, or empennage – in coarse detail, using large shell and bar elements on the order of 100 mm [14]. Rather than providing specific loads at each feature, a GFEM provides average loads in the general vicinity of the feature. These loads are used as inputs for idealised analytical methods, using solutions obtained from reference books or software such as NASGRO. These solutions use the remote loading along with idealised two-dimensional geometry to calculate the stress intensity factor for the feature, and therefore predict the crack propagation rate. Since a GFEM only provides average loads, it is unsuitable for performing crack propagation analysis calculations directly. For this to be accomplished, a finer DFEM (detailed finite element model) needs to be created for the specific feature being analysed, with mesh elements on the order of 1 mm or less. The difference in idealisation between a GFEM and a DFEM is presented in Figure 1.2.

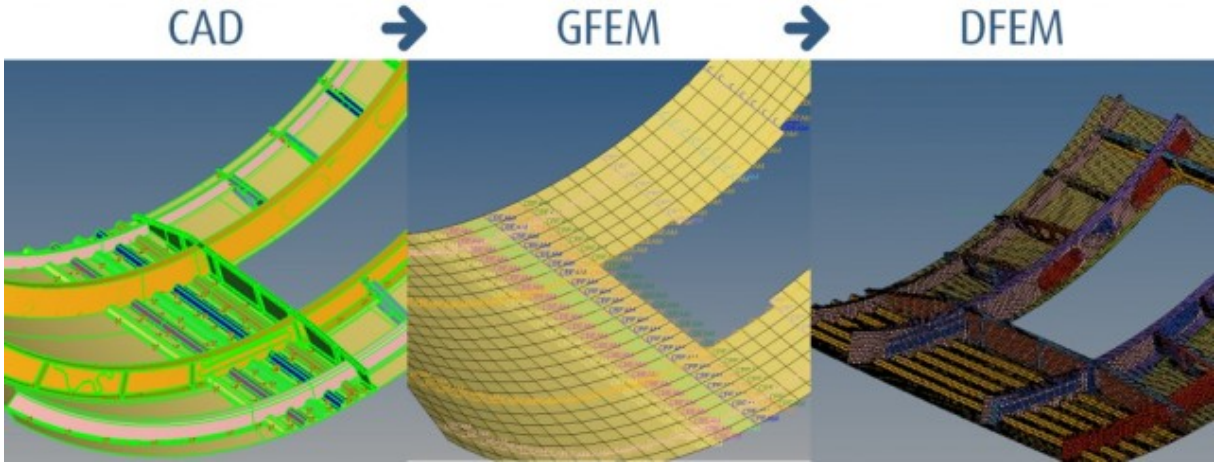

Figure 1.2: An overview of the difference in structure idealisation between a GFEM and a DFEM. The CAD model captures the geometry accurately. The GFEM idealises the CAD model using a coarse mesh of shell and bar elements, while the DFEM uses a fine mesh of solid elements [2].

Creating a DFEM is a complex task in itself, as it is necessary to ensure that the model closely represents the geometry and boundary conditions of the real-world structure, while minimising computational costs by introducing simplifications where possible. However, creating a DFEM for use in crack propagation analysis is even more complex, due to the limitations present in the type of mesh that must be used. Most finite element software packages are only capable of performing crack propagation analysis using a structured mesh of brick elements. However, creating these meshes is a complex and labour-intensive process for many crack configurations – particularly where re-meshing must be performed repeatedly as the crack advances. The alternative approach is to use auto-meshing tools, that are able to automatically and repeatedly mesh a DFEM using an unstructured tetrahedral mesh as the crack advances. However, most software does not directly support three-dimensional crack propagation analysis using an unstructured mesh, and the specialised software available that is available is expensive, and lacks the comprehensive technical support of the main software packages.

The potential was therefore identified for significant time and cost savings to be realised, if three-dimensional crack propagation analysis could be easily performed using an unstructured, tetrahedral mesh. It has been demonstrated that accurate solutions can be obtained via this approach [15] [16] [17]. However, this capability remains limited to specialised software packages and internal company tools – neither of which are freely available or customisable. It was the aim of this project to develop a piece of software which was able to accurately perform crack propagation analysis using a three-dimensional model with a tetrahedral mesh, which could be made freely available for use, extension, and customisation.

1.2 Aims and Objectives

The overall aim of this project was to develop a piece of software that was able to accurately calculate the stress intensity factor of a sharp crack within a three-dimensional finite element model, which was meshed using an unstructured mesh generated via an auto-meshing tool. The software could then serve as a base for further development – providing a free and customisable tool that was available for other engineers to build upon.

This overarching aim was broken down into the following three objectives:

1. Perform a literature review in order to understand the state of the art in terms of computational modelling of crack propagation, with a particular focus on finite element analysis using three-dimensional tetrahedral meshes. Select the most appropriate methods and tools for the implementation of the software.
2. Develop a piece of software which could interface with an industry-standard finite element analysis software package, extract the results of an analysis performed using a model with an unstructured mesh, and calculate the stress intensity factor of a crack within that model. The following limitations were specified, in order to control the complexity and time-scale of the project:
    - Only the analysis of straight crack fronts was required. The analysis of curved crack fronts was out of scope.
    - Only the analysis of static cracks was required. Automatically re-meshing and performing repeated calculations for an advancing crack was out of scope.
    - Only the analysis of thin structures under plane-stress assumptions was required. Although the eventual intended application of the software was the analysis of thick structures, the prototype was required only to be capable of analysing a three-dimensional solid model, and not necessarily the capability of analysing thick structures under plane-strain assumptions.
3. Validate the software using a case-study of a through-thickness edge crack within a finite plate, under plane-stress assumptions. Compare the outputs of the software using both a 2D and a 3D model to results obtained via analytical methods and the available literature to demonstrate the validity of the method and the implementation.

1.3 Methodology

1.3.1 Literature Review

A review of the available literature was conducted, with the aim of providing additional context for project and the industry requirement for the software. The available methods and tools which could be used for the implementation of the software were also reviewed. The aim of this section was to provide the theoretical background necessary for the implementation, as well as selecting a method of calculating the stress intensity factor which was likely to provide sufficient accuracy and was able to be implemented correctly within the available timescales.

1.3.2 Case Study Definition

A case study was then defined which would be used to validate the developed software. A through-thickness edge-crack was selected, due to the fact that it was simple to model, had widespread practical use within aerospace fatigue and damage tolerance analysis, and had well-established analytical methods which could be used to obtain results for comparison.

1.3.3 Model Definition

A finite element analysis package was first selected to be used as a base for the custom software that was developed. Abaqus was found to be the most suitable package, due to its widespread use in the aerospace sector and its comprehensive scripting API (application programming interface). The models for the edge-crack case study were then modelled in Abaqus – one using a 2D triangular mesh, and a second using a 3D tetrahedral mesh. A linear-static analysis was then performed in Abaqus using the standard implicit solver.

1.3.4 Software Implementation

A selection process was first performed in order to determine the most suitable programming language for the software tool. Python was selected as the language due to its fast development speed, wide availability of numerical libraries, and compatibility with the Abaqus API. The software was then developed, split into three main functions:

1. Automating the creation and analysis of the case study models, in order to allow parameters to be varied more easily, and to ensure that a consistent modelling approach was used across all of the analyses.

2. Exporting the results from the proprietary Abaqus open database (ODB) format, to an open JSON format.

3. Performing the calculation of the stress intensity factor, via calculation of the J-integral, using the equivalent domain integral method.

1.3.5 Software Validation

The results produced using the software were then validated, to ensure consistency and accuracy. A mesh sensitivity study was performed first, in order to demonstrate that the results converged as the element size decreased. The domain-independence of the results was then verified – this was a key step to ensure that the equivalent domain integral method had been implemented correctly. The impact of different weight functions was also investigated, to ensure the selection of the most suitable function. Finally, the J-integrals and stress intensity factors calculated using the software for a variety of crack lengths were compared to analytically obtained results, to ensure that accurate results were produced.

Figure 1.3 presents the methodology of the project in the form of a diagram, with sections of the project split by colour, and relationships between different stages of the project shown by arrows.

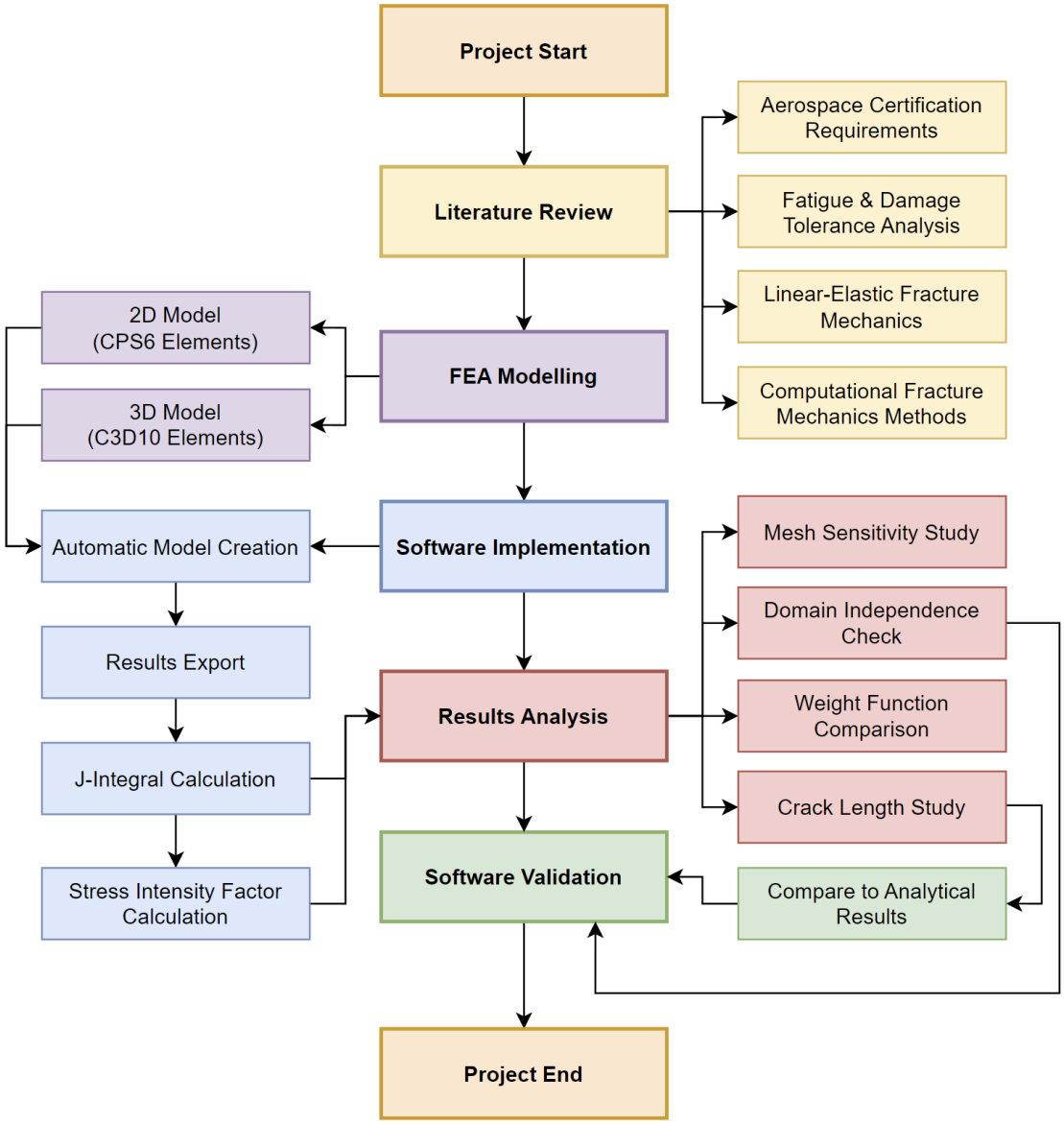

Figure 1.3: Methodology diagram for the project, with sections of the project split by colour, and relationships shown by arrows.

1.4 Thesis Structure

Following this introduction, the thesis is separated into four further sections, along with an appendix.

- Literature Review (§2) – This section details a review of the available literature relevant to the project. Additional context is given on the certification requirements and methods for damage tolerance within the civil aerospace sector. This is followed by a review of the field of linear-elastic fracture mechanics, and a discussion of the most important and relevant relationships. Finally, the application of computational numerical methods to fracture mechanics analysis is discussed.

- Implementation (§3) – This section first details the selection of the most suitable methods and tools for the development of the software, based on the findings of the literature review. The implementation details of the software are then discussed, including the architecture, the creation of the finite element models, the export of the results, and finally the calculation of the stress intensity factor via the J-integral.

- Results (§4) – This section discusses the validation of the results obtained from the software, using a case study of a through-thickness edge-crack in a finite plate – in both 2D and in 3D. The rationale behind the selection of suitable parameters for the case study is provided, including the mesh density, weight functions, and integration domains. Finally, the J-integral and stress intensity factor results obtained from the software are compared to analytically-obtained results, in order to verify that the selected method has been implemented correctly, and the results produced via the software are accurate.

- Conclusion (§5) – This section summarises the project, and discusses how the project was able to meet its original aims and objectives. Some limitations of the selected method and implementation are discussed, and information is provided on the further work necessary to continue the development of the software if it were to be fully realised as a useful analysis tool in the civil aerospace industry.

- Appendix (§6) – This section provides the source code for the software. A link to the GitHub repository for the project is also provided.

Chapter 2

Literature Review

The aim of this section is to review the available literature in the areas of aerospace certification requirements, fatigue and damage tolerance analysis, linear-elastic fracture mechanics, and computational fracture mechanics. Aerospace certification requirements are first reviewed, in order to give context on the necessity of crack propagation calculations within the aerospace sector. An overview of fatigue and damage tolerance analysis is then provided, followed by a section on linear-elastic fracture mechanics, which introduces the theoretical basis for the software implementation. Finally, the section on computational fracture mechanics discusses the methods available for the implementation of the software.

2.1 Aerospace Certification Requirements

As discussed in §1.1, fatigue and damage tolerance are critical factors influencing structural design in the civil aerospace sector. The fatigue and damage tolerance requirements that must be met in order for a large civil aircraft to be certified within Europe are defined in EASA Regulation CS §25.571 – 'Damage tolerance and fatigue evaluation of structure' [18]. These regulations have changed and developed over time, as new findings have been obtained from research, experience, and investigations of structural failures due to fatigue. There are four main approaches that have been used over the past century, all of which still applied today to various parts of a modern airframe.

Safe-Life (Safety by Retirement)

This is the oldest and most conservative approach. Under this approach, a structure is designed such that the chance of crack initiation occurring within a specified service life is almost zero, using a combination of load spectra, physical testing, and factors of safety. The structure is retired once this service life has been reached, regardless of its actual condition. This approach has the highest margin of safety, but leads to higher costs and lower performance due to a combination of unnecessary weight, and potentially retiring components which still have a significant amount of life remaining. The safe-life approach has been superseded in most cases, but is still used for heavily-loaded components where the inspection of cracks before they become critical can be difficult [19].

Fail-Safe (Safety by Design)

The fail-safe approach was developed in order to remedy a key failing with the safe-life approach - namely that if a safe-life structure were to fail, the results can be catastrophic due to the fact that there is no redundancy considered within the approach. The key argument of the fail-safe approach is that a structural component should have a redundant load path – a secondary component that can temporarily take the load of a failed component until the structure is inspected and the failed component either repaired or replaced [20].

Damage Tolerance (Safety by Inspection)

This approach relies on the regular inspection of components in order to detect cracks and replace components before failure occurs. An initial crack is assumed to exist in a component at the beginning of its life, and fracture mechanics principles are used to calculate the rate of crack growth, and the point at which failure will occur. These values are then used to set an inspection threshold and repeat inspection intervals, to ensure that the component will be expected in the interval between the crack becoming detectable and the crack becoming critical, at which point the component will be repaired or replaced [20].

Widespread Fatigue Damage (WFD)

This is a relatively new requirement for aircraft, the introduction of which was necessary as the age of in-service aircraft grew over time. WFD refers to the possibility of multiple small cracks developing within different places within the structure. This can either be multi-side damage (MSD) for cracks present within different areas of the same structure, or multi-element damage (MED) for cracks present within similar adjacent structural elements. A limit of validity (LOV) must be defined for an aircraft, which is the limit at which available engineering data can demonstrate that WFD will not occur in the aircraft structure. The data may include test evidence, analysis, in-service data, and teardown inspection results of high-life aircraft [8].

2.2 Fatigue & Damage Tolerance Analysis

The analysis approaches used within the civil aerospace sector to demonstrate compliance with the previously discussed certification requirements can be broadly split into fatigue analysis, and damage tolerance analysis. Pure fatigue analysis is used to demonstrate a structure's compliance with the safe-life or fail-safe requirements, while additional damage tolerance analysis is used to demonstrate a structure's compliance with the damage tolerance and widespread fatigue damage requirements. Most primary structures are required to be damage tolerant, and are therefore subjected to both fatigue and damage tolerance analysis.

2.2.1 Fatigue Analysis

Fatigue analysis is used to demonstrate compliance for structures that fall under the certification classification of safe-life or fail-safe. The key output of a fatigue analysis is a factored fatigue life for a structural component, given in flight cycles (FC). An unfactored fatigue life is the number of aircraft ground-air-ground (GAG) cycles that a component can be demonstrated to withstand before any crack initiation occurs – assuming that the as-manufactured component was pristine and free of any cracks. This is multiplied by some factor of safety (usually either three or five) in order to produce the factored fatigue life. Fatigue analysis requires several key inputs, including data on the component's loading, geometry, and material, along with physical test data [20].

Load Spectra

The load spectrum of an aircraft is a statistical representation of the series of cyclic loads that the average aircraft of that type will be subjected to during a typical flight cycle. Various load types are considered, including ground loads, cruise flight loads, manoeuvre loads, gust loads, and pressurisation loads. The load spectrum is created from a combination of computational methods, statistical analysis, and physical testing such as wind tunnel testing. It can further be tuned using flight test data, once it becomes available. The load spectrum is used along with a global finite element model (GFEM) in order to determine the individual loads on each component, which are then used to calculate individual stresses for each component. Finally, techniques such as the rainflow counting algorithm are used to sum up the effects of the individual loading cycles, and produce a single stress value that can be used to calculate the fatigue life for a component - this is known as the fatigue equivalent stress, or $S_{EQ_{FAT}}$ [20].

Geometry

In order to convert the global aircraft loads into local component stresses, geometry is a key consideration. The most critical geometric influence on the fatigue life is the stress

concentration factor (SCF). The stress concentration factor is a dimensionless measure, which is a ratio between the local stresses observed at a geometric feature - for example, a hole, fillet, or radii - and the remote stress observed in a plain area of the component. The stress concentration factor can be calculated using empirical data – such as those available in [21] – or using linear-elastic finite element models. Stress concentrations are significant in fatigue analysis as they act as preferential sites for crack initiation, and can significantly reduce the fatigue life of a component.

Material Properties

Material properties are also a significant contributor to the fatigue life of a component, as materials differ significantly in their resistance to fatigue. Fatigue properties of a material are usually captured within a stress-life curve (S-N curve), which plots the applied fatigue stress versus the number of cycles to failure for a material. Values for this curve are generally obtained via coupon tests of the material in question, performed at various stress levels. The domain of most interest for aerospace structures is the high-cycle fatigue regime, which covers from around 10^3 cycles up to around 10^6 cycles. Factors such as the R-ratio, size effect, and surface finish of the coupons must be considered in order to be able to apply the results of the coupon testing accurately to the actual structure.

Fatigue Life Calculation

Using the available loading, geometry, and material data, an unfactored fatigue life can be calculated for the component. Safety factors are then applied to this to create a factored fatigue life. The magnitude of the safety factor used is dependent on the structure being analysed and the available test evidence. Finally, the factored fatigue life can be compared to the design service goal (DSG) of the aircraft – which is the number of flight cycles that the aircraft is being certified for. If the factored fatigue life is greater than the design service goal, then the analysed component has an extremely low risk of crack initiation during the lifetime of the aircraft, and can therefore be safely used. If the factored fatigue life is less than the design service goal, then it may be necessary to introduce inspections for the component into the maintenance plan, or replace the component midway through the aircraft's service life

2.2.2 Damage Tolerance Analysis

Damage tolerance analysis is used to demonstrate the compliance of a structure with the damage tolerance and widespread fatigue damage certification requirements. In contrast to fatigue analysis, damage tolerance analysis is mainly concerned with the growth rate of pre-existing cracks in a structure, and the determination of inspection intervals which enable these cracks to be detected and mitigated before reaching the critical length that will lead to structural failure. Linear-elastic fracture mechanics (LEFM) is the key theoretical basis for damage tolerance analysis, which is discussed in more detail in §2.3. Damage tolerance analysis involves determining the likely sizes and locations of any flaws, calculating the rate at which that they will grow, and determining the necessary inspection interval to ensure they are detected in a timely manner.

Initial Flaws

Damage tolerance analysis presumes that small flaws (i.e. cracks) exist within any structure, which are present from the beginning of the service life of the aircraft. The sizes and prevalences of these flaws are estimated based on the statistical analysis of quality inspection data obtained during manufacture and in service, while knowledge of non-destructive testing (NDT) techniques is used to determine the likelihood of detecting cracks at a given size. The aim of this process is to conservatively estimate the maximum size of flaw which is realistically able to escape the manufacturing environment, without being discovered during a quality inspection. This is known as a rogue flaw, and a value of 0.05" or 1.27 mm is commonly used [22]. A stress survey is then performed using a GFEM in order to select a primary crack location for a structure based on its criticality. This process takes into account factors such as loading and geometry in order to determine the location at which the presence of a crack would lead to the failure of the structure within the lowest number of loading cycles. A secondary location is also selected, which is generally the redundant structure which would be subjected to the redistributed load upon failure of the primary structure. At the secondary location, a smaller crack is assumed to be present – this is known as a quality flaw, and a value of 0.005" or 0.127 mm is used – one order of magnitude less than the rogue flaw. These assumptions represent a conservative "worst-case" scenario for crack propagation, where cracks are present at both the most highly stressed location, along with its redundant structure. An example of this for a skin-stringer panel section is presented in Figure 2.1.

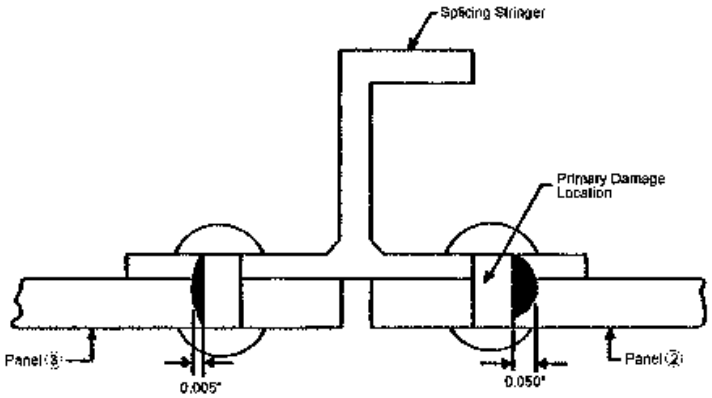

Figure 2.1: Primary and secondary damage locations within a skin-stringer section. The primary location has a 0.05" rogue flaw and the secondary location has a 0.005" quality flaw. The flaws impact both the skin panel and the stringer, as both are drilled during the same operation [3].

Crack Growth and Failure

Once the primary and secondary damage locations have been selected, the crack propagation life of the structure is then calculated – this is the number of flight cycles that causes the crack to grow to the length necessary to cause failure of the structure. Most aircraft structures are thin-walled, and therefore can be idealised as two-dimensional. Well-established analytical methods are then available to calculate the stress intensity factor (SIF) for these idealised structures – this is discussed in further detail in §2.3.1. These analytical methods are not usually applicable to thicker three-dimensional structures, and therefore numerical approaches must be used (discussed in further detail in §2.4). The loading of the structure is determined using the approach described in §2.2.1, which is then used to calculate the crack propagation equivalent stress – $S_{EQ_{CP}}$. This parameter is similar to $S_{EQ_{FAT}}$, but additionally considers the angle of crack growth relative to the direction of applied loading. Once an accurate stress intensity factor has been obtained, the rate of crack growth can then be determined using the Paris-Erdogan equation (Equation 2.7). This process is repeated by extending the crack, and recalculating the stress intensity factor for this slightly longer crack. At each step, the limit stress (σ_{limit}) – which is the maximum stress that the structure is likely to be subjected to during its entire service life – is compared to the residual strength of the component. A visualisation of the crack growth curve vs the residual strength of the structure is presented in Figure 2.2. When the stress intensity factor of the crack under an applied stress of σ_{limit} reaches the critical stress intensity factor of the material (K_{IC}), the structure fails. The length of the crack at this point is known as the critical crack length (a_{crit}) [23].

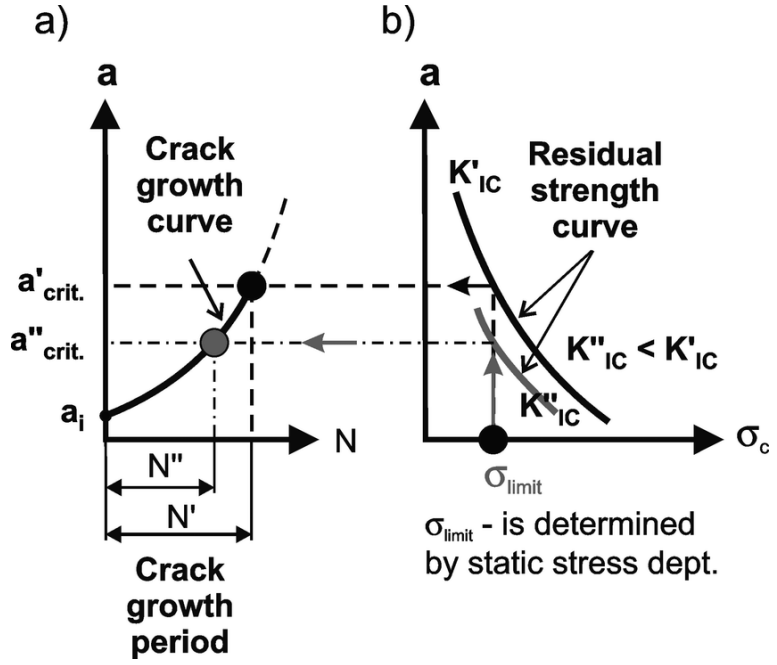

Figure 2.2: Diagram showing the crack growth curve (a) along with the residual strength curve (b). When the stress intensity factor of the crack under an applied stress of σ_{limit} reaches the critical stress intensity factor of the material, the structure fails [4].

Threshold and Inspection Intervals

The previously calculated crack propagation lives can then be used to determine the inspection threshold and inspection intervals for the analysed structure. The inspection threshold is the point at which the first inspection of the structure is required, while the inspection intervals are the points at which repeat inspections are required. For example, a structure with an inspection threshold of 24,000 FC and an inspection interval of 6,000 FC would require an initial inspection once the aircraft has completed its first 24,000 flight cycles, and additional inspections after every subsequent 6,000 flight cycles. The inspection threshold is generally half of the structure's factored fatigue life, while the inspection intervals are determined based on the crack propagation lives determined via damage tolerance analysis [1].

The parameters of interest when determining the inspection intervals are a_{crit}, a_{det} $N_{primary}$ and $N_{secondary}$. The critical crack length (a_{crit}), is the crack length at which the secondary damage location fails, while ($N_{primary}$) and ($N_{secondary}$) are the number of crack growth cycles to failure for the primary and secondary damage locations respectively. The final parameter (a_{det}) is the detectable crack length, which is the length at which the crack in the primary damage location becomes detectable via the inspection method specified for the structure. The selected method differs depending on the accessibility and visibility of the structure in question.

The inspection intervals are determined based on the number of cycles that occur between the crack at the primary location becoming detectable (when the crack length at the primary location reaches a_{det}), and the crack in the secondary location causes failure of the

component (when the crack length at the secondary location reaches a_{crit}). The interval between these two parameters provides a window of opportunity, where a crack can be reliably detected, but before structural failure has occurred. The inspection interval is then divided by a safety factor of two. This provides two opportunities to detect the crack, to account for a the possibility of an inspection failing to detect a crack that should have been detected. Figure 2.3 demonstrates the period of safe crack growth that is used to determine the damage tolerance inspection intervals for an aircraft structure. Note that the rate of crack growth at the secondary location increases once the primary member has failed, due to load being distributed from the primary to the secondary location [5].

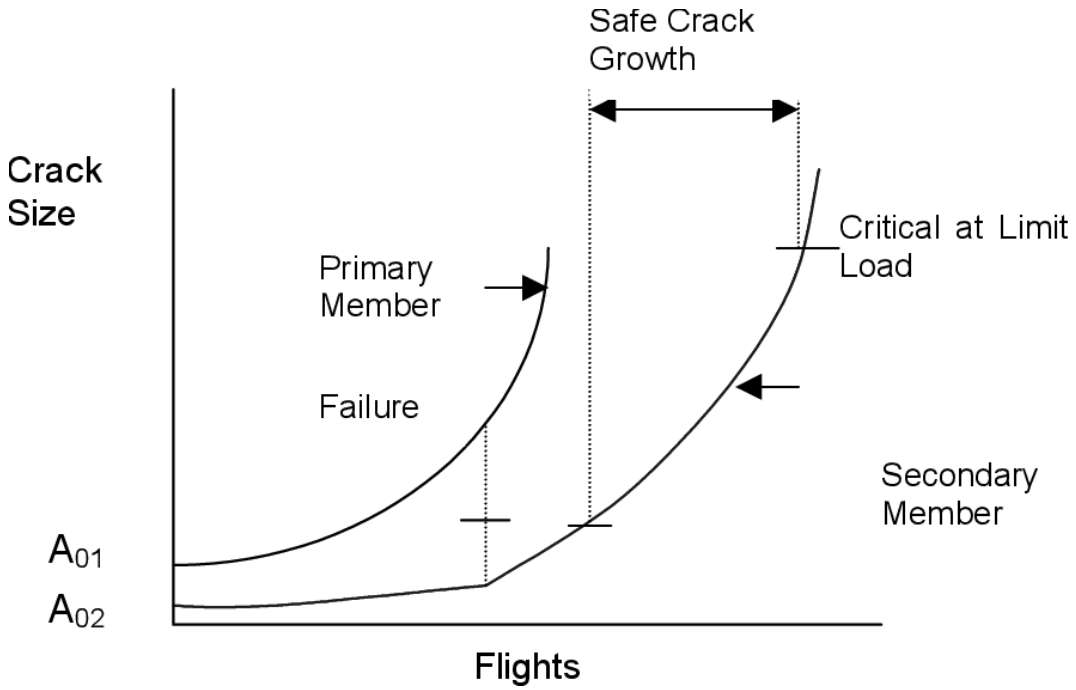

Figure 2.3: Crack propagation chart showing the growth of a crack at a primary and secondary location as the number of flight cycles increase. Once the primary location has failed, redistributed load accelerates the growth of the crack at the secondary location. [5].

2.3 Linear Elastic Fracture Mechanics

Linear Elastic Fracture Mechanics (LEFM) is a fundamental framework within the field of fracture mechanics. The key underlying assumption of LEFM is that the overall deformation of a cracked body is linear-elastic. This means that the plastic zone at the crack tip is small relative to the length of the crack, and that the stress state at the crack tip can be accurately described using only the elastic stresses in the material. LEFM was first developed by Griffith in 1920, via the application of the first law of thermodynamics to the problem of crack formation and propagation [24].

The first law of thermodynamics states that energy must always be conserved – this implies that when a system moves from a non-equilibrium state to an equilibrium state, there must be no net increase in energy. Griffith applied this principal to fracture mechanics in the form of a global energy balance – where the elastic energy stored in a stressed body was balanced against the increase in surface energy necessary to create new crack surfaces in that body.

The growth of a crack within a body results in an increase in the surface energy of that body – this is because the creation of new crack surfaces requires work to be done in order to break atomic bonds at those surfaces. The creation of new surfaces relaxes the stresses within the body, and therefore reduces the elastic potential energy at the crack faces. Griffith's model stated that crack extension would occur when the elastic potential energy stored in a cracked body was equal to the increase in the surface energy of the body due the extension of the crack. Restated, crack extension would occur at the point where there was no net change in the total energy of the body due to the extension of the crack.

Equation 2.1 presents Griffith's equation for a through-thickness crack in an infinitely wide plate subjected to a remote tensile stress. Griffith's model provided good agreement with experimental data for brittle materials. However, the surface energy predicted by the model was unrealistically high for ductile materials.

$$\sigma_f = \sqrt{\frac{2E\gamma_s}{\pi a}} \tag{2.1}$$

- σ_f is the tensile stress necessary to cause fracture of the material.
- a is the crack length.
- E is the elastic modulus.
- γ_s is the surface energy per unit area.

Irwin later modified Griffith's model to account for the effects of localized plastic deformation within ductile materials, and therefore increase the predictive accuracy of the model for these materials. Irwin found that in ductile materials, a plastic zone develops at the tip of the crack, resulting in additional work being done and heat being dissipated during the process of crack extension. Irwin added an additional term (γ_p) to Griffith's equation, which is the plastic work done per unit area of surface created. Equation 2.2 presents Irwin's corrected equation for fracture stress.

$$\sigma_f = \sqrt{\frac{2E(\gamma_s + \gamma_p)}{\pi a}} \tag{2.2}$$

Irwin later proposed an energy approach for fracture, known as the strain energy release rate [25]. This was essentially equivalent to Griffith's model, but was revised into a more convenient form. Irwin defined the strain energy release rate (G), which is a measure of the decrease in total potential energy per increase in fracture surface area. Equation 2.3 gives the formula for the strain energy release rate G, for a wide plate under plane stress conditions with a centre crack of length $2a$ and an applied tensile stress of σ. Crack extension occurs when G reaches the critical value of G_c, where G_c is a measure of the fracture toughness of the material.

$$G = \frac{\pi \sigma^2 a}{E} \tag{2.3}$$

2.3.1 Stress Intensity Factor

The stress concentration factor (K_t) is used to characterise stress distributions around features such as notches and holes, and is defined as the ratio between the stress in the immediate vicinity of the feature, and the stress at some point in the bulk material, remote from any features. However, K_t ceases to become a meaningful concept near the tip of a sharp crack – as the radius of the crack tip approaches zero, the stress approaches a value of infinity. The definition of the stress intensity factor (SIF) by Irwin was therefore a key development in the field of linear-elastic fracture mechanics, as it enabled the stress state at the tip of a sharp crack to be characterised using a single parameter [26]. For a polar coordinate system defined with the origin at the crack tip, it can be shown that the stress field in any linear elastic cracked body can be represented by Equation 2.4 [8].

$$\sigma_{ij} = \left(\frac{k}{\sqrt{r}}\right) f_{ij}(\theta) + \sum_{m=0}^{\infty} A_m r^{\frac{m}{2}} g_{ij}^{(m)}(\theta) \tag{2.4}$$

- σ_{ij} is the stress tensor.
- r and θ are parameters defining the polar coordinate system presented in Figure 2.4.
- k is a constant.
- f_{ij} is a dimensionless function of θ.
- A_m is the amplitude.
- $g_{ij}^{(m)}$ is a dimensionless function of θ for the mth term.

The parameters in the second part of the expression are dependent on the specific configuration of the problem being analysed. However, the first part of the expression is proportional to $\frac{1}{\sqrt{r}}$ for any possible geometry. As r approaches zero, the first part of the expression approaches infinity, while the second part approaches zero. Therefore, Equation 2.4 demonstrates that the stress near the crack tip is proportional to the $\frac{1}{\sqrt{r}}$ term, regardless of the configuration of the cracked body.

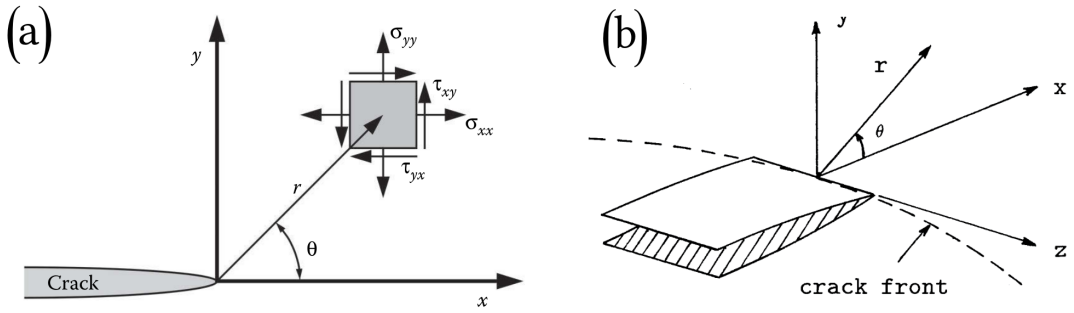

Figure 2.4: A diagram showing crack tip geometry in two dimensions (a) and three dimensions (b), showing the polar coordinate system constructed using r and θ. [6].

Equation 2.4 may then be used to define the stress intensity factor (K) by introducing a π term for the sake of convenience, as given by Equation 2.5. The proportionality constants k and f_{ij} are dependent on the configuration of the problem, such as the geometry, loading, and boundary conditions. This therefore implies that K is also dependent on these factors.

$$K = k\sqrt{2\pi} \tag{2.5}$$

The stress intensity factor can then be obtained in terms of the applied stress and the crack length – this is presented in Equation 2.6. The parameter β_i is a shape factor that accounts for the configuration of the specific problem, for the specific loading mode in question. For a straight crack of length $2a$ which is embedded in an infinite plate and subjected to a uniform stress of σ acting perpendicular to the crack, β is equal to one. For other configurations, β must be determined via analytical equations, numerical methods, or physical testing.

$$K_i = \beta_i \sigma \sqrt{\pi a} \tag{2.6}$$

- K_i is the stress intensity factor for the crack loading mode i.
- β_i is a dimensionless shape factor relating to the specific configuration of the problem, for the crack loading mode i.
- σ is the applied stress.
- a is the crack length.

Stress intensity factor solutions for many two-dimensional plane-stress configurations are readily available both within reference handbooks Sih [27] and commercial software such as NASGRO and AFGROW. An example is presented in Figure 2.5, which shows a SIF solution within NASGRO – along with its limits of validity – applicable to an edge crack in a finite plate. These solutions provide a method of quickly determining the stress intensity factor, without necessitating the comprehensive validation required when using a DFEM. However, most SIF solutions are only applicable to simplified and idealised geometry, and therefore they are not suitable for every problem. For example, the TC14 solution presented in Figure 2.5 is only valid for length-to-width ratios between 0.2 and 10. Obtaining the stress intensity factor for configurations which do not have solutions already available is more complex, and usually requires the use of finite element analysis. This is discussed further in §2.4.

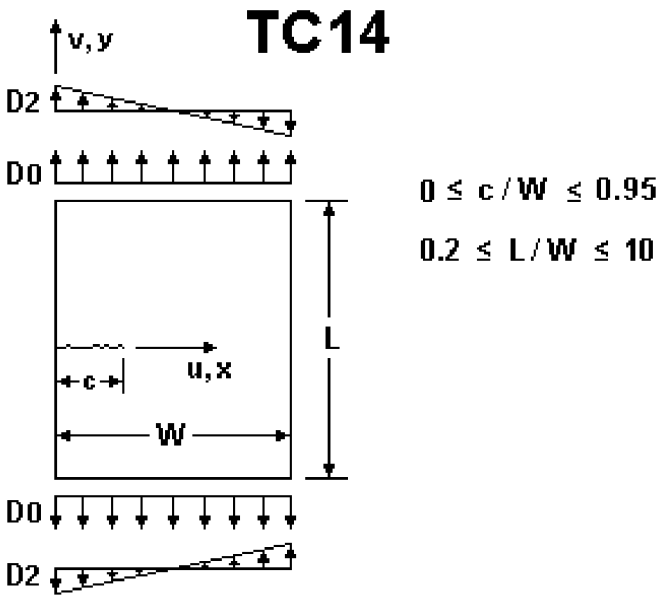

Figure 2.5: Diagram showing SIF solution TC14 from the NASGRO manual, which can be used to analyse through-thickness edge cracks within a finite plate [7].

The rate of crack growth has been experimentally shown to be a function of the range of the stress intensity factor exhibited within a loading cycle. This is a critical relationship within aerospace damage tolerance analysis, as it allows the crack propagation life of a structure to be calculated using the stress intensity factor. This relationship is described by the Paris-Erdogan equation, presented as Equation 2.7 [8].

$$\frac{da}{dN} = C(\Delta K)^m \tag{2.7}$$

- da/dN is the change in crack length per loading cycle.
- ΔK is the change in the stress intensity factor within the loading cycle, equal to $(K_{max} - K_{min})$.
- C and m are material-specific properties which are obtained experimentally.

2.3.2 Crack Loading Modes

Cracks may be subjected to three different types of loading, which are demonstrated in Figure 2.6. Mode I loading occurs when the principal load is applied in the y direction in order to open the crack - this is known as the opening mode. Mode II loading occurs when when an in-plane shear load is applied in the x direction - this is known as the sliding mode. Finally, Mode III loading occurs when an out-of-plane shear load is applied in the z direction - this is known as the tearing mode.

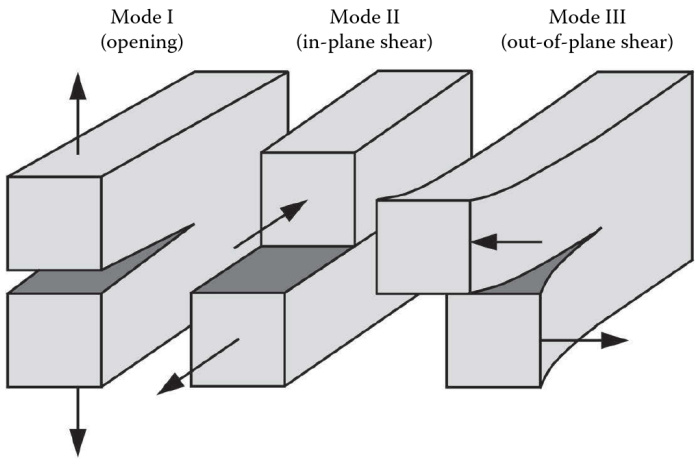

Figure 2.6: Diagram showing Mode I, Mode II, and Mode III crack loading [8].

Separate expressions for the stress intensity factor can be defined for each mode of loading. These are given in Equations 2.8, 2.9, and 2.10 [8].

$$\lim_{r\to 0} \sigma_{ij}^{(I)} = \frac{K_I}{\sqrt{2\pi r}} f_{ij}^{(I)}(\theta) \tag{2.8}$$

$$\lim_{r\to 0} \sigma_{ij}^{(II)} = \frac{K_{II}}{\sqrt{2\pi r}} f_{ij}^{(II)}(\theta) \tag{2.9}$$

$$\lim_{r\to 0} \sigma_{ij}^{(III)} = \frac{K_{III}}{\sqrt{2\pi r}} f_{ij}^{(III)}(\theta) \tag{2.10}$$

Generally the mode of most concern when analysing crack propagation problems is Mode I, as this is the mode that occurs most frequently and produces the most damage. It has therefore received the most attention with regards to the research and development of damage tolerance methods. However, consideration of the other modes is still a significant factor, particularly when considering three-dimensional components which are made up of ductile materials and are subjected to multi-axial stresses. In these scenarios, a crack experiences a combination of different loading modes simultaneously – this is known as mixed-mode fracture mechanics.

The principal of linear superposition can be used to calculate the sum the contributions of each loading mode to the overall stress tensor using Equation 2.11 [8].

$$\sigma_{ij}^{(total)} = \sigma_{ij}^{(I)} + \sigma_{ij}^{(II)} + \sigma_{ij}^{(III)} \tag{2.11}$$

The stress intensity factor is also specific to loading mode, and can also be superimposed via linear superposition. This is demonstrated by Equation 2.12 [8]. This equation can be used to calculate the total stress intensity factor of a crack that is subjected to mixed-mode loading – by calculating the stress intensity factor for each mode, and then summing the results. The process may also be performed in reverse, by calculating the total stress intensity factor for a crack, and then decomposing it into the separate stress intensity factors for each mode of loading. However, this is out of scope for this thesis, which focuses on a case study relating solely to Mode I loading.

$$K_{total} = K_I + K_{II} + K_{III} \tag{2.12}$$

2.3.3 J-Integral

As discussed previously, the stress intensity factor is an extremely useful parameter which can be used to accurately calculate the crack propagation lives of structures. However, the available stress intensity factor solutions are mostly limited to two-dimensional idealisations, meaning that calculating the stress intensity factor for cracks in three-dimensional requires individual analysis using the finite element method. Therefore, a method of obtaining the stress intensity factor from the results of a finite element analysis is necessary. This is often accomplished via the use of the J-integral.

The J-integral was first described by Rice in 1968, and is an energy-based parameter which is closely related to the strain energy release rate (G) discussed briefly in §2.3. The J-integral is a path-independent contour integral around the tip of a crack, and is defined in Equation 2.13 [28].

$$J = \int_{\Gamma} \left(W\, dy - T_i \frac{\partial u_i}{\partial x}\, ds \right) \tag{2.13}$$

- W is the strain energy density.
- dy is an infinitesimal section taken perpendicular to the direction of crack growth.
- T_i is the traction vector.
- $\frac{\partial u_i}{\partial x}$ is the rate of change of the displacement vector in the direction of crack growth.
- ds is an infinitesimal section of a line integral around the crack tip.

The first term of the equation essentially captures the state of elastic deformation around the crack tip, and represents the energy stored in the stretching of the atomic bonds of the material. The second term captures the effects of the work done by external forces to create new free surfaces as the crack grows. Both terms together therefore quantify the total energy release rate, which encompasses the contributions from both elastic and plastic deformation of the material. As J-integral is able to capture the effects of plastic deformations, it can also be used to perform elastic-plastic fracture mechanics (EPFM) analysis – where the plastic deformation around the crack tip is large relative to the crack length.

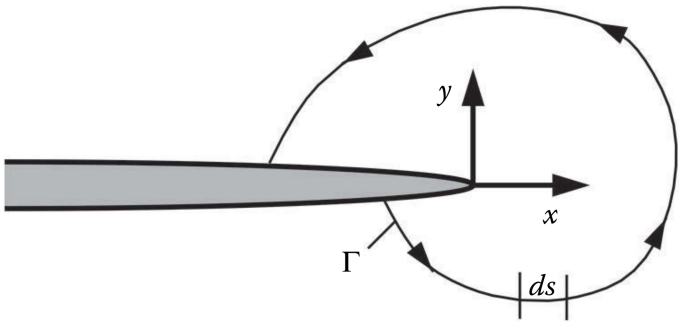

Figure 2.7: Diagram of a crack, showing the closed contour Γ that may be used to calculate J-integral [8].

For linear elastic materials, J-integral has been shown to be equal to the strain energy release rate (G) discussed in §2.3. This means that the J-integral can be directly related to the stress intensity factor, which is the key piece of theory underpinning this thesis. This relationship is given in Equation 2.14.

$$J = G = \frac{K_I^2 + K_{II}^2}{E'} + \frac{K_{III}^2}{2\mu} \tag{2.14}$$

- E' is the effective elastic modulus of the material, which is equal to E for plane stress conditions, and $E/(1-\nu)^2$ for plane strain conditions.
- ν is the Poisson's ratio of the material.
- μ is the shear modulus, which is equal to $E/2(1+\nu)$

Assuming that a given crack is subjected to Mode I loading only, Equation 2.14 can be simplified into Equations 2.15 and 2.16, which may be used to calculate J and K_I respectively for Mode I crack configurations.

$$J = \frac{K_I^2}{E'} \tag{2.15}$$

$$K_I = \sqrt{JE'} \tag{2.16}$$

This relationship means that if J can be determined for a given crack problem, then it is possible to calculate the stress intensity factor for that crack, and therefore calculate the crack propagation life of the structure using the Paris-Erdogan equation. However, the standard contour integral form of the J-integral is difficult to implement within finite element analysis software – particularly when using an unstructured mesh. The implications of this are discussed in §2.4.6.

2.4 Computational Fracture Mechanics

As discussed previously, the lack of readily available stress intensity factor solutions for three-dimensional crack configurations means that the use of numerical techniques such as finite element analysis is required. Several approaches are available, which can be broadly split into direct methods, and energy-based methods (such as the J-integral). In general, energy-based approaches are preferred due to their higher accuracy. However, direct approaches may still be used for validation, as they are simple enough to perform using a hand calculations [12].

2.4.1 Mesh-Based and Mesh-Free Methods

The available approaches for discretising the problem to enable the use of numerical methods can be split into mesh-based and mesh-free methods. Mesh-based methods – such as the finite element method (FEM), finite difference method (FDM), and finite volume method (FVM) – discretise a structure by converting it from a single continuous domain into a set of discrete elements, known as a mesh. This provides a clear representation of the problem domain, along with well defined connectivity between the elements and nodes which make up the mesh. A comparison of a mesh-based and a mesh-free discretisation of the same component is presented in Figure 2.8.

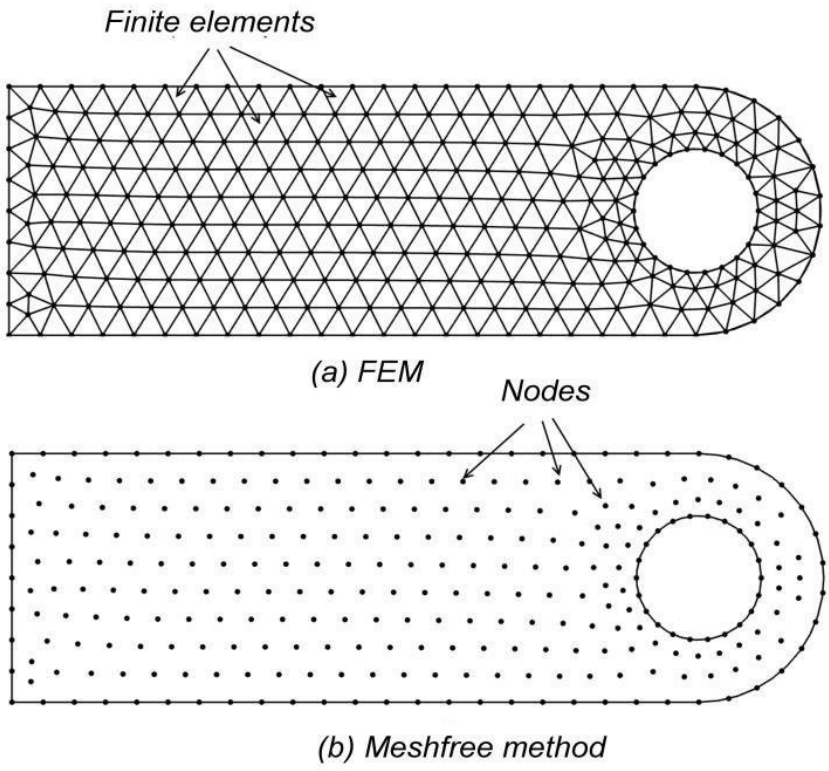

Figure 2.8: A comparison between a mesh-based and a mesh-free method. The upper model (a) is discretised into a mesh using nodes and elements, while the lower model (b) is discretised into a set of particles using only nodes [9].

Mesh-based methods are very well-established due to their relatively long history, and a variety of well-supported and well-validated software is available – particularly for the finite element method. However, the meshing process is computationally expensive, which is of particular concern for crack propagation analysis – where multiple steps of re-meshing may be required as the crack advances. This has been somewhat mitigated by the use of improvements to the original finite element method – such as the extended finite element method (XFEM), which uses enriched elements near the crack tip to attempt to accurately capture the singularity and model its growth without re-meshing [29]. However, these enriched methods are more complex and less well-supported in terms of available software, and can lead to numerical instability and a loss of convergence if correct modelling approaches are not followed [30].

Mesh-free methods forgo the use of a mesh - instead discretizing the domain by splitting it into a series of independent points or particles. This approach provides some flexibility when analysing certain problems, as the particles are more free to undergo large displacements without requiring computationally expensive re-meshing. Smoothed-particle hydrodynamics (SPH) and the element-free Galerkin method (EFGM) have both been successfully applied to crack propagation analysis [31] [32]. However, the conventional Eulerian smoothed-particle hydrodynamics formulation suffers from tensile instability [33], and the standard enriched EFGM method is capable of modelling straight cracks only [34]. Mesh-free methods are also generally more computationally expensive than mesh-based methods [35], and at present are mainly limited to research rather than industrial applications.

The finite element method is by far the most commonly utilised method for performing structural analysis within industry, due to its versatility, reliability, long history of use, and the wide availability of mature software packages. Therefore, the remainder of the literature review focuses on this method specifically.

2.4.2 Element Formulations

A key problem encountered when attempting to model a crack using the finite element method is the difficulty of accurately representing the singularity at the crack tip when discretising the structure. This leads to non-convergence of the results, since the polynomial interpolation functions used within standard elements are unable to accurately model the stress and strain singularities at the crack tip [36]. The most simple mitigation for this problem is to use a very fine mesh at the crack tip, which has been shown to be able to recover the full convergence of the finite element method [37]. However, increasing the mesh density also increases the solution time of the problem.

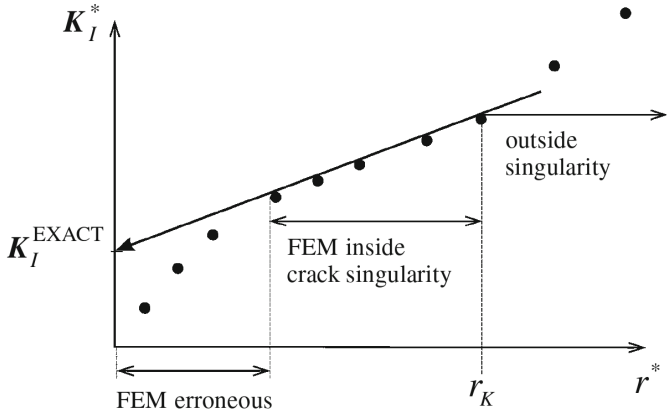

Figure 2.9: Diagram of the validity range of SIFs obtained using a fine mesh of regular standard elements to model a crack [10].

The alternative method of modelling the crack tip singularity is the use of specialised crack-tip elements, in which the shape functions contain singular crack-specific functions. These elements are used to model the area directly around the crack tip, while conventional elements are used to model the rest of the structure [10]. The most commonly used of these is the quarter-point element (QPE), in which the mid-side nodes along the two edges intersecting the crack-tip are moved from the half-point of the edge to the quarter-point – closer to the crack tip. These are relatively simple to implement for triangular elements, but more complex to implement for tetrahedral elements. However, quarter-point tetrahedral elements have been implemented successfully by separating them into corner-based or edge-based elements, depending on how they intersect the crack tip [38].

2.4.3 Stress & Displacement Extrapolation

A method of calculating the stress intensity factor by extrapolating the stresses or displacements near the crack tip was first proposed by Chan in 1970 [39]. This method is based on the correlation of the stresses and displacements obtained from finite element results to analytical equations for the crack tip stress and displacement fields. For plane strain, this relationship is given by Equation 2.17 [8].

$$K_I = \lim_{r \to 0} \left[\frac{E u_y}{4} \sqrt{\frac{2\pi}{r}} \right] \qquad (\theta = \pi) \tag{2.17}$$

- K_I is the Mode I stress intensity factor.
- E is the elastic modulus of the material.
- u_y is the crack opening displacement.
- r and θ make up a polar coordinate system, with $r = 0$ at the crack tip.

This equation relies on the fact that the stress intensity factor is related to both the applied stress and the distance from the crack tip, as illustrated in Equation 2.8. Therefore, if the stresses and displacements near the crack tip (at some value of r) can be accurately calculated, they can by extrapolated to the crack tip – where $r = 0$ – in order to calculate the stress intensity factor. More accurate results are generally obtained by extrapolating the displacements rather than the stresses, since stresses become singular as $r \to 0$, while displacements are proportional to $\sqrt{r}$. This means that the displacement field is smoother and thus less sensitive to mesh discretisation errors compared to the stress field. The key disadvantage of this method is that it requires a very fine mesh around the crack tip, and is very sensitive to the nodes which are selected to perform the extrapolation. These methods have largely been superseded by energy-based methods, which are generally more robust in practical applications [8].

2.4.4 Virtual Crack Extension

The virtual crack extension (VCE) method is an energy-based approach first described by Parks [40] and Hellen [41] in the late 1970s. The basis of the method is the simulation of a small virtual extension of a crack, from which the rate of change in potential energy can be calculated. The total potential energy of the system is given by Equation 2.18.

$$\Pi = \frac{1}{2}\mathbf{u}^T\mathbf{K}\mathbf{u} - \mathbf{u}\mathbf{F} \tag{2.18}$$

- Π is the total potential energy.
- $\mathbf{u}$ is the displacement vector.
- $\mathbf{K}$ is the stiffness matrix.
- $\mathbf{F}$ is the externally applied force.

If this expression is differentiated with respect to a small extension of crack length a, and the assumption is made that the external forces (F) do not change, Equation 2.18 can be simplified to produce Equation 2.19.

$$G = -\frac{\partial \Pi}{\partial a} = \frac{1}{2}\mathbf{u}^T\frac{\partial \mathbf{K}}{\partial a}\mathbf{u} \tag{2.19}$$

It can therefore be observed that the strain energy release rate G is dependent on the derivative of the stiffness matrix $\mathbf{K}$, based on the change in stiffness between the initial crack state and the extended crack state. The displacement solution $\mathbf{u}$ is only required for the initial state, simplifying the analysis. In order to calculate the change in stiffness, a small subset of elements around the crack tip can be shifted by the change in crack length (Δa) in order to change the stiffness matrix slightly in the region of the crack tip [8]. The key advantage of this approach is that it can be performed in the post-processing stage,

without requiring another finite element analysis run to be performed. This method is more accurate than the displacement extrapolation method, and a good level of accuracy can be obtained with the use of standard elements. However, some numerical sensitivity is present in terms of the choice of Δa. The most significant disadvantage of this method is that it is not possible to separate the stress intensity factors of each loading mode (K_I, K_{II}, K_{III}) directly, and an additional step must be performed in order to decompose the results. This significantly reduces the utility of this method [11].

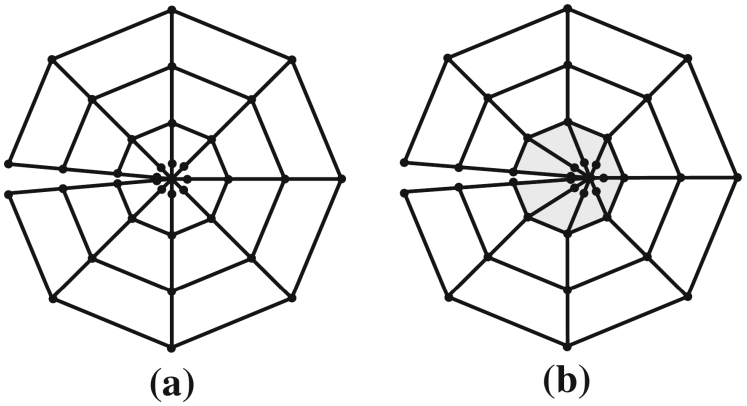

Figure 2.10: Diagram of the virtual crack extension technique. The first image (a) shows the initial crack state, while the second (b) shows the virtual extension of the crack tip elements (highlighted in colour) [11].

2.4.5 Virtual Crack Closure

The virtual crack closure technique (VCCT) was first described by Rybicki and Kanninen – again in the 1970s. It is an energy-based method which utilises Irwin's crack closure integral – which relates the energy release rate to the stresses and displacements near the crack tip – and adapts it for use as a computational method. In the simple case, this involves running two finite element analyses, whereby the crack is extended a small amount by separating the mesh near the crack tip. The work required to close the crack back to its initial state can then be determined by using Irwin's crack closure integral to calculate the energy release rate G [12]. This method was extended in order to develop the modified crack closure integral (MCCI) method [42]. With this method, only a single finite element analysis is required, with the crack in its initial state. The displacements after the crack has been extended can then be approximated via interpolation, assuming that a small extension of the crack does not significantly change the loading state at the crack tip [11] A diagram of this process is presented in Figure 2.11.

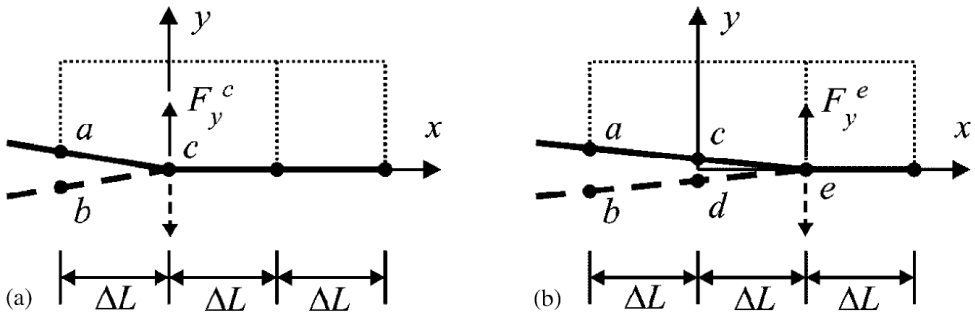

Figure 2.11: Diagram of the MCCI technique. The first analysis (a) shows the initial crack state, while the second (b) shows the state after the crack has been extended. The displacements at c and d in (b) can be approximated by using the displacements at a and b in (a), without requiring a second analysis to be performed [12].

2.4.6 Equivalent Domain Integral

As discussed in §2.3.3, the J-integral is a path-independent contour integral around the tip of a crack, which can be used to determine the stress intensity factor. The standard contour-based formulation of the J-integral is relatively difficult to implement using the finite element method. The options for defining the contour are either to define the contour through a set of nodes, or to define the contour as a circle and allow it to pass through elements when required. Defining the contour through nodes only potentially can produce a highly distorted contour unless a structured mesh is used, while defining it to pass through elements necessitates the interpolation of values within the element. Both of these approaches can lead to inaccurate results if not done carefully.

Shih et al. [43] introduced an alternative method – known as the equivalent domain integral (EDI) method – where the contour integral was converted into a domain integral using the divergence theorem. This allowed the line integral to be restated in terms of area, converting the original equation for the J-integral (Equation 2.13) into Equation 2.20.

$$J = \int_A \left[\sigma_{ij} \frac{\partial u_i}{\partial x_1} - W \, \delta_{1j} \right] \frac{\partial q}{\partial x_j} \, dA \tag{2.20}$$

- u_i is the displacement field.
- σ_{ij} is the stress tensor.
- W is the strain energy density.
- δ_{1j} is the Kronecker delta.
- q is a smooth weight function that equals 1 at the inner boundary of the integral domain, and decays to 0 at the outer boundary of the integral domain.

- A is the area over which the integration is performed. For the 3D analysis, V was substituted here instead, which was the volume over which the integration was performed.

The conversion of the contour integral into a domain integral necessitated the introduction of a smooth weight function. This is an arbitrary function that is equal to 1 at the inner boundary of the integral domain, equal to 0 at the outer boundary of the domain, and decays smoothly throughout the domain. Spreading the integral across a larger domain of elements vs a single contour reduces the sensitivity of the results to local mesh discontinuities, while the weight function acts to ensure that elements closer to the crack tip are more strongly represented within the J-integral.

The equivalent domain integral method is closely related to the virtual crack extension method discussed in §2.4.4. While the virtual crack extension method explicitly calculates the change in potential energy caused by the extension of the crack, the equivalent domain integral method implicitly performs the same calculation via the use of the weight function. The equivalent domain integral can also be extended to three-dimensions by conversion to a volume integral, which is created by extending the area integral along the crack tip, creating a cylinder over which to perform the integration. This is presented in Figure 2.12.

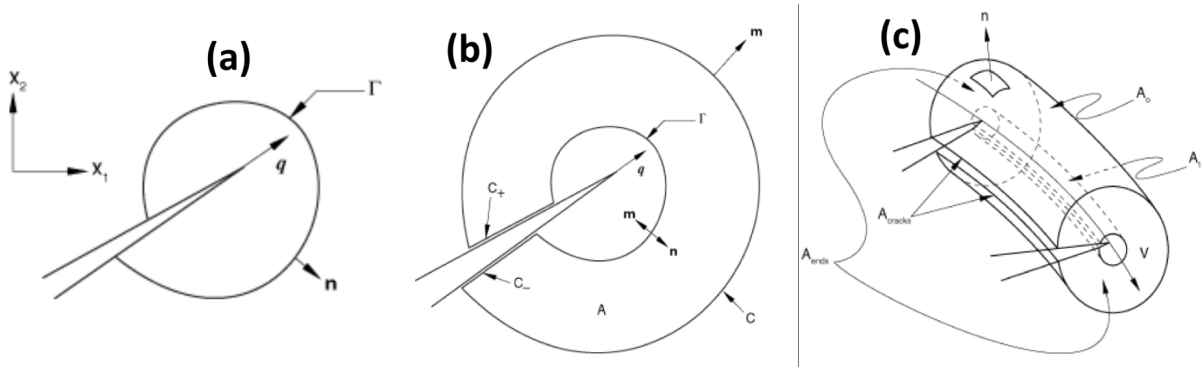

Figure 2.12: Diagram visualising the different J-integral formulations. The contour integral in 2D (a) is shown, along with the domain integral in both 2D (b) and 3D (c) [13].

2.5 Summary

This literature review began with a discussion of aerospace fatigue and damage tolerance certification requirements (§2.1), which highlighted the importance of understanding crack initiation and propagation when designing safe and reliable aircraft structures. The available methods for fatigue and damage tolerance analysis were then discussed in §2.2. The key aim of this section was to emphasise the importance of being able to accurately calculate crack propagation lives in order to set safe inspection thresholds and intervals. §2.3 then went on to review the key theoretical concepts underpinning damage tolerance analysis, and discussed key concepts such as the stress intensity factor and the J-integral.

The final section (§2.4) explained the difficulties in calculating these parameters analytically for three-dimensional crack configurations, and provided an overview of some computational methods which were able to accomplish this task. The overall project was therefore put into context. Aerospace certification requirements necessitate the requirement of accurate crack propagation lives. These can be obtained by calculating the stress intensity factor for the crack, and using as an input for the Paris-Erdogan equation. However, obtaining the stress intensity factor for three-dimensional configurations requires a laborious manual meshing process. Therefore, a need has been demonstrated for a piece of software which is capable of accurately calculating the stress intensity factor for a three-dimensional crack, using an unstructured mesh which can be generated quickly and automatically. At present, such capabilities are limited to expensive commercial software or proprietary company tools. Therefore, this project aims to fill that gap by developing such a piece of software. The implementation of this software is discussed in the following section.

Chapter 3

Implementation

This section details the implementation of the software which was developed for this project, which had the requirement of being able to accurately calculate the J-integral and stress intensity factor for a crack, using finite element analysis results generated via a commercial software package. The rationale for the methods and tools used is provided, followed by a brief summary of the architecture of the software. Finally, the implementation of the software itself is described, which is broadly split into the creation of the finite element model, the export of the analysis results from the commercial software package to a raw data format, and the calculation of the J-integral and stress intensity factor from these exported results.

3.1 Methods & Tools

3.1.1 Method Selection

As discussed in §2.4.1, the finite element method is the most commonly used method in industry for performing crack propagation calculations, and has the widest selection of commercial software, the broadest range of competent practitioners, and the most comprehensive set of validation data. Therefore, the finite element method was selected as the most suitable method for use in the implementation of this software, rather than a mesh-free method or an alternative mesh-based method. The standard element formulation was also selected – rather than an enriched formulation such as XFEM – as it has been demonstrated to be able to provide sufficiently accurate results for the stress intensity factor when using a tetrahedral mesh, while minimising additional complexity [38]. The equivalent domain integral method was selected as the most suitable method, due to its higher robustness compared to direct methods, and its ability to generate results from only a single finite element analysis run.

3.1.2 Software Selection

There are many different software packages available which are able to perform crack propagation calculations using finite element analysis. These can be broadly split into widespread, general-purpose packages, smaller and more niche packages, and in-house packages developed within engineering companies for specific purposes.

The general-purpose packages are developed by large companies, and are not sector-specific – these include NASTRAN, HyperWorks, Ansys, and Abaqus. These packages usually deal with the end-to-end process, where geometry definition, meshing, analysis, and post-processing can all be performed using the same piece of software. Modules are generally available for specialised types of analysis – such as crack propagation analysis. However, these modules are less comprehensive than those available in more niche packages, due to the one-size-fits-all approach taken. Due to the popularity of these packages and the resources available to the developers, tools are often provided which can be used by customers to extend the software further by developing custom scripts.

Other – more niche – packages are also available, which are generally tailored towards a small set of specific applications such as crack propagation. These packages utilise a more established software package for their base feature set – for example, meshing the geometry and solving the differential equations – while incorporating new features such as the ability to re-mesh automatically, or the ability to incorporate additional post-processing and visualisation of the results. Some of the available software in this category for crack propagation specifically include Zentech, BEASY, and Z-Cracks. These packages can be very useful as they provide additional features which are not available in the more common packages, and can be used off-the-shelf by customers. However, they generally aren't as extensible as the larger software packages.

For some specific applications, even niche software packages may not be able to meet the needs of a customer, and this can necessitate the development of bespoke analysis tools within an engineering company itself. These tools can be costly to develop and maintain, but can be necessary for certain applications specific to a particular company. As these tools are developed internally and contain a company's valuable intellectual property, they are generally not available to the public for use. They provide maximum flexibility, as the company is free to develop them to meet whichever needs they may require. These may be standalone tools, or extensions for commercial software packages.

The software selected for the development of the script was Abaqus (v2023). Abaqus is a finite element analysis software package that is ubiquitous within industry – particularly for non-linear structural analysis. Abaqus provides the Abaqus Scripting Interface – this is an application programming interface (API) which can be used to interact with Abaqus programatically, and perform tasks such as generating models, exporting results, and performing post-processing. Calculations can either be performed within Abaqus, or the analysis results data can be exported in order to perform the calculations outside of Abaqus.

3.1.3 Language Selection

The Abaqus API supports two programming languages – Python and C++. Python is a high-level interpreted language, which offers a relatively simple development process along with a large collection of libraries for data processing and scientific computing. The key advantage of Python is that it can shorten the development time required to produce a viable piece of software – however, as a high-level interpreted language, its performance when dealing with large-scale computations is relatively lacklustre. This can somewhat be compensated for when using high-performance libraries such as NumPy and SciPy which are written in more performant languages such as C, C++, and FORTRAN [44].

C++, on the other hand, is a compiled language that operates at a lower level than Python, and which is commonly used in the development of computationally demanding software due to its high performance and ability to manually manage memory allocation. However, software developed using C++ generally requires a longer development time when compared to Python because of its increased complexity and the absence of high-level features available in Python. An alternative approach considered was to use either Python or C++ to create the validation model and export the results from Abaqus, while using a separate language to perform the analysis. This approach would have potentially been the most flexible – however, using multiple languages would have increased the overall complexity of the program [45].

The decision was made to use Python for the entirety of the development process, including the model generation, the results export, and the analysis. This allowed simplicity to be maximised while minimising the development time and complexity. As the test cases were fairly simple, the benefits of developing in C++ would not have been fully realised, while the use of high-performance Python libraries to perform the analysis allowed the lower performance of Python to be somewhat mitigated. Using two languages for the

development was rejected as being too complex for a proof-of-concept project. If the software were eventually to be extended and used within a production environment, it would be relatively straightforward to translate the code into a higher-performance language, given that the program's logic had already been implemented and tested.

3.2 Architecture

The software was developed using an object-oriented approach whereby methods were separated into different classes – each with a separate and well-defined purpose. For the sake of simplicity, a combination of command-line arguments and a `config.ini` configuration file were used to specify the options for each run. The software was broadly split into three modular sections in order to maximise flexibility – the creation of the validation model, the export of the results, and the performance of the calculations. This approach meant, for example, that it was possible to run multiple analyses on the same set of Abaqus results – using a different set of parameters – without having to re-create and re-run the model each time. A fourth section of the code managed the running of the software itself. The four sections of the software were managed by four separate classes:

- `create_model.py` – Generates a validation model and runs an analysis job within Abaqus, using parameters defined within the `config.ini` configuration file. This step was managed by the `model.py` class.
- `export_results.py` – Exports the results from the proprietary Abaqus Output Database (ODB) file format to an open JSON format, so that the analysis could be run in a stand-alone manner, without needing Abaqus to be open.
- `analyse_results.py` – Performs all the calculations necessary to determine the J-integral and stress intensity factor of the crack using the exported results, and saves the calculation results to CSV and JSON files.
- `runner.py` – Performs administrative duties such as parsing the command line arguments, creating directories for each individual run, and calling the classes for each part of the analysis using the correct arguments.

The overall architecture of the software is presented in Figure 3.1. Further information on how to actually run the software is provided within the `readme.md` file, available in the GitHub repository for the project, and in the appendix of this document. A link to the GitHub repository is also provided in the appendix (§6).

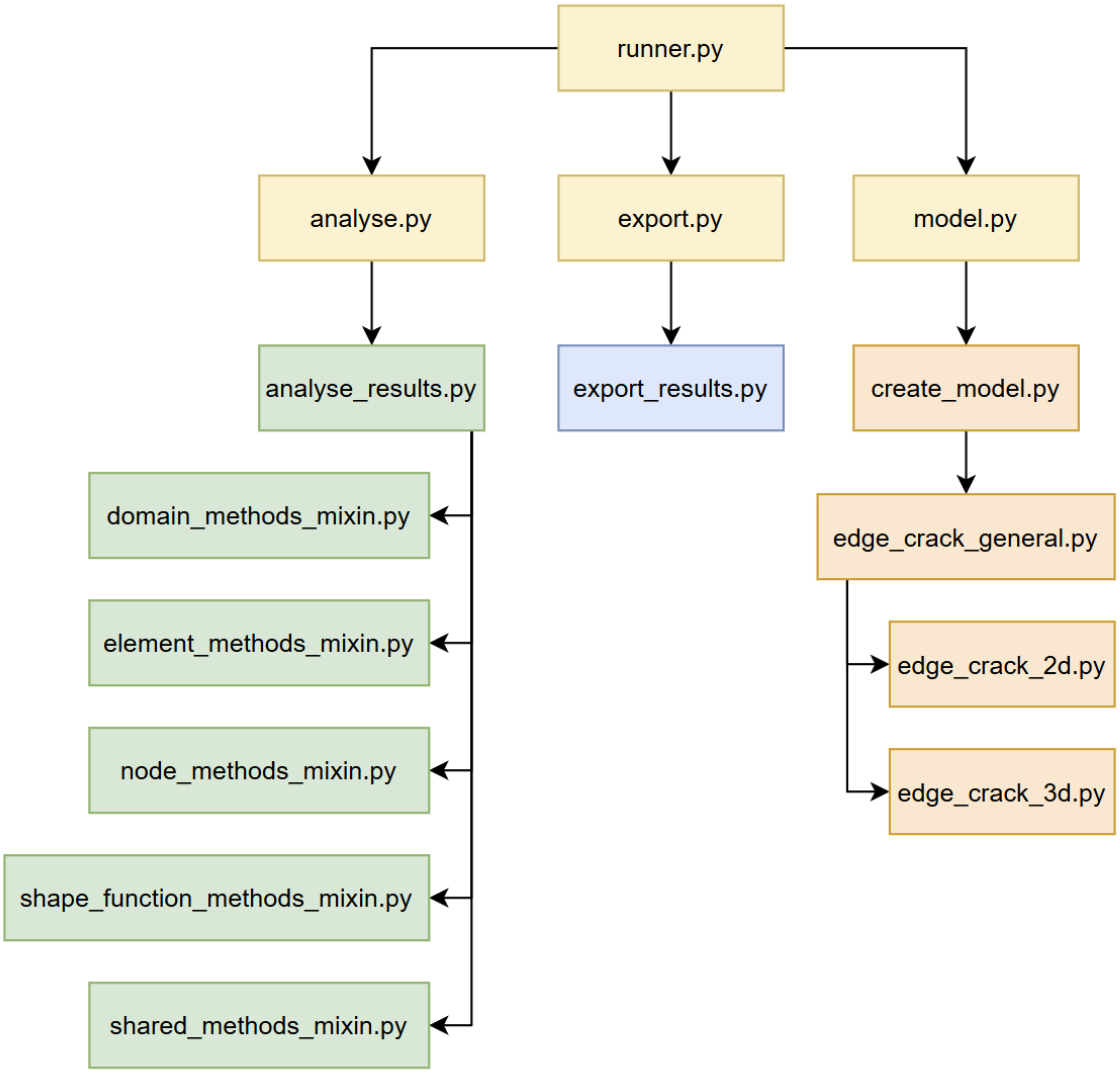

Figure 3.1: Architecture diagram of the J-integral calculation software developed for this project, demonstrating the separation of functionality into four sections. The files relating to the run manager, model creation, results export, and results analysis are in yellow, orange, blue, and green respectively.

3.3 Model Creation

As discussed in §1.3.2, a through-thickness edge crack in a finite plate was selected as the case study to demonstrate the validity of the software. This is because it was relatively simple to model, and was represented well in the literature – with both analytical and numerical solutions readily available for comparison. In order to provide maximum traceability and reproducibility of the results, Python scripts were developed which could generate the model parametrically, based on a configuration file. This allowed factors such as the applied stress, plate dimensions, crack length, and mesh density to be adjusted quickly in order to investigate the impact of these factors on the results of the analysis.

3.3.1 Units & Geometry

The set of units used for the model were N, mm, and MPa. This meant meant that the units for the J-integral were in $N\ mm^{-1}$, and the units for the stress intensity factor were in $MPa\ \sqrt{mm}$. A plate with an aspect ratio of $1:2$ (width to height) was selected, as this geometry was found to be well-represented in the literature. A unit thickness of 1.0 mm was used – for both the 2D and 3D models – in order to ensure plane stress conditions. The geometry of the plate is presented in Table 3.1, for the reference configuration. The crack length was varied between 2 mm and 25 mm as part of the analysis, in order to understand the impact of crack length on the J-integral and stress intensity factor. However, an initial crack length of 10 mm was used as the reference configuration, in order to validate the modelling approach.

Table 3.1: Reference geometry for the 2D and 3D edge-cracked models.

Height (mm)	**Width (mm)**	**Thickness (mm)**	**Crack Length (mm)**
100.0	50.0	1.0	10.0

3.3.2 Coordinate System

A common approach when performing crack-related finite element analysis is to define a separate coordinate system for the crack, so that measurements such as nodal coordinates and displacements can be described in terms relative to the crack tip. However, because this work used a model which was parametrically generated through via the Abaqus API, the global coordinate system was used instead for increased simplicity. The crack tip was placed at the origin of the global coordinate system, and the rest of the part geometry was created relative to the crack tip. A configuration parameter – `crack_tip_coordinates` – was also specified, which allowed for the use of different coordinate system origins to be more easily integrated into future updates.

3.3.3 Material

The material used for the model was not a significant parameter when calculating the stress intensity factor, as the stress intensity factor is a material-independent property which only considers the crack length, part geometry, and applied stress. Only the critical stress intensity factor – which quantifies the fracture toughness of a material – is material-specific, and this parameter was not necessary for the purposes of this work. Therefore, aluminium was arbitrarily selected as the material to be used, and was specified in Abaqus using a linear-elastic material type, along with the parameters presented in Table 3.2.

Table 3.2: Parameters used to specify the linear-elastic material in Abaqus.

Material Name	**Young's Modulus - E (MPa)**	**Poisson's Ratio - ν**
Aluminium	72,000	0.3

3.3.4 Partitioning

The model was partitioned in order to be able to increase the element size near the crack tip to maximise accuracy, while maintaining a coarse element size away from the crack tip to minimise computation time. Three horizontal partitions were first created, both above and below a central partition which contained the crack. The first partition was placed at a distance equal to the crack length, and the other partitions were placed based on a bias ratio defined in the configuration, which was used to alter their relative sizes. The central section was then partly partitioned in order to introduce the crack. A line equal to double the crack length was defined from the left edge at the middle of the central partition. This was then further partitioned into two lines of equal length via the introduction of a point at the halfway mark. Finally, a crack seam was assigned to the partition line representing the crack, to enable the nodes in that area to be separated during the analysis and allow the crack to open. Figure 3.2 shows the partitioning strategy used for the baseline 2D analysis, with a crack length of 10 mm and a horizontal partition bias of 1.5. The first yellow plane passes through the crack tip, while the second yellow plane passes through the end of the linear partition used to refine the mesh ahead of the crack tip.

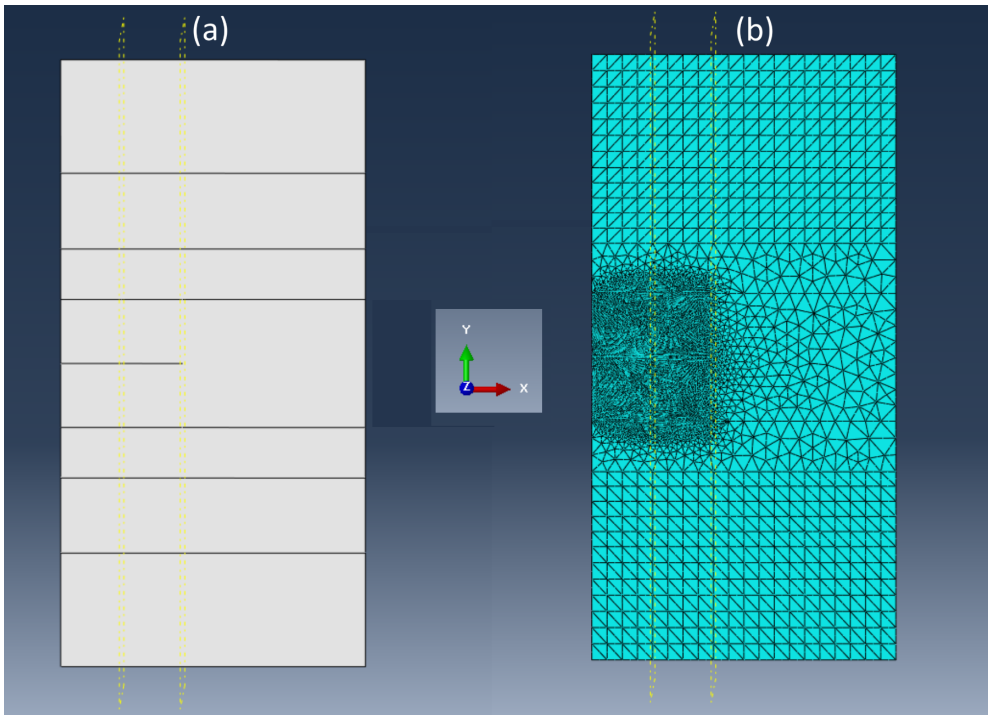

Figure 3.2: Screenshot of the 2D edge crack model within Abaqus. The first image (a) shows the partitioned geometry, while the second image (b) shows the meshed model.

3.3.5 Meshing

Initially, a radial mesh and partitioning scheme was used for the 2D model to maximise the structure of the mesh and the accuracy of the results. However, it was found that this meshing strategy did not transfer over to the 3D model, as the numerous partitions meant that the auto-mesher struggled to generate the triangular boundary elements necessary to join the separate meshes of each partition. Therefore, the meshing strategy was altered to one which was able to mesh both the 2D and 3D models automatically, while still enabling mesh refinement near the crack tip. Figure 3.3 shows the mesh seeds for the reference model. The parameter `crack_element_count` was used to define the number of elements along the biased edges, and was set at 50. The parameter `crack_element_bias` was used to set the bias ratio along the biased edges, and was set at 2. Finally, the `through_thickness_element_count` parameter was used to define the number of through-thickness elements within the crack tip partition – for the 3D model only – and was set at 5. This meant that the mesh density increased towards the crack tip, while remaining coarse in the areas remote from the crack tip. This allowed the high stress gradients near the crack tip to be captured as accurately as possible, while minimising the solution time of the model.

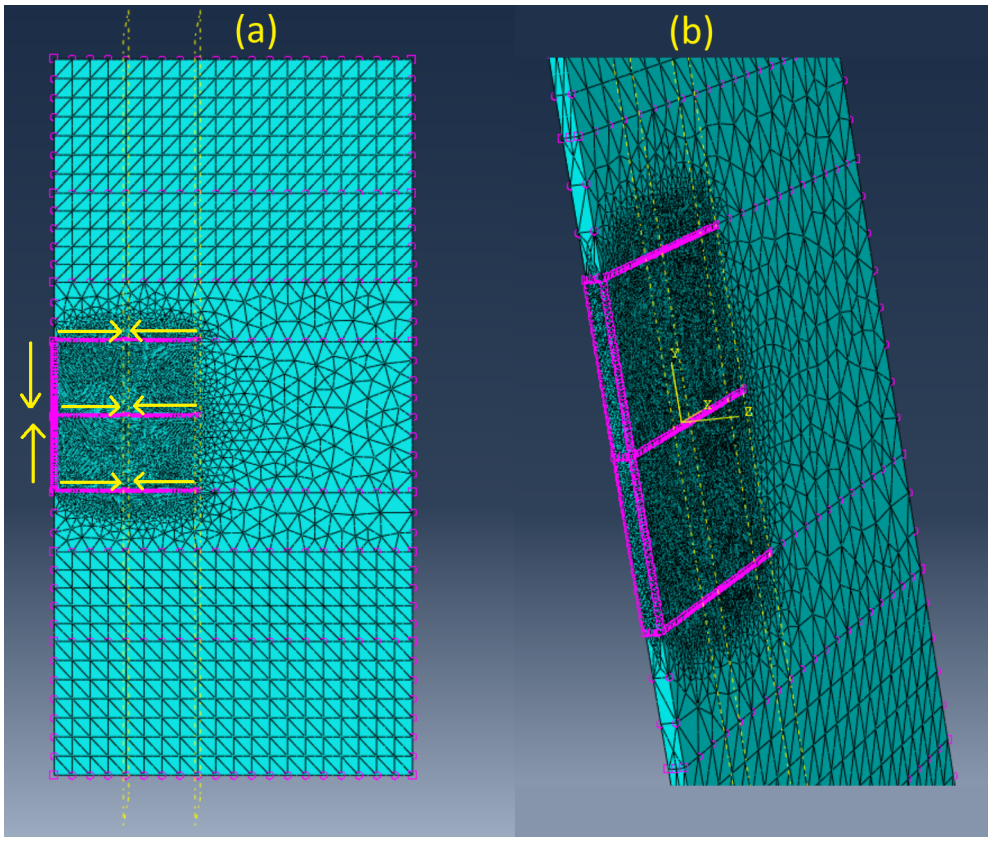

Figure 3.3: Screenshot of the 2D and 3D models within Abaqus, showing the element biasing. The yellow arrows show the biased edges, along with the direction of bias, while the unlabelled edges show the coarse mesh seeds. The first image (a) shows the biasing of the 2D model, while the second image (b) shows the biasing of the 3D model.

For the 2D model, CPS6 elements were specified, which are six-node quadratic triangular shell elements. For the 3D model, C3D10 elements were specified, which are ten-node quadratic tetrahedral elements. Quadratic elements have been demonstrated to improve accuracy when compared to linear elements – particularly in areas of high stress gradients, such as the tip of a crack [10].

3.3.6 Loading & Boundary Conditions

For the 2D model, sets were created for the top edge, the bottom edge, and the bottom-left vertex, while for the 3D model, sets were created for the top face, the bottom face, and the bottom-left edge. A fixed support was applied to the bottom-left set to constrain rigid-body motion without over-constraining the model, and a roller support was applied to the bottom set to constrain displacements in the y-direction. A uniform negative pressure load of 100 MPa was then applied to the top edge or surface, to represent the applied uniform stress, and model a pure mode 1 crack opening scenario.

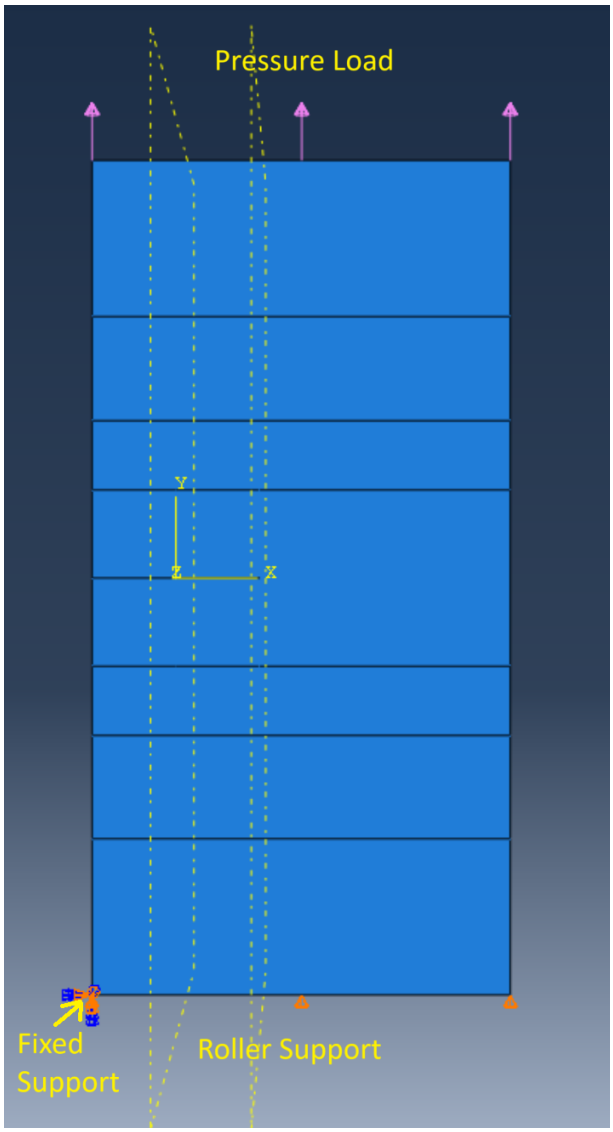

Figure 3.4: Screenshot of the 2D edge crack model within Abaqus, showing the loading and boundary conditions. A fixed support was applied to the bottom-left corner, a roller support was applied to the bottom edge, and a pressure load was applied to the top edge.

3.4 Results Export

Abaqus outputs results in the output database (ODB) format. This is a proprietary file format which requires an Abaqus license to open. Although the J-integral calculations could have been performed by directly accessing the ODB file using the Abaqus API, it was decided instead to use the Abaqus API to export the required results from the ODB to a JSON file, which could then be used to perform the calculations outside of Abaqus. This approach was selected as it maximised the flexibility of the tool – allowing the calculations to be performed without an Abaqus license, provided that a set of exported results were already available to the user. The development time for the software was reduced by simplifying the debugging process, and allowing the use of Python 3 libraries rather than being limited solely to the Python 2 libraries available within the version of Abaqus used for the analysis. The results export was split into `node_data`, and `element_data`, which were used as the top-level keys within the JSON output file. The structure of the JSON file is given below.

```
{
    "node_data": {
        "node_label":
            "coordinates_original": [],
            "displacement": [],
            "connected_elements": []
    },
    "element_data": {
        "element_label":
            "stress_tensor": {"integration_point": []}
            "strain_tensor": {"integration_point": []},
            "connected_nodes": []
    }
}
```

A more detailed description of the data stored within each key is also provided here.

- Node Data:
  - `"node_label"` - The numerical identifier assigned to the node by Abaqus.
  - `"coordinates_original"` - A 2x1 (in 2D) or 3x1 matrix (in 3D), containing the original (undeformed) coordinates of the node in the global coordinate system.
  - `"displacement"` - The 2x1 (in 2D) or 3x1 (in 3D)matrix, containing the displacement vector for the node.
  - `"connected_elements` - A list of the labels of all elements connected to the node.
- Element Data:
  - `"element_label"` - The numerical identifier assigned to the node by Abaqus.
  - `"stress_tensor"` - A 2x2 (in 2D) or 3x3 (in 3D) matrix for each integration point within the element, containing the stress tensor at that integration point, in tensorial form.
  - `"strain_tensor"` - A 2x2 (in 2D) or 3x3 (in 3D) matrix for each integration point within the element, containing the strain tensor at that integration point, in tensorial form.
  - `"connected_nodes"` - A list of the labels of all nodes connected to the element.

According to the Abaqus theory manual [46], stresses are output in their tensorial form as true (Cauchy) stresses, while strains are output in their engineering form, using engineering shear strains. It was therefore necessary to convert the engineering strain into the tensorial strain by halving the shear strain components. This was performed during the export rather than at a later stage for the sake of simplicity.

3.5 Results Analysis

3.5.1 Analysis Summary

The calculation of the J-integral and stress intensity factor was performed using the JSON file which was generated from the Abaqus ODB file during the export stage. A summary of the analysis process is provided below, while the steps are described in more detail in the following sections.

- Filter the data to retain only the elements and nodes that lie within the maximum integral domain.
- Calculate the displaced nodal coordinates for each node.
- Define the integration domains as annular regions around the crack tip.
- Compute the shape functions and shape function derivatives for the element type in the natural coordinate system.
- For each integration point of each element, calculate:
  - The coordinates in the global coordinate system.
  - The Jacobian matrix and determinant.
  - The strain energy density.
  - The displacement gradients.
  - For each domain, calculate:
    - The value and derivative of the weight function (q and $\frac{dq}{dr}$).
    - The contribution of that integration point to the total J-integral for the domain.
- Then, for each domain:
  - Sum the J-integral contribution from each element for each domain, to calculate the total J-integral.
  - Use the J-integral to calculate the stress intensity factor.
- Calculate the stress intensity factor analytically, using the analytical formula for the configuration parameter β for an edge-crack in a finite plate.
- Calculate an analytical value for the J-integral, using the analytically calculated stress intensity factor.

3.5.2 Data Filtering

The JSON file generated from the Abaqus ODB file contained data on all of the elements and nodes within the analysis, to provide maximum traceability of the results. However, only the data for elements and nodes close to or within the radius of the largest integral domain was required to perform the analysis. Therefore, the first step was to filter the data to include only the relevant elements and nodes. This improved the performance of the software by minimising the amount of data that was read into memory, and ensured that resources were not wasted on calculating parameters for irrelevant elements and nodes. The configuration parameter `domain_r_max_factor` was used to define the radius of the largest integral domain, as a fraction of the crack length. An additional factor was then applied to this to ensure that the data around the outside of the largest domain was also captured – to avoid, for example, capturing data for an element near the outside of the domain but failing to capture the data on all its connected nodes.

3.5.3 Displaced Nodal Coordinates

The nodal data obtained from Abaqus included the original coordinates for each node, along with their displacement vectors. The displaced coordinates of each node were required for further calculation steps – therefore, they were calculated using Equation 3.1.

$$\boldsymbol{x'} = \boldsymbol{x} + \boldsymbol{u} \tag{3.1}$$

- $\boldsymbol{x'}$ is a vector of size $(n_{dim}, 1)$ containing the displaced nodal coordinates.
- $\boldsymbol{x}$ is a vector of size $(n_{dim}, 1)$ containing the original nodal coordinates.
- $\boldsymbol{u}$ is a vector of size $(n_{dim}, 1)$ containing the nodal displacements.
- n_{dim} is the number of dimensions for the model.

3.5.4 Integral Domain Definition

Two approaches for defining the integration domains were evaluated. Both approaches used multiple domains of gradually increasing size, using the parameters `domain_r_min`, `domain_r_max`, and `number_of_domains` to define the minimum and maximum radii of the domain, along with the total number of domains to use. The first domain for both approaches was a circle or cylinder (for the 2D and 3D models respectively) of radius r_{min}, which included all of the elements around the crack tip. Further domains were then generated – with gradually increasing radii – until $n_{domains}$ domains had been generated, with a final domain radius of r_{max}. The reason for using multiple domains for each analysis was to verify the domain-independence of the results. If the equivalent domain integral method had been implemented correctly, then minimal differences between the J-integral

values obtained from each domain should have been observed. This was a key step in validating the outputs of the software.

The first domain approach utilised circles of increasing radius – i.e. domain 2 included all of the integration points present in domain 1, along with additional integration points. This approach was found to be less accurate, as the elements very near to the crack tip were very highly represented – becoming increasingly so as the domain size increased due to the use of the weight function. As standard (rather than quarter-point) elements were used at the crack tip, this approach was found to be very dependent on the mesh density near the crack tip.

The second approach utilised annular rings of increasing radius – rather than circles – meaning that each integration point only appeared in a single domain, with the crack tip elements only being represented within the first domain. This approach reduced the dependence of the results on the crack tip elements, since they only appeared in the innermost domain. This allowed the J-integral to be calculated using the more accurate stresses further away from the crack tip, and therefore improved the accuracy of the results. A comparison of the two approaches is presented in Figure 3.5.

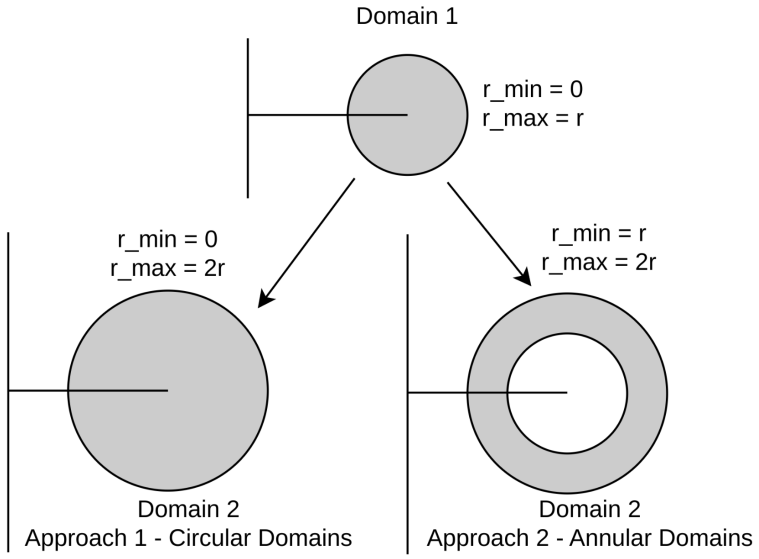

Figure 3.5: Diagram showing the two tested domain definition approaches. Domain 1 is the same for both approaches. Domain 2 for Approach 1 expands the domain into a larger circle, while Domain 2 for Approach 2 instead uses an annular ring around domain 1.

Elements were assigned to specific domains based on their nodal coordinates. Effectively, if a node lay within the domain boundaries, then all elements attached to that node were assigned to that domain. This meant that elements could appear in more than one domain – for example, if two nodes of that element each lay in separate domains. However, the use of the weight function (discussed in §3.5.10) ensured that each integration point was only represented in the J-integral calculation for a single domain. An integration point that was assigned to a domain but that did not actually lie within the boundaries of that domain would be assigned a weight of zero, and would therefore not affect the calculation.

This method produced accurate results by allowing the use of stresses directly from the integration points, while avoiding any need to interpolate the stresses to other points within the element. The basic logic used to assign elements to each domain is described in the pseudo-code below.

```
Define each annular domain based on a minimum and a maximum radius

    For each element:
        For each node attached to that element:
            Get the distance between the node and and the crack tip
            For each domain:
                If the node lies within the domain:
                    Add the element to the domain

Filter each domain to ensure each element only appears once per domain
```

Figure 3.6 presents a visualisation of the assignment of integration points to a domain. Elements 1, 2, and 3 are all included in the domain since at least one node of each element lies within the domain boundary. However, the weight function means that the integration points that fall outside of the domain boundary (in red) do not contribute to the J-integral calculation, while the integration points within the domain (in green) do contribute.

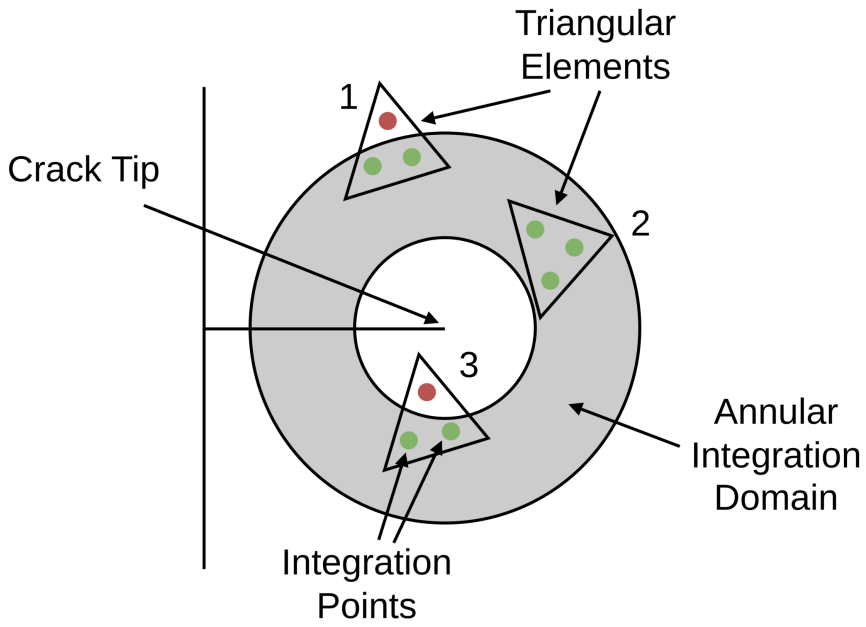

Figure 3.6: Diagram showing the filtering of integration points within a domain. All of the integration points of element 2 are included in the domain, while elements 1 and 3 have one integration point that falls outside the outer and inner boundaries of the domain respectively.

3.5.5 Shape Functions & Derivatives

Element shape functions are necessary in order to interpolate the field variables – for example displacements, stresses, and strains – at any point inside the element using the nodal data. For this project, they were used to determine the coordinates of the element integration points, which allowed the calculation of their weights. The derivatives of the shape function derivatives were also necessary to calculate the Jacobian matrices and determinants at each integration point and the displacement gradients. The element shape function expressions were obtained from the Abaqus Theory Manual [47], while the integration point coordinates in the natural coordinate system were obtained from the Code_Aster documentation [48].

The SymPy library was used to create symbolic representations of the shape functions in the natural coordinate system. These were then differentiated in order to obtain the shape function derivatives – also in the natural coordinate system. Finally, the expressions were evaluated using the natural points for each integration point in order to obtain the coordinates of each integration point in the natural coordinate system. These values were saved within an array that could be used along with the specific nodal coordinate data for each element in future calculation steps. Figure 3.7 shows the nodal numbering for the CPS6 and C3D10 elements utilised within this project.

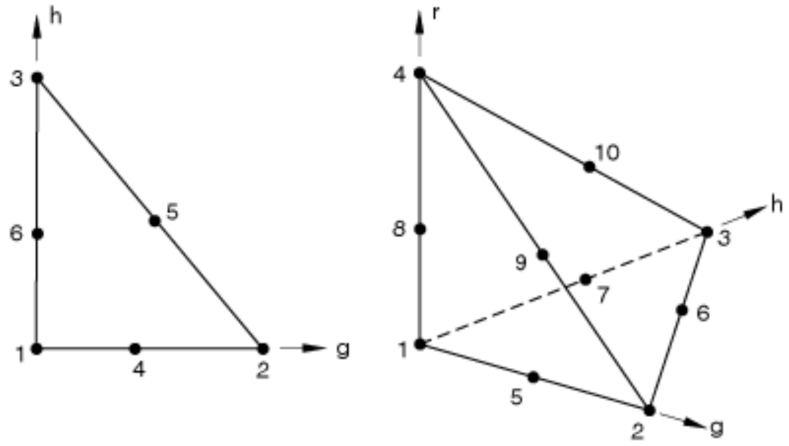

Figure 3.7: Diagram showing the node numbering for the second order triangular and tetrahedral elements within Abaqus (CPS6 and C3D10 respectively).

3.5.6 Integration Point Coordinates

It was necessary to determine the coordinates of each integration point in order to calculate their weights – as the weight was based on the distance between the integration point and the inner radius of the domain. The shape functions calculated in the previous step defined the integration point coordinates in the natural coordinate system. These were then combined with the nodal coordinates for each element in order to define the integration point coordinates in the global coordinate system, using Equation 3.2.

$$\boldsymbol{p} = \boldsymbol{S} \cdot \boldsymbol{n} \tag{3.2}$$

- $\boldsymbol{p}$ is a matrix of shape (n_{ip}, n_{dim}) containing the integration point coordinates for the element.
- $\boldsymbol{S}$ is a matrix of shape (n_{ip}, n_{nodes}) containing the numerical evaluations of the shape functions in the natural coordinate system for the element.
- $\boldsymbol{n}$ is a matrix of shape (n_{nodes}, n_{dim}) containing the nodal coordinates for the element.
- n_{nodes}, n_{ip}, and n_{dim} are the number of nodes per element, the number of integration points per element, and the dimensions of each element respectively. These are $(6, 3, 2)$ for CPS6 elements, and $(10, 3, 3)$ for C3D10 elements.

3.5.7 Jacobian Matrices & Determinants

The Jacobian matrix gives the mapping between a set of coordinates in the natural (reference) coordinate system and the same relative coordinates within the global (deformed) coordinate system – it effectively maps the shape of the undeformed element to the shape of the deformed element. The Jacobian determinant is simply the determinant of the Jacobian matrix, and gives the ratio of the area or volume of the undeformed element to that of the deformed element. As the formulation for J-integral is an integration over dA or dV (for 2D and 3D elements respectively), the Jacobian determinant was therefore necessary in order to perform the integration – as using the undeformed areas or volumes would introduce inaccuracies. The Jacobian matrix was calculated using a similar equation to Equation 3.2, but using the shape function derivatives in place of the shape functions themselves. This equation is presented in Equation 3.3. The Jacobian determinant was then trivial to calculate, by taking the determinant of $\boldsymbol{J}$.

$$\boldsymbol{J} = \frac{\partial \boldsymbol{N}}{\partial \boldsymbol{\xi}} \cdot \boldsymbol{n} \tag{3.3}$$

- $\boldsymbol{J}$ is a 3D tensor of shape $(n_{ip}, n_{dim}, n_{dim})$ containing the Jacobian matrix for each integration point within the element.
- $\frac{\partial \boldsymbol{N}}{\partial \boldsymbol{\xi}}$ is a 3D tensor of shape $(n_{ip}, n_{nodes}, n_{dim})$ containing the numerical evaluations of the shape function derivatives in the natural coordinate system for the element.
- $\boldsymbol{n}$ is a matrix of shape (n_{nodes}, n_{dim}) containing the nodal coordinates for the element.
- n_{nodes}, n_{ip}, and n_{dim} are the number of nodes per element, the number of integration points per element, and the dimensions of each element respectively. These are $(6, 3, 2)$ for CPS6 elements, and $(10, 3, 3)$ for C3D10 elements.

3.5.8 Strain Energy Density

The strain energy density was calculated for each integration point of each element, using the stress and strain tensors extracted from Abaqus for that integration point. This was accomplished using Equation 3.4

$$W = 0.5\ \boldsymbol{\sigma}_{ij}\ \boldsymbol{\epsilon}_{ij} \tag{3.4}$$

- $\boldsymbol{\sigma}_{ij}$ is the stress tensor
- $\boldsymbol{\epsilon}_{ij}$ is the strain tensor

3.5.9 Displacement Gradients

The displacement gradients represent the spatial derivatives of the displacement field, and effectively quantify how the displacement varies throughout an element. They are later combined with the stress tensor to determine the work done within the material as part of the J-integral expression. The first step in calculating the displacement gradients was to convert the shape function derivatives were to the global (deformed) coordinate system, using Equation 3.5

$$\frac{\partial \boldsymbol{N}}{\partial \boldsymbol{x}} = \left(\boldsymbol{J}^{-1} \left(\frac{\partial \boldsymbol{N}}{\partial \boldsymbol{\xi}} \right)^{T} \right)^{T} \tag{3.5}$$

- $\frac{\partial \boldsymbol{N}}{\partial \boldsymbol{x}}$ is a matrix of shape (n_{dim}, n_{dim}), containing the shape function derivatives in the global (deformed) coordinate system for each integration point.
- $\frac{\partial \boldsymbol{N}}{\partial \boldsymbol{\xi}}$ is a matrix of shape (n_{nodes}, n_{dim}) containing shape function derivatives in the natural (reference) coordinate system for each integration point.
- $\boldsymbol{J^{-1}}$ is a matrix of shape (n_{dim}, n_{dim}), and the inverse of the Jacobian matrix $\boldsymbol{J}$.

The displacement gradients at each integration point were then calculated by combining the shape function derivatives in the global (deformed) coordinate system with the nodal displacements for each element, using Equation 3.6.

$$\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}} = \boldsymbol{U}^{T} \cdot \frac{\partial \boldsymbol{N}}{\partial \boldsymbol{x}} \tag{3.6}$$

- $\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}}$ is a matrix of shape (n_{dim}, n_{dim}) representing the displacement gradient at the integration point.

- $\boldsymbol{U}$ is a matrix of shape (n_{nodes}, n_{dim}) containing nodal displacement values.
- $\frac{\partial \boldsymbol{N}}{\partial \boldsymbol{x}}$ is a matrix of shape (n_{nodes}, n_{dim}) containing global shape function derivatives.

3.5.10 Weight Functions

Weight functions are used within the domain integral method in order to apply different weights to each element or integration point within the domain, thereby increasing or decreasing the significance of their contribution relative to that of other elements or integration points. The actual weight function used can be arbitrary, providing that it is a smooth function that is equal to one at the inner radius of the domain, and is equal to zero at the outer radius of the domain. Three weight functions were used within the analysis in order to perform a comparison – linear, polynomial, and Gaussian functions were used. The first step was to define the normalised distance of each integration point within the domain, using the parameters presented in Figure 3.8.

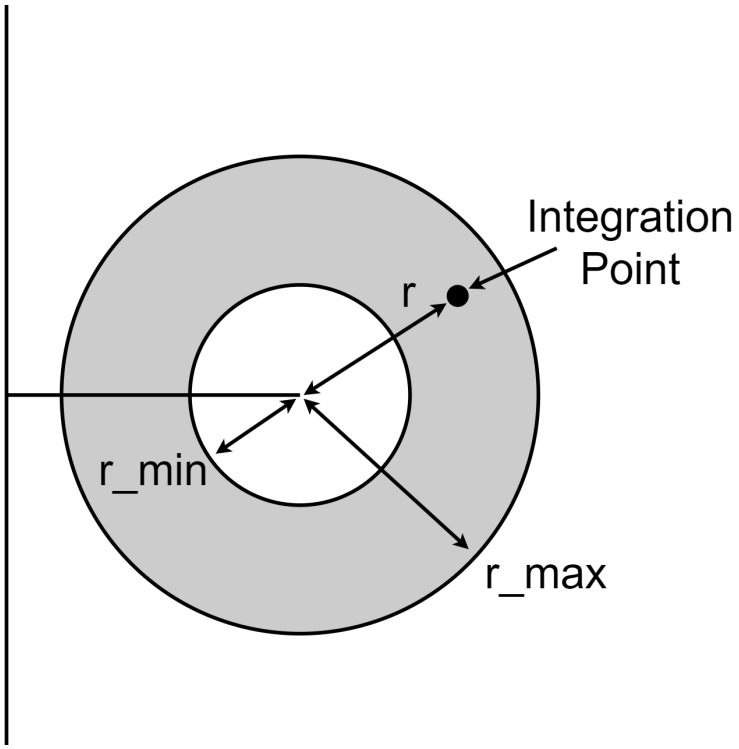

Figure 3.8: Diagram showing the measurements used to calculate the weight of each integration point using the weight function. The crack is the horizontal line perpendicular to the left edge of the part, while the grey-shaded annular region is the domain.

The normalised distance was effectively the relative position of the integration point within the domain – for example, a normalised distance of 0.75 would signify that the integration point was 75% of the way through the domain. This distance was defined as s, and was calculated using Equation 3.7.

$$s = \frac{r - r_{min}}{r_{max} - r_{min}} \tag{3.7}$$

The normalised distance was then used to calculate the weight functions using the equations below. The derivatives of each function were also calculated by hand and defined as variables within the software. The linear weight function and its derivatives are given in Equations 3.8 and 3.9.

$$q = 1 - s \tag{3.8}$$

$$\frac{dq}{dr} = \frac{-1}{r_{max} - r_{min}} \tag{3.9}$$

A polynomial of degree 3 was selected for the polynomial weight function. The polynomial weight function and its derivatives are given in Equations 3.10 and 3.11.

$$q = 1 - 3s^2 + 2s^3 \tag{3.10}$$

$$\frac{dq}{dr} = \frac{-6s + 6s^2}{r_{max} - r_{min}} \tag{3.11}$$

Finally, a Gaussian weight function was specified with a constant of 2.5, which was found to provide the best agreement with published results for the J-integral. The Gaussian weight function and its derivatives are given in Equations 3.12 and 3.13.

$$q = \mathrm{e}^{-2.5\mathrm{s}^2} \tag{3.12}$$

$$\frac{dq}{dr} = \frac{-2.5s \cdot \mathrm{e}^{-2.5\mathrm{s}^2}}{r_{max} - r_{min}} \tag{3.13}$$

3.5.11 J-Integral Calculation

The previously calculated values were then combined in order to calculate the J-integral for each domain. The domain formulation of the J-integral was first presented in Equation 2.20. However, this continuous equation could not be directly implemented using the finite element analysis results, so it was therefore necessary to convert it to a discrete form, which allowed summation over each element and integration point. The discrete form is given in Equation 3.14.

$$J = \sum_{e=1}^{e=n_e} \sum_{i=1}^{i=n_{ip}} \left[\sigma_{ij} \frac{\partial u_i}{\partial x_1} - W \, \delta_{1j} \right] \frac{\partial q}{\partial x_j} \cdot \det J \cdot w_{ip} \tag{3.14}$$

- n_e is the number of elements within the domain.
- n_{ip} is the number of integration points within each element (excluding integration points which sat outside the domain).
- $\det J$ is the Jacobian determinant at the integration point.
- w_{ip} is the integration point weight, obtained from the element formulation ($\frac{1}{6}$ for CPS6 elements and $\frac{1}{24}$ for C3D10 elements [48]).

During the evaluation of the integral, the actual area of the element was not necessary – it's impact was accounted for by the Jacobian determinant and the integration point weights. The Jacobian determinant described the ratio between the original area of the element and the deformed area, and the integration point weight described the ratio between the area accounted for by the integration point, and the total area of the reference element. For example, a triangular CPS6 element has a reference area of $\frac{1}{2}$, given that a triangle occupies half of the area of a 1×1 unit square. Since the CPS6 element contains three integration points, the weight of each integration point is therefore $\frac{1}{2} \times \frac{1}{3} = \frac{1}{6}$.

3.5.12 Stress Intensity Factor Calculation

As the plate used for the analysis was 100 mm in height, 50 mm in width, and 1 mm in thickness, an assumption of plane stress could be made. Equally, as the plate was subjected to purely tensile loading, the assumption that only Mode I crack opening occurred could also be made. The equation for the stress intensity factor was presented in Equation 2.6. An analytical solution was obtained for the configuration factor β_i for an edge crack in a finite-width plate with a uniform tensile load, and is presented in Equation 3.15 [49].

$$\beta_I = 1.12 - 0.23 \left(\frac{a}{b}\right) + 10.6 \left(\frac{a}{b}\right)^2 - 21.7 \left(\frac{a}{b}\right)^3 + 30.4 \left(\frac{a}{b}\right)^4 \tag{3.15}$$

- a is the crack length,
- b is the plate width

This configuration factor could then be substituted into Equation 2.6 in order to obtain the equations for K_I specific to the analysed configuration, which was dependent on the crack length. A function was created to calculate the stress intensity factor via the analytical equation – as well as via the J-integral obtained from the analysis – so that the values could be compared. A percentage error was also calculated.

Chapter 4

Results

This section discusses the results of the case study used to validate the software developed for this project, using 2D and 3D models of an edge crack in a finite plate. The parameters of both the 2D and 3D models were adjusted in order to perform finite element analyses for varying mesh densities, integration domains, weight functions, and crack lengths. The software was then used to export the results of these finite element analyses, and then to calculate the J-integrals and stress intensity factors for each set of parameters. This allowed the impact of these parameters on the results to be understood, and therefore used to demonstrate that the software implemented the equivalent domain integral method correctly. The results obtained via the software were also compared to analytically calculated results to validate that the J-integral and stress intensity factors produced were correct.

4.1 Mesh Convergence

A mesh convergence study was conducted in order to find the point at which the results obtained from the finite element analysis became independent of the density of the mesh used. Mesh convergence studies are a common practice when performing finite element analysis, as they allow the competing demands of solution accuracy and computational resources to be balanced in order to obtain sufficiently accurate results as efficiently as possible [6]. In order to perform the mesh convergence study, the configuration parameter `crack_element_count` was used to vary the size of the elements located close to the crack tip. A function was also created to determine the average element size directly around the crack tip, in order to obtain a more direct measurement of the element size rather than simply using the number of elements.

The mesh convergence study was limited to the area close to the crack tip, since this was the area that was being used to calculate the J-integral. A coarse mesh of 2.5 mm elements was considered sufficient to model the remote areas of the problem, given the simple loading and boundary conditions used. Although a stress value is usually used as the value for comparison, the options for stress in this case were to use a stress value remote from the crack tip, or use the stresses close to the crack tip. The remote stresses were not relevant to the actual solution, while the crack tip stresses were prone to significant variation due to the nature of the crack tip singularity. Therefore, the J-integral was used to measure the mesh sensitivity, as it was the most relevant parameter to the specific problem being analysed. An initial investigation of the results found that the solution maintained domain independence across various configurations (discussed in the following section) – therefore, the mesh convergence study was conducted using solely the fifth annular domain from the crack tip.

Figure 4.1 presents the results of the mesh sensitivity study, by plotting the J-integral against the crack tip element size. Significant deviation was observed in the results between crack tip elements of size 1.5 mm and 0.6 mm. However, the results converged at a crack tip element size of around 0.5 mm. The crack tip element size used for the remainder of the analysis was 0.461 mm, which corresponded to 15 elements along the 10 mm crack (`crack_element_count`), with a bias ratio of 2 (`crack_element_bias`).

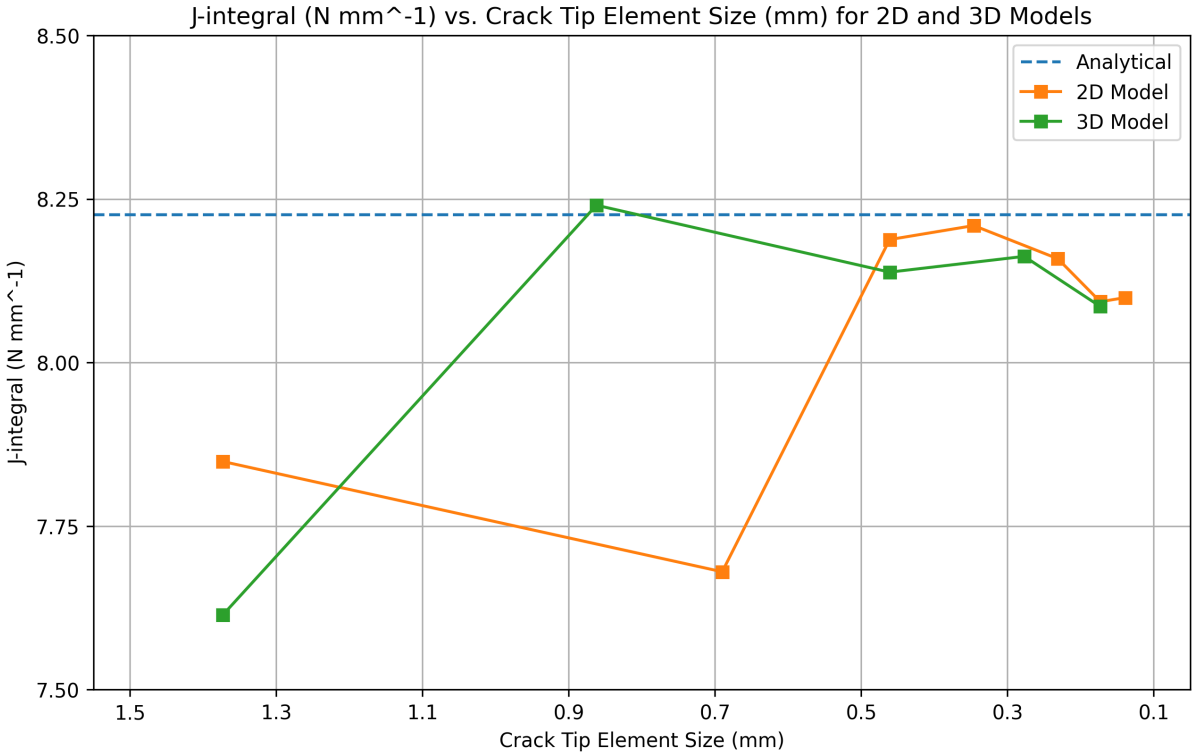

Figure 4.1: J-integral ($N\ mm^{-1}$ vs crack tip element size (mm) for a 10 mm edge crack).

4.2 Domain Independence

A key feature of the J-integral is the demonstration of path-independence – that is, the calculated value should be independent of the path around the crack which was selected. This equally applies to domain selection when using the equivalent domain integral method. Therefore, demonstrating that the J-integral values calculated by the software were domain-independent was critical in order to validate that the method had been implemented correctly.

To accomplish this, the software was implemented such that multiple non-overlapping domains were defined, and a separate J-integral was calculated for each domain (as described in §3.5.4). This allowed the domain independence of the results to be verified by comparing the J-integral results obtained for each domain. The J-integral results for each domain – for the reference model with a crack length of 10 mm – were recorded and plotted , along with a analytical value obtained via the analytical expression for the stress intensity factor (Equations 2.6 and 3.15). This is presented in Figure 4.2.

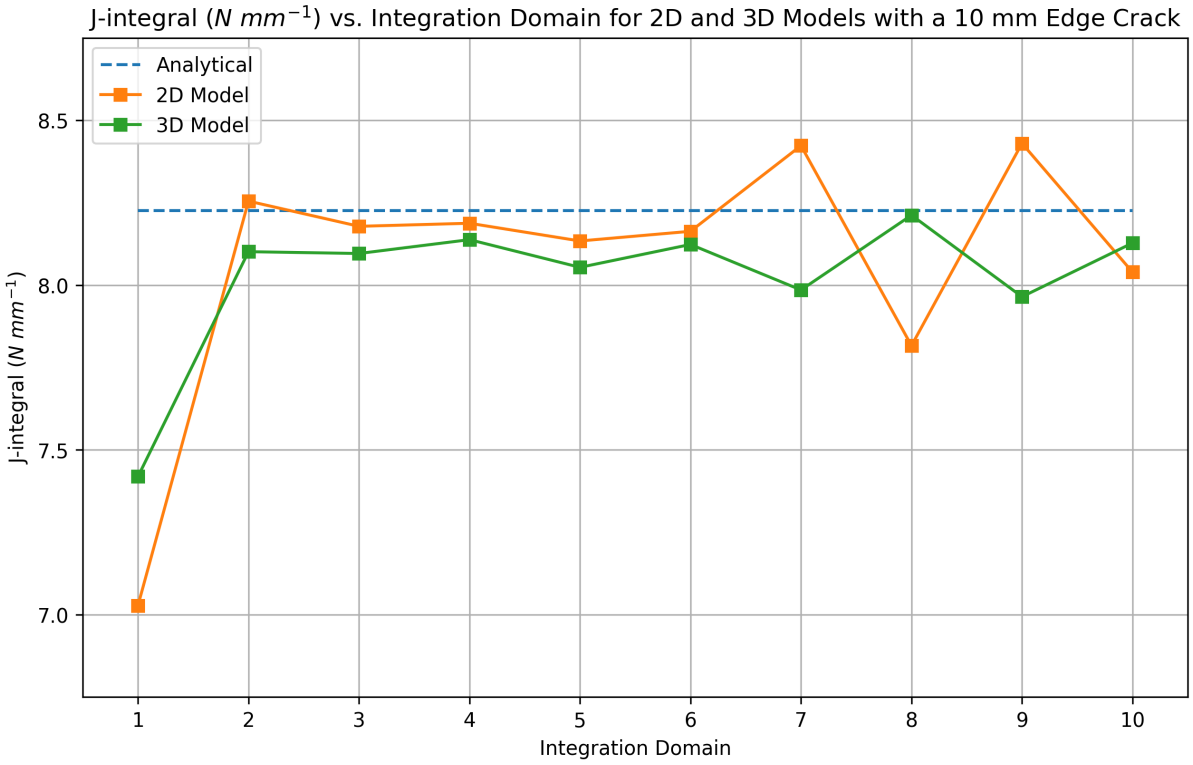

Figure 4.2: J-integral ($N\ mm^{-1}$ vs integration domain for a 10 mm edge crack).

Overall, the J-integral definitively exhibited domain independence beyond the first domain, with maximum deviations of under 2% from the mean. The J-integral calculated for the first domain (the circular domain enclosing the crack tip) showed the most deviance from the rest of the results – being around 6% less than the mean value. This was likely due to the fact that the first domain included the crack tip elements, which exhibited significant stress gradients due to the singularity present at the crack tip. It has been demonstrated that quarter-point tetrahedral elements are able to accurately model the crack tip singularity [38] – these are tetrahedral elements where the mid-side nodes of the crack tip elements are moved closer to the crack tip, to sit at the quarter-point rather than the mid-point of the element edges. Quarter-point elements were implemented for two-dimensional triangular elements as part of this project, and the use of these elements was added as a configurable parameter – however, they were not found to significantly alter the J-integral results obtained for the first domain.

4.3 Weight Functions

Several weight functions were investigated during the development of the software, in order to ensure that a weight function was selected which could accurately model the problem. As discussed in §2.4.6, the equivalent domain integral method utilises an arbitrary weight function, which begins at a value of 1 at the inner radius of the domain, and smoothly decays to a value of 0 at the outer radius of the domain. Three weight functions were selected for the final comparison – a linear function, a third-degree polynomial function,

and a Gaussian function – as described in §3.5.10. A plot was created for these weight functions in order to visualise their effects relative to the distance through the domain – this is presented in Figure 4.3. This figure shows how the weightings of elements vary based on their distance through the domain, with a distance of 0 signifying that an element integration point lies on the inner boundary of the domain, and a distance of 1 signifying that an element integration point lies on the outer boundary of the domain.

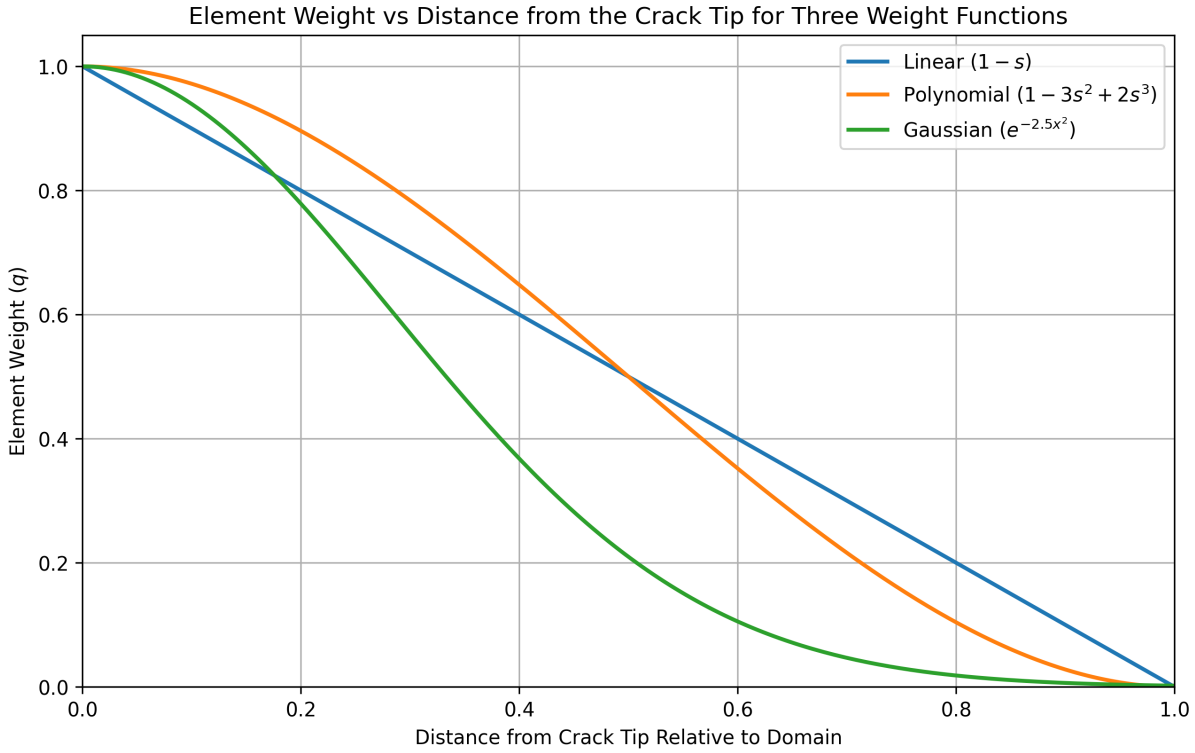

Figure 4.3: Element weighting vs the distance of the element from the crack tip – relative to the domain – for the three weight functions investigated.

The linear weight function naturally exhibited a steady decrease of element weight along with the distance through the integration domain, and acted as the base of comparison for the other weight functions. The polynomial weight function placed the highest weighting on integration points close to the crack tip compared to the other weight functions, before intersecting the linear weight function at a relative distance of 0.5 before exhibiting a weights between those of the linear and Gaussian weight functions for integration points with relative distances of between 0.5 and 1. The Gaussian weight function placed a relatively high emphasis on the integration points close to the crack tip, before decaying very quickly and specifying element weights below those of the linear and polynomial weight functions for relative distances above around 0.2.

The J-integral obtained using each weight function for the reference configuration was then again plotted against the integration domain, for both the 2D and 3D models – this is presented in Figure 4.4. All of the three weight functions showed good agreement with each other beyond the first domain, with minimal variation being observed between each weight function for the 3D model. The linear and Gaussian weight functions for the 2D model exhibited some variation at the outer domains (between domains 6 and 10), while

those same weight functions exhibited better agreement for the 3D model. This was likely due to numerical fluctuations, as the deviance did not exhibit a bias towards errors above or below the mean, and the errors between the linear and Gaussian weight functions were also different.

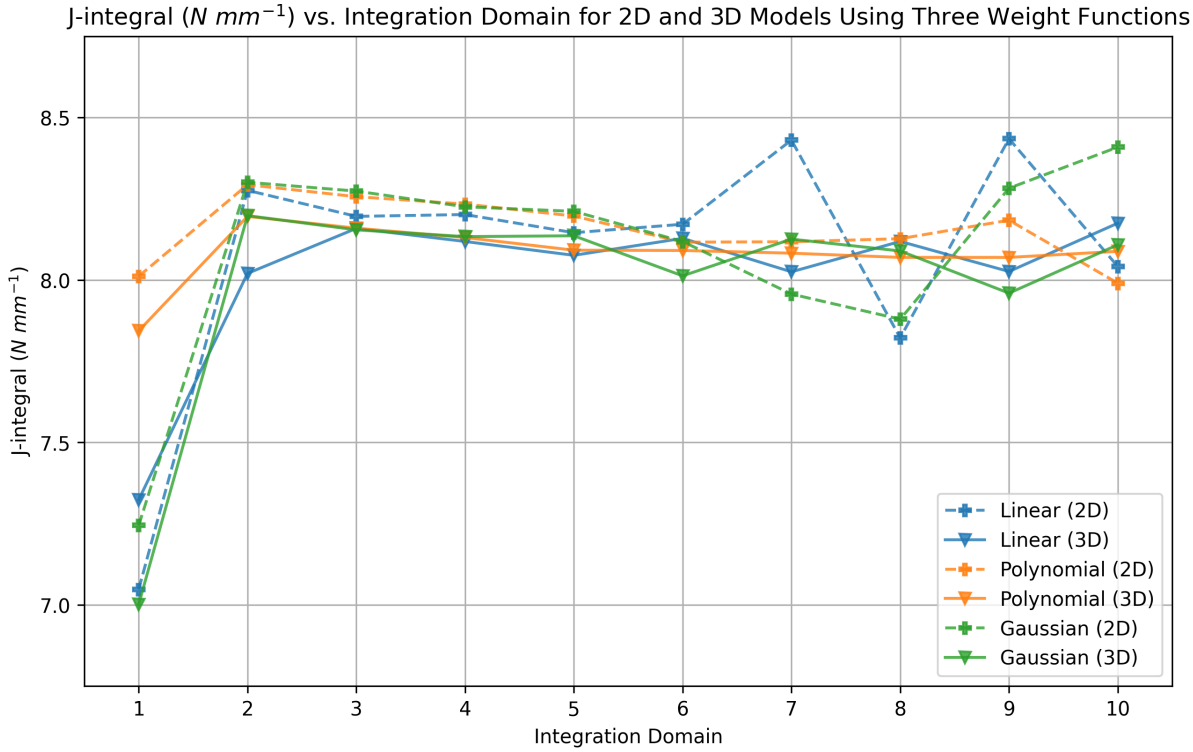

Figure 4.4: Element weighting vs the distance of the element from the crack tip – relative to the domain – for the three weight functions investigated.

The main difference between the weight functions was exhibited in the first domain, where the polynomial weight function produced J-integrals close to the analytical value of around 8.2, while the linear and Gaussian weight functions produced significantly lower J-integrals of between 7 and 7.5. This signified that the linear and Gaussian weight functions were potentially underestimating the contributions of the crack tip elements within the first domain. As the polynomial weight function decayed the most slowly of the three weight functions, it would have therefore placed the highest weights on the crack tip elements within the first domain. This increased weighting of the inner elements likely had a lesser impact within domains 2-10, but had a significant contribution within the first domain, which would have been dominated by the effects of the crack tip elements.

The polynomial weight function also showed the greatest amount of agreement between the 2D and 3D models - with the J-integral values showing close agreement for every domain. This was not the case for the linear and Gaussian weight functions, which exhibited somewhat significant disagreements between the results for the 2D and 3D models, for the outer domains. As the polynomial weight function produced the most domain-independent results while simultaneously exhibiting the closest agreement between the 2D and 3D models, it was selected as the most suitable weight function to perform the rest of the analysis. Although additional validation would be necessary to fully demon-

strate the superiority of the polynomial weight function (for example, testing it against multiple crack lengths and configurations), the validation performed was considered to be satisfactory for this application.

4.4 Crack Length

Finally, once the mesh density, domains, and weight functions had been validated, the impact of the crack length on the J-integral and the stress intensity factor was investigated. Equation 2.14 demonstrates that the J-integral is proportional to the square of the stress intensity factor, while Equation 2.6 demonstrates that the stress intensity factor is proportional to the square root of the crack length. Therefore, assuming a constant shape factor B_i, it would be expected that the stress intensity factor would exhibit a linear relationship with the crack length, while the J-integral would exhibit a second-degree polynomial relationship with the crack length. However, for the specific edge-crack configuration analysed, the shape factor exhibited a non-linear relationship with the crack length. Taking Equation 3.15 and plotting it between a crack length of 0 mm and 25 mm for a plate with a constant width of 50 mm demonstrates the relationship between the crack length and the shape factor β_i, which is presented in Figure 4.5.

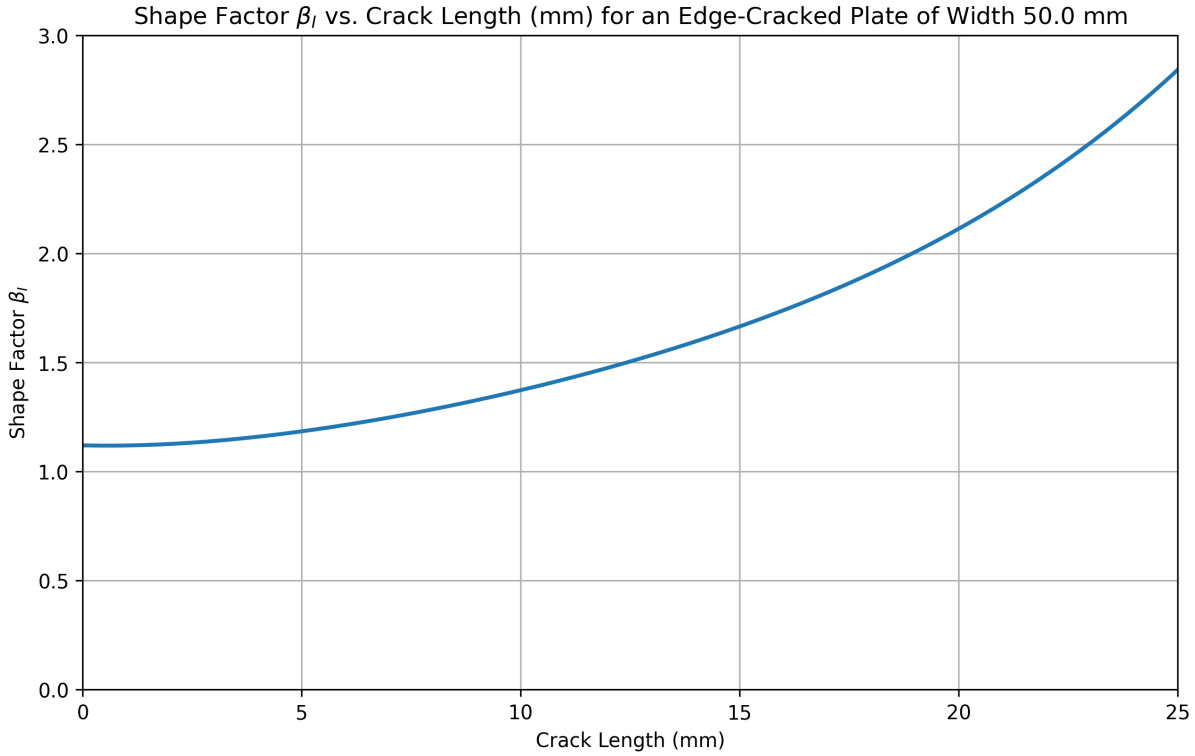

Figure 4.5: Shape Factor (β_i) vs Crack Length (mm) for an Edge-Cracked Plate with a Width of 50 mm.

Figure 4.5 clearly demonstrates that the shape factor β_i increases as the crack length increases (and therefore the ratio of the crack length to the plate width $\frac{a}{b}$ also increases).

This is because as the crack grows longer, there is a smaller remaining ligament (uncracked plate ahead of the crack tip) to carry the load – as the plate is of a finite width. Therefore, there is less material available to carry the remote tensile stress applied to the plate, and the influence of the free surface ahead of the crack becomes increasingly more significant [8]. As the stress intensity factor is directly proportional to β_i, it would therefore be expected that the relationship between the J-integral and the crack length, and the stress intensity factor and the crack length, would be of a higher-order than would be expected for a centre-cracked plate. It can also be observed that the plotted line crosses the y-axis at a value of 1.12 – this is the shape factor for the special case of an edge crack in a semi-infinite body, where the shape factor remains constant regardless of the crack length due to the infinite ligament available ahead of the crack tip [49].

A comparison was then performed between the J-integral results obtained from the outputs generated by the software for the 2D and 3D models, and the analytically obtained values – using crack lengths between 2 mm and 25 mm. This is presented in Figure 4.6, which demonstrates the expected relationship between the J-integral and the crack length – with the J-integral increasing as the crack length increases. The J-integral displayed a non-linear relationship with the crack length, with a polynomial of degree 4 showing a close fit with the plotted data. This was expected, due to the influence of the shape factor on the stress intensity factor, and therefore the J-integral. The inclusion of higher-order terms within the equation for the shape factor meant that the J-integral was expected to rise more quickly for this edge crack study, versus a configuration using a shape factor that did not change along with the crack length.

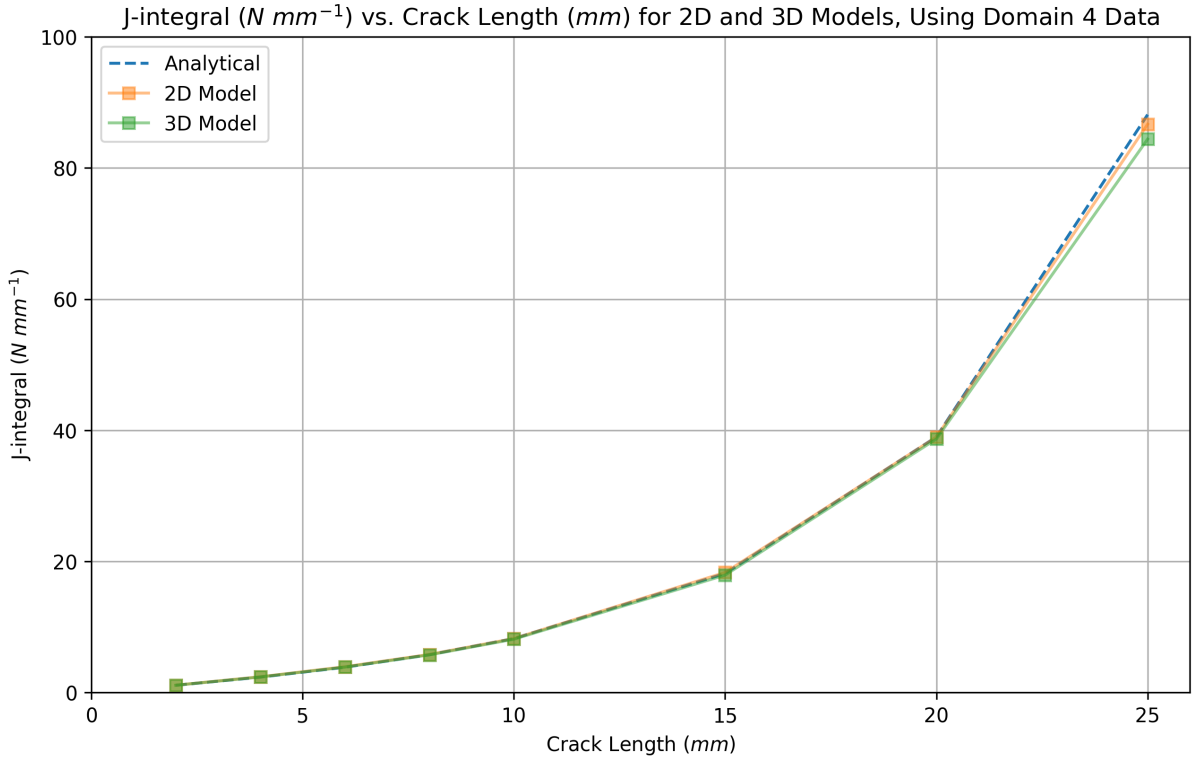

Figure 4.6: J-integral (Nmm^{-1}) vs crack length (mm) for 2D and 3D models using data from domain 4 and a polynomial weight function, compared to analytically calculated values.

The J-integral calculated using the software agreed extremely well with the data obtained via the analytical method, with errors on the order of 1-2% – this also compared well with error rates observed within the literature [49]. Errors on this order are generally expected when using finite element analysis, due to the approximate nature of the method and the potential for small numerical errors. The 2D and 3D models used for the case study also produced results which agreed extremely closely – again with differences of around 1-2%. This clearly demonstrated that the equivalent domain integral method was implemented correctly within the software, and also showed that the method was equally valid for CPS6 and C3D10 elements in the selected case study.

A similar graph was plotted to understand the relationship between the stress intensity factor and the crack length – again for the 2D and 3D models along with the analytical results. This is presented in Figure 4.7. The relationship between the stress intensity factor and the crack length also exhibited a non-linear relationship, with the stress intensity factor rising with the crack length – although at a slower rate than the J-integral. This was expected as the J-integral is calculated using the square root of the J-integral, and therefore it would be expected to show a lower-order relationship with the crack length when compared to the J-integral. Again, the results obtained using the 2D and 3D models compared very favourably to the analytically-obtained results, verifying that the software could be used to calculate sufficiently accurate stress intensity factors, and therefore validating that the software met its intended purpose outlined in §1.2.

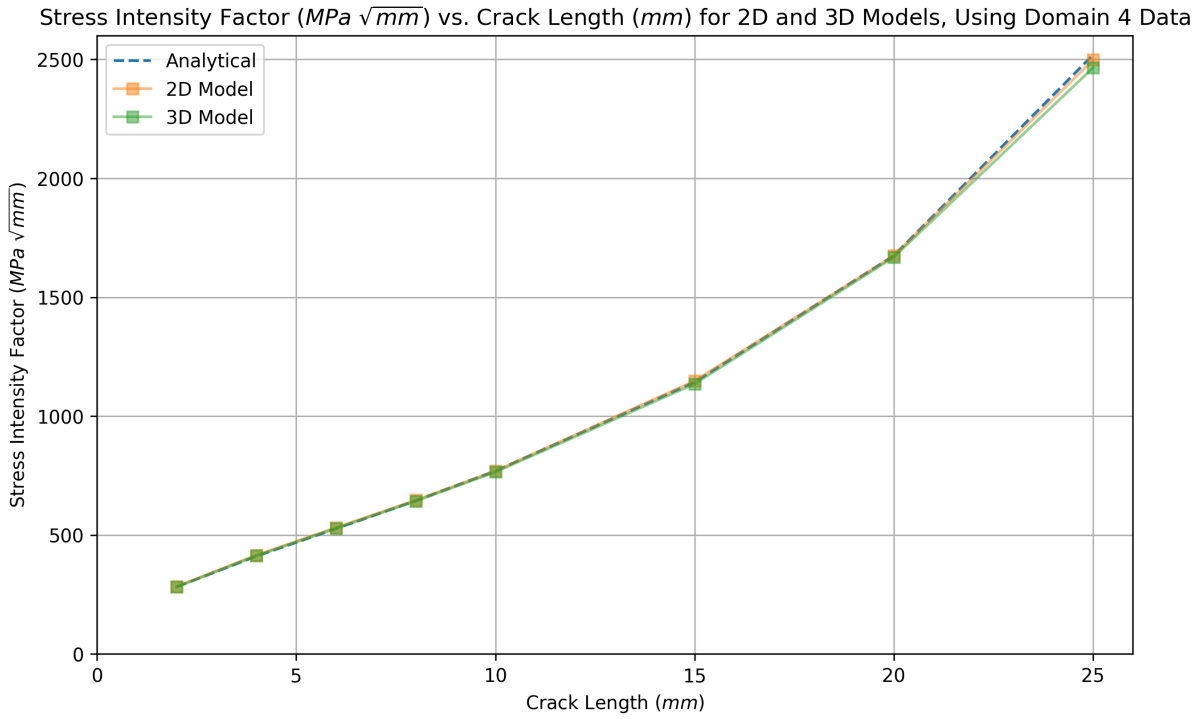

Figure 4.7: Stress intensity factor ($MPa\sqrt{mm}$) vs crack length (mm) for 2D and 3D models using data from domain 4 and a polynomial weight function, compared to analytically obtained values.

For completeness, the stress intensity factors were recorded for each crack length, for both the 2D and 3D models. These are presented in Table 4.1.

Table 4.1: Comparison of J-integral values for various crack lengths: 2D vs. 3D FEA vs. analytical solution, with percentage error.

Crack Length	mm	2	6	10	15	20	25
Analytical SIF	$MPa\sqrt{mm}$	282.4	527.0	769.6	1143.2	1675.3	2519.1
2D SIF	$MPa\sqrt{mm}$	283.7	531.8	767.0	1148.3	1676.9	2498.1
2D Error	%	+0.47	+0.90	+0.05	+0.83	+0.45	-0.02
3D SIF	$MPa\sqrt{mm}$	281.7	528.2	765.1	1135.6	1668.6	2465.1
3D Error	%	-0.22	+0.23	-0.58	-0.66	+0.40	-2.14

Chapter 5

Conclusion

This final chapter provides a summary of the project, and assesses the outcomes of the project against its initial aims. Some limitations of the developed solution are discussed, and some points of further work are identified which could bring the software closer to a tool which can be used practically within industry.

5.1 Project Summary

The overall project was a success, and definitively met its key objectives defined in §1.2.

1. A literature review was undertaken, and a trade study was performed in order to select the most suitable method, software, and programming language to use for the implementation of the software.
2. The software was implemented successfully, and was able to obtain a domain-independent value for the J-integral using results from a tetrahedral-meshed finite element model, by applying the previously selected methods and tools.
3. The case study of an edge crack in a finite plate demonstrated that the J-integral and stress intensity factor produced by the software exhibited a very close agreement (on the order of 1-2%) with results obtained via well-proven analytical methods.

Therefore, this project can be confidently said to have met its overall aim, which was to develop a tool which could determine the stress intensity factor of a crack within a three-dimensional finite element model meshed using a tetrahedral mesh.

However, some simplifications were made during the development of the tool, in order to control the scope and time-scale of the project. Firstly, the three-dimensional case study was simplified by retaining the unit thickness of 1 mm used for the two-dimensional case study. This meant that a state of plane stress was maintained, and therefore that the contributions of through-thickness stresses and crack modes other than mode I were relatively insignificant. In reality, these factors could potentially alter the results significantly for a thicker structure, and therefore the applicability of the tool to such structures could be limited without further development and validation work.

A further simplification was made regarding the crack tip geometry, along with the definition of the integration domains and the weight functions. Regarding the crack-tip geometry for the three-dimensional case study, the two-dimensional crack was extended in a third dimension, while maintaining a straight crack front. This was a good approximation, but disregards the potential for a crack tip to become curved within a thicker structure. Capturing this behaviour would necessitate modelling a curved crack front within the finite element model, along with a new coordinate system defined at separate points along the crack tip. Analysis of such a model not be possible to analyse using this software without further development. This also relates to the definition of the integration domains and the weight functions – a curved crack would also require a curved annular domain, and a weight function which decayed in the through-thickness direction, as well as the crack growth direction.

The final simplification made was the use of a case study which only considered pure mode I crack opening behaviour. In reality, thicker structures are more likely to exhibit mixed-mode behaviour, which increases the complexity of calculating the stress intensity factors, since the J-integral must be split into the contribution from each crack loading mode.

However, the software was still able to capture many important attributes of three-dimensional analysis. A 3D solid model was analysed successfully, and results were obtained which agreed extremely well with the analytically-obtained results. The shape functions of the C3D10 tetrahedral elements were able to be successfully incorporated within the tool, and the three-dimensional annular domains were able to be defined successfully. The volumetric domain integral formulation was implemented correctly, and the domain-independence of the results was demonstrated. The software developed for this project can therefore act as a base, which can be extended in the future in order to incorporate more complex methods that more accurately model the real-world behaviour of cracks within three-dimensional structures.

5.2 Further Work

The key items of further work necessary to improve the functionality of the software are as follows:

1. Analysis of curved crack fronts, where a new coordinate system is defined at points along the crack tip, and weight functions can be specified which decay in the z-direction as well as the x and y directions.
2. Analysis of mixed-mode loading configurations, whereby a total stress intensity factor can be calculated and decomposed into a separate stress intensity factor for each loading mode.
3. Performance of adaptive re-meshing as a crack propagates, and calculation of crack propagation lives using the Paris-Erdogan equation.
4. Validation via a further case study using a three-dimensional component under plane-strain assumptions, with the results being compared to results obtained via physical testing.
5. Implementation of modified tetrahedral elements at the crack tip – for example, quarter point elements.

References

[1] J. Gallagher, C. Babish, and J. Malas, "Damage Tolerant Risk Analysis Techniques for Evaluating the Structural Integrity of Aircraft Structures," *11th International Conference on Fracture 2005, ICF11*, vol. 1, Jan. 2005.

[2] S. Wasserman, "Altair Release Focuses on Aerospace Stress Analysis and Modeling," Sept. 2016.

[3] AFGROW, "DTDHandbook | Introduction | Summary of Damage Tolerance Design Guidelines | Fail Safe Structure | Initial Flaw Considerations."

[4] E. S. Dzidowski, "The Effect of Secondary Metalworking Processes on Susceptibility of Aircraft to Catastrophic Failures and Prevention Methods," *Archives of Metallurgy and Materials*, vol. 58, Dec. 2013.

[5] J. Tomblin, T. Lacy, B. L. Smith, S. Hooper, and A. Vizzini, "Review of Damage Tolerance for Composite Sandwich Airframe Structures," Aug. 1999.

[6] T. Hellen, *How To Undertake Fracture Mechanics Analysis with Finite Elements*. NAFEMS, Aug. 2001.

[7] NASGRO, "Fatigue Crack Growth Analysis," in *NASGRO Manual*, Aug. 2022.

[8] T. L. Anderson, *Fracture Mechanics: Fundamentals and Applications, Fourth Edition*. Boca Raton: CRC Press, 4 ed., Mar. 2017.

[9] N. Kalutskiy, "Meshfree Methods of Airframe Stress Analysis," *IOP Conference Series: Materials Science and Engineering*, vol. 1024, p. 012007, Jan. 2021.

[10] M. Kuna, *Finite Elements in Fracture Mechanics: Theory - Numerics - Applications*, vol. 201 of *Solid Mechanics and Its Applications*. Dordrecht: Springer Netherlands, 2013.

[11] M. Kuna, "FE-Techniques for Crack Analysis in Linear-Elastic Structures," in *Finite Elements in Fracture Mechanics: Theory - Numerics - Applications*, vol. 201 of *Solid Mechanics and Its Applications*, Dordrecht: Springer Netherlands, 2013.

[12] A. R. Ingraffea and P. A. Wawrzynek, "Numerical and Computational Methods," in *Comprehensive Structural Integrity*, vol. 3, Elsevier Science, 1 ed., 2003. ISBN: 9780080437491.

[13] D. Systemes, "2.16.1 J-integral evaluation," in *Abaqus Theory Manual v6.5*.

[14] A. Dharmasaroja, C. Armstrong, A. Murphy, T. Robinson, S. McGuinness, L. Iorga, and J. Barron, "Load Case Characterization for the Aircraft Structural Design Process," *AIAA Journal*, vol. 55, pp. 1–10, May 2017.

[15] M. Nejati, A. Paluszny, and R. W. Zimmerman, "A disk-shaped domain integral method for the computation of stress intensity factors using tetrahedral meshes," *International Journal of Solids and Structures*, vol. 69-70, pp. 230–251, Sept. 2015.

[16] T. Koshima and H. Okada, "Three-dimensional J-integral evaluation for finite strain elastic–plastic solid using the quadratic tetrahedral finite element and automatic meshing methodology," *Engineering Fracture Mechanics*, vol. 135, pp. 34–63, Feb. 2015.

[17] O. Tabaza, H. Okada, and Y. Yusa, "A new approach of the interaction integral method with correction terms for cracks in three-dimensional functionally graded materials using the tetrahedral finite element," *Theoretical and Applied Fracture Mechanics*, vol. 115, p. 103065, Oct. 2021.

[18] E. , "Easy Access Rules for Large Aeroplanes (CS-25) - Revision from January 2023 | EASA," Jan. 2023.

[19] R. Lazzeri, "A COMPARISON BETWEEN SAFE LIFE, DAMAGE TOLERANCE AND PROBABILISTIC APPROACHES TO AIRCRAFT STRUCTURE FATIGUE DESIGN," *L'Aerotecnica, Missili e Spazio*, vol. 2, Jan. 2002.

[20] "Load Spectra," in *Fatigue of Structures and Materials* (J. Schijve, ed.), pp. 259–293, Dordrecht: Springer Netherlands, 2009.

[21] W. D. Pilkey and D. F. Pilkey, *Peterson's Stress Concentration Factors*. John Wiley & Sons, Jan. 2008. Google-Books-ID: JUAdYkKh0V8C.

[22] D. Davidson, K. Chan, R. McClung, and S. Hudak, "4.05 - Small Fatigue Cracks," in *Comprehensive Structural Integrity* (I. Milne, R. O. Ritchie, and B. Karihaloo, eds.), pp. 129–164, Oxford: Pergamon, Jan. 2003.

[23] AFGROW, "DTDHandbook | Examples of Damage Tolerant Analyses | Damage Tolerance Analysis Procedure."

[24] A. A. Griffith, "VI. The phenomena of rupture and flow in solids," *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, vol. 221, pp. 163–198, Jan. 1921.

[25] G. R. Irwin, "ONSET OF FAST CRACK PROPAGATION IN HIGH STRENGTH STEEL AND ALUMINUM ALLOYS," Tech. Rep. NRL-4763; PB-121224, Naval Research Lab., Washington, D.C., May 1956.

[26] G. R. Irwin, "Analysis of Stresses and Strains Near the End of a Crack Traversing a Plate," *Journal of Applied Mechanics*, vol. 24, pp. 361–364, Sept. 1957.

[27] G. C. Sih, *Handbook of Stress-intensity Factors: Stress-intensity Factor Solutions and Formulas for Reference*. Lehigh University, Institute of Fracture and Solid Mechanics, 1973. Google-Books-ID: Wob7xAEACAAJ.

[28] J. R. Rice, "A Path Independent Integral and the Approximate Analysis of Strain Concentration by Notches and Cracks," *Journal of Applied Mechanics*, vol. 35, pp. 379–386, June 1968.

[29] T.-P. Fries and T. Belytschko, "The extended/generalized finite element method: An overview of the method and its applications," *International Journal for Numerical Methods in Engineering*, vol. 84, no. 3, pp. 253–304, 2010. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.2914.

[30] Y. Feng, D. Wu, M. G. Stewart, and W. Gao, "Past, current and future trends and challenges in non-deterministic fracture mechanics: A review," *Computer Methods in Applied Mechanics and Engineering*, vol. 412, p. 116102, July 2023.

[31] D. Mu, H. Qu, Y. Zeng, and A. Tang, "An improved SPH method for simulating crack propagation and coalescence in rocks with pre-existing cracks," *Engineering Fracture Mechanics*, vol. 282, p. 109148, Apr. 2023.

[32] T. Belytschko, Y. Y. Lu, and L. Gu, "Crack propagation by element-free Galerkin methods," *Engineering Fracture Mechanics*, vol. 51, pp. 295–315, May 1995.

[33] M. R. I. Islam and C. Peng, "A Total Lagrangian SPH method for modelling damage and failure in solids," *International Journal of Mechanical Sciences*, vol. 157-158, pp. 498–511, July 2019.

[34] M. Pant, I. V. Singh, and B. K. Mishra, "A novel enrichment criterion for modeling kinked cracks using element free Galerkin method," *International Journal of Mechanical Sciences*, vol. 68, pp. 140–149, Mar. 2013.

[35] G. R. Liu and S. S. Quek, "Chapter 10 - Special Purpose Elements," in *The Finite Element Method (Second Edition)* (G. R. Liu and S. S. Quek, eds.), pp. 289–300, Oxford: Butterworth-Heinemann, Jan. 2014.

[36] V. Murti and S. Valliappan, "A universal optimum quarter point element," *Engineering Fracture Mechanics*, vol. 25, pp. 237–258, Jan. 1986.

[37] I. Fried and S. K. Yang, "Best Finite Elements Distribution around a Singularity," *AIAA Journal*, vol. 10, pp. 1244–1246, Sept. 1972. Publisher: American Institute of Aeronautics and Astronautics.

[38] M. Nejati, A. Paluszny, and R. W. Zimmerman, "On the use of quarter-point tetrahedral finite elements in linear elastic fracture mechanics," *Engineering Fracture Mechanics*, vol. 144, pp. 194–221, Aug. 2015.

[39] S. K. Chan, I. S. Tuba, and W. K. Wilson, "On the finite element method in linear fracture mechanics," *Engineering Fracture Mechanics*, vol. 2, pp. 1–17, July 1970.

[40] D. Parks, "The virtual crack extension method for nonlinear material behavior," *Computer Methods in Applied Mechanics and Engineering*, vol. 12, pp. 353–364, Dec. 1977.

[41] T. K. Hellen, "On the method of virtual crack extensions," *International Journal for Numerical Methods in Engineering*, vol. 9, pp. 187–207, Jan. 1975.

[42] E. Rybicki and M. Kanninen, "A finite element calculation of stress intensity factors by a modified crack closure integral," *Engineering Fracture Mechanics*, vol. 9, pp. 931–938, Jan. 1977.

[43] C. F. Shih, B. Moran, and T. Nakamura, "Energy release rate along a three-dimensional crack front in a thermally stressed body," *International Journal of Fracture*, vol. 30, pp. 79–102, Feb. 1986.

[44] NumPy, "Performance — NumPy v2.2 Manual."

[45] F. Zehra, M. Javed, D. Khan, and M. Pasha, "Comparative Analysis of C++ and Python in Terms of Memory and Tim," 2020.

[46] D. Systemes, "1.2.2 Conventions," in *Abaqus Theory Manual v6.5*.

[47] D. Systemes, "3.2.6 Triangular, tetrahedral, and wedge elements," in *Abaqus Theory Manual v6.5*.

[48] C. Aster, "11.1. [R3.01.01] : Shape functions and integration points — Code-Aster v14 0.1 documentation."

[49] H. Yuan and H. Zhang, "Research on the stress intensity factor of crack on the finite width plate with edge damage based on the finite element method," *Journal of Physics: Conference Series*, vol. 2493, p. 012013, May 2023.

Chapter 6

Appendix

This section includes the source code for the software developed for the project. This is also available at https://github.com/allabright/abaqus-j-integral-sif.

6.1 User Guide

This section provides the readme for the project, which contains a brief user guide on how to use the software to create and run a model, export the results, and calculate the J-integral and stress intensity factor.

````
# Abaqus Stress Intensity Factor Calculator

This repository contains a tool that can be used to calculate the stress intensity factor (SIF) of a crack in a finite element model. The part must be first created in Abaqus, and meshed using a triangular or tetrahedral mesh. The model can be either 2D or 3D.

# Usage

## Arguments

### Mandatory

`--dimension` - The dimension of the model being analysed, either `2d` or `3d`.

`--model` - The name of the model being analysed. The only model supported at present is `Edge_Crack`. However, additional models could be developed. If you have your own ready-made model that you want to analyse, it needs to be dragged into the `data/models` directory, and the name of the directory needs to be passed as the value for this argument.

### Optional

`--input` -- By default, the number of the last directory is used. For example, if the last directory in the `data/models/Edge_Crack_2D` directory is `Model_008`, then that will be used for the export by default. If you want to use another model - for example `Model_005`, then `005` needs to be passed as an argument here.

`--config` -- By default the `config.ini` file in the `config` directory is used to specify the configuration of the analysis. If you want to use a different file, then drag it into the `config` directory and pass the filename as the value for this argument.

## Functions

### Model Creation

The model must first be created in Abaqus, using the 'model.py' file in the 'src' directory. This can be done manually as long as the correct approach is followed, but it is automated here for the edge-crack case study. Further validation models could be added in the future. Usage is as follows:

```
python model.py --args
```

This creates a model using the parameters specified in the configuration file, and runs the analysis. A new directory is created within the `data/models/Edge_Crack_{dimension}` directory for this specific model. The configuration file is copied to this directory, and the results are saved here.

### Results Export

Before the analysis can be performed, it is necessary to dump the raw data (e.g. stresses, strains) from Abaqus. This is done using the `export.py` file contained in the `src` directory. This is a command line script, and is run using:

```
python export.py --args
```

This script creates a new directory for the export under `data/exports/Edge_Crack_{dimension}`, and copies over the model data, and the configuration data used for that model. The data is then dumped using the Abaqus API into a JSON file called `export.
````

```
    json`. This script must be run on a machine which has Abaqus CAE installed.

### Results Analysis

After exporting the raw data from Abaqus, the analysis can be performed. This is done using the `analysis.py` file contained in the `src` directory. This is also a command line script, and is run using the same arguments as the previous script:

```
python analysis.py --args
```

This script creates a new for the analysis under `data/analysis/Edge_Crack_{dimension}`, and copies over the model data, export data, and configuration data. The analysis is then performed, and the results are stored in a JSON file and a CSV file within the run directory.

Example Use

Creating a New Model

No model has been created yet, so you call:

```python model.py --dimension 3d --model edge_crack```

This creates the directory `/data/models/Edge_Crack_3D`, copies over the `config/config.ini` configuration file, then creates and runs a 3D analysis within Abaqus.

Performing an Export

Now you have run four models, but you want to perform an export using the third. You call:

```python export.py --dimension 3d --model edge_crack --input 003```

A new directory is created within `data/exports/Edge_Crack_3D`. The latest model located within `data/models/Edge_Crack_3D/Model_004` is ignored because the `--input` argument was passed. Instead, the data is copied from ``data/models/Edge_Crack_3D/Model_004` to the run directory. The results of the analysis are exported, and saved in the run directory.

Performing an Analysis

Now you have a set of exports within the `data/exports` directory. However, you want to change the weight function used for the analysis, without having to re-run your model and export. You therefore alter the `config/config.ini` file to change the weight function, and save it as `config/config_edited.ini`. You then call:

```python analyse.py --dimension 3d --model edge_crack --config config_edited.ini```

Since no `--input` argument was specified, the most recent export in the `data/exports/Edge_Crack_3D` directory is used. A run directory is created for the analysis in `data/analysis/Edge_Crack_3D`, and the model, export, and configuration data is copied over. Since a `--config` argument was specified, the customised configuration file is used for the analysis. The analysis is performed, and the results are saved in the run directory.
```
```

6.2 Configuration

6.2.1 `config.ini`

This file provides the configuration options for the software, which can be adjusted in order to alter the parameters used for the analysis.

```
1
2  [Analysis]
3  # Leave these as the default values.
4  configuration = deformed
5  stress_state = plane_stress
6
7  # "linear", "gaussian", or "polynomial"
8  weight_function_type = polynomial
9
10 # Number of annular domains to use. Minimum and maximum radii as fractions of the crack
       length.
11 number_of_domains = 10
12 domain_r_min_factor = 0.0
13 domain_r_max_factor = 1.0
14
15 [Modelling]
16 # "regular" uses the same partition style as the 3D model. "circular" uses a circle
       around the crack tip (only available for 2D).
17 partitioning_method = regular
18
19 # Name of the analysis step. Should be no need to change this.
20 step_name = Load_Step
21
22 # Whether to run the job or just create the model and wait.
23 run_job = True
24
25 # Only available for 2D
26 use_quarter_point_elements = False
27
28 [Geometry]
29 # Height of the part in the y-direction (perpendicular to the direction of crack growth),
        in mm.
30 part_height_mm = 100.0
31
32 # Width of the part in the x-direction (in the direction of crack growth), in mm.
33 part_width_mm = 50.0
34
35 part_thickness_mm = 1.0
36
37 # Length of the crack from the left edge of the part to the crack tip, in mm.
38 crack_length_mm = 25.0
39
40 # Given as xyz coordinates. For 2D analysis, just x and y are used.
41 crack_tip_coordinates = (0.0, 0.0, 0.0)
42 tolerance_mm = 0.1
43
44 [Loading]
45 # Tensile stress applied to the top of the plate. Sign doesn't matter as absolute value
       is used.
46 applied_stress_mpa = 100.0
47
48 [Material]
49 material_name = Aluminium
50 material_E = 72000.0
51 material_nu = 0.3
52
53 [Partitioning]
54 # Number of horizontal partitions of the part from top to bottom (per half).
55 horizontal_partition_count = 4
56
57 # How the horizontal partitions should be biased. A higher number means smaller
```

```
    partitions closer to the crack.
horizontal_partition_bias = 1.5

# Parameters for 2D circular meshing

# Number of concentric circles to partition around the crack.
circular_partition_count = 6

# Bias ratio for the circles. A higher bias ratio places more circles closer to the crack tip.
circular_partition_bias = 1.5

# Size of the minimum and maximum circular partitions, as fractions of the crack length.
circle_inner_radius_factor = 0.05
circle_outer_radius_factor = 0.95

# Number of radial spokes to partition around the crack.
spoke_partition_count = 8

[Meshing]

# The number of elements along the crack tip, and along the adjacent sides.
crack_element_count = 15

# The bias ratio of elements along the crack tip. A higher bias ratio means smaller elements near the crack tip.
crack_element_bias = 2

# The seed size for the rest of the part, in mm.
coarse_seed_size_mm = 2.5

# Parameters for 2D circular meshing

# Number of elements for each arc segment of the circle (between radial spokes).
circle_arc_element_count = 12

# Number of elements for each radial spoke segment (between each concentric circle).
circle_spoke_element_count = 12

# Parameters for 3D meshing
through_thickness_element_count = 5
```

6.3 Model Creation

6.3.1 `create_model.py`

This class provides the model-creation methods that are not specific to the edge-crack model, making it easier to develop further validation models.

```
from abaqus import mdb
from abaqusConstants import *
import ast
import ConfigParser
import mesh
import os
import sys

class CreateValidationModel(object):

    def __init__(self):
        pass

    def parse_arguments(self):

        self.dimensions = sys.argv[-4]
        self.model_name = sys.argv[-3]
        self.run_dir = sys.argv[-2]
        self.model_path = sys.argv[-1]

        os.chdir(self.run_dir)

        return self

    def read_configuration_data(self):

        config_path = os.path.join(self.run_dir, "config.ini")
        config = ConfigParser.ConfigParser()
        config.read(config_path)

        # Read geometry data.
        self.part_height_mm = float(config.get("Geometry", "part_height_mm"))
        self.part_width_mm = float(config.get("Geometry", "part_width_mm"))
        self.part_thickness_mm = float(config.get("Geometry", "part_thickness_mm"))
        self.crack_length_mm = float(config.get("Geometry", "crack_length_mm"))
        self.crack_tip_coordinates = ast.literal_eval(config.get("Geometry", "
    crack_tip_coordinates"))

        if self.dimensions == "2d":
            self.crack_tip_coordinates = self.crack_tip_coordinates[:2]

        self.tolerance_mm = float(config.get("Geometry", "tolerance_mm"))

        # Read loading data.
        self.applied_stress_mpa = -abs(float(config.get("Loading", "applied_stress_mpa"))
    )

        # Read material data.
        self.material_name = config.get("Material", "material_name")
        self.material_E = float(config.get("Material", "material_E"))
        self.material_nu = float(config.get("Material", "material_nu"))

        # Read modelling data.
        self.partitioning_method = config.get("Modelling", "partitioning_method")
        self.step_name = config.get("Modelling", "step_name")
        self.run_job = config.getboolean("Modelling", "run_job")
        self.use_quarter_point_elements = config.getboolean("Modelling", "
    use_quarter_point_elements")

        # Read partitioning data.
        self.horizontal_partition_count = int(config.get("Partitioning", "
```

```
        horizontal_partition_count"))
59          self.horizontal_partition_bias = float(config.get("Partitioning", "
        horizontal_partition_bias"))
60
61          if self.dimensions == "2d" and self.partitioning_method == "circular":
62              self.circular_partition_count = int(config.get("Partitioning", "
        circular_partition_count"))
63              self.spoke_partition_count = int(config.get("Partitioning", "
        spoke_partition_count"))
64              self.circular_partition_bias = float(config.get("Partitioning", "
        circular_partition_bias"))
65              self.circle_inner_radius_factor = float(config.get("Partitioning", "
        circle_inner_radius_factor"))
66              self.circle_outer_radius_factor = float(config.get("Partitioning", "
        circle_outer_radius_factor"))
67
68          # Read meshing data.
69          self.crack_element_count = int(config.get("Meshing", "crack_element_count"))
70          self.crack_element_bias = float(config.get("Meshing", "crack_element_bias"))
71          self.coarse_seed_size_mm = float(config.get("Meshing", "coarse_seed_size_mm"))
72
73          if self.dimensions == "2d" and self.partitioning_method == "circular":
74              self.circle_arc_element_count = int(config.get("Meshing", "
        circle_arc_element_count"))
75              self.circle_spoke_element_count = int(config.get("Meshing", "
        circle_spoke_element_count"))
76
77          elif self.dimensions == "3d":
78              self.through_thickness_element_count = int(config.get("Meshing", "
        through_thickness_element_count"))
79
80          return self
81
82      def print_abaqus(self, title=None, spacers=False, spacer_line = "-" * 120):
83          if spacers == True:
84              print >> sys.__stdout__, spacer_line
85          if title is not None:
86              print >> sys.__stdout__, title
87              if spacers == True:
88                  print >> sys.__stdout__, spacer_line
89          return self
90
91      def set_z_and_transform(self):
92          """
93          Set the general z-dimension to be used for the model. For the 2D model this is 0,
         for the 3D model it is half of the part
94          thickness, as the model is created in two halves. The transform variable is used
        to ensure that the crack tip stays on the
95          origin, even in the z-direction.
96          """
97
98          if self.dimensions == "2d":
99              self.z = 0.0
100             self.transform = (1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0)
101         elif self.dimensions == "3d":
102             self.z = self.part_thickness_mm / 2.0
103             self.transform = (1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, self
        .part_thickness_mm/2)
104
105         return self
106
107     def create_model(self):
108         """
109         Create the model in Abaqus using the name defined in the configuration parameters
        .
110         """
111         self.model = mdb.Model(name="Model_" + self.analysis_name)
112         if 'Model-1' in mdb.models.keys():
113             del mdb.models['Model-1']
114
115         return self
116
117     def create_section(self):
```

```
118         """
119         Create the section to be used for the model. This is a solid section for both the
        2D and 3D models.
120         """
121         if self.dimensions == "2d":
122             self.model.HomogeneousSolidSection(
123                 name='PlateSection',
124                 material=self.material_name,
125                 thickness=self.part_thickness_mm
126             )
127         elif self.dimensions == "3d":
128             self.model.HomogeneousSolidSection(name='SolidSection', material=self.
    material_name)
129 
130         return self
131 
132     def assign_section(self):
133         """
134         Assign the defined section to the model. The 2D model assigns it to faces, and
    the 3D model assigns it to cells.
135         """
136 
137         if self.dimensions == "2d":
138 
139             self.part.Set(name='AllFaces', faces=self.part.faces[:])
140             self.part.SectionAssignment(
141                 region=self.part.sets['AllFaces'],
142                 sectionName='PlateSection',
143                 offset=0.0,
144                 offsetType=MIDDLE_SURFACE,
145                 offsetField='',
146                 thicknessAssignment=FROM_SECTION
147             )
148 
149         elif self.dimensions == "3d":
150 
151             self.part.Set(cells=self.part.cells[:], name='WholeBlock')
152             self.part.SectionAssignment(
153                 region=self.part.sets['WholeBlock'],
154                 sectionName='SolidSection')
155 
156         return self
157 
158     def create_material(self):
159         """
160         Define the material using the parameters defined in the configuration file.
161         """
162 
163         self.model.Material(name=self.material_name)
164         self.model.materials[self.material_name].Elastic(table=((self.material_E, self.
    material_nu),))
165 
166         return self
167 
168     def define_mesh_options(self):
169         """
170         Define the mesh options for the 2D and 3D elements.
171         """
172 
173         if self.dimensions == "2d":
174             elem_type_tri = mesh.ElemType(elemCode=CPS6, elemLibrary=STANDARD)
175             self.part.setMeshControls(regions=self.part.faces[:], elemShape=TRI,
    technique=FREE)
176             self.part.setElementType(regions=(self.part.faces[:],), elemTypes=(
    elem_type_tri,))
177 
178         elif self.dimensions == "3d":
179             elem_type_tet = mesh.ElemType(elemCode=C3D10, elemLibrary=STANDARD)
180             self.part.setMeshControls(regions=self.part.cells[:], elemShape=TET,
    technique=FREE)
181             self.part.setElementType(regions=(self.part.cells[:],), elemTypes=(
    elem_type_tet,))
182 
```

```
183         return self
184
185     def mesh_part(self):
186
187         if self.dimensions == "2d":
188             self.part.generateMesh()
189         elif self.dimensions == "3d":
190             self.assembly.generateMesh(regions=((self.instance),))
191
192         return self
193
194     def create_load_step(self):
195
196         self.model.StaticStep(name=self.step_name, previous="Initial")
197
198         return self
199
200     def create_assembly(self):
201
202         self.assembly = self.model.rootAssembly
203         self.assembly.DatumCsysByDefault(CARTESIAN)
204
205         return self
206
207     def create_instance(self):
208
209         if self.dimensions == "2d":
210             self.instance = self.assembly.Instance(name="Instance_" + self.analysis_name,
     part=self.part, dependent=ON)
211         elif self.dimensions == "3d":
212             self.instance = self.assembly.Instance(name="Instance_" + self.analysis_name,
     part=self.part, dependent=OFF)
213
214         return self
215
216     def set_field_outputs(self):
217
218         if 'F-Output-1' in self.model.fieldOutputRequests.keys():
219             foRequest = self.model.fieldOutputRequests['F-Output-1']
220             foRequest.setValues(variables=('S', 'EE', 'U', 'RF', 'SENER'))
221         else:
222             foRequest = self.model.FieldOutputRequest(name='F-Output-1',
223                                                       createStepName=self.step_name,
224                                                       variables=('S', 'EE', 'U', 'RF'))
225
226         return self
227
228     def create_job(self):
229
230         self.job = mdb.Job(
231             name = self.analysis_name,
232             model = "Model_" + self.analysis_name,
233             description = 'Static Strength Analysis Job.'
234             )
235
236         return self
237
238     def submit_job(self):
239
240         if self.run_job == True:
241             self.job.submit()
242             self.job.waitForCompletion()
243
244         return self
245
246     def save_model(self):
247
248         mdb.saveAs(pathName=self.model_path)
249
250         return self
```

6.3.2 edge_crack_general.py

This class provides the model-creation methods that are specific to the edge-crack model, but not specific to whether the model is 2D or 3D.

```
from abaqusConstants import *
import regionToolset

from src.model.create_model import CreateValidationModel

class CreateValidationModelEdgeCrack(CreateValidationModel):

    def __init__(self):
        super(CreateValidationModelEdgeCrack, self).__init__()

    def define_part_geometry(self):
        """
        Define a dictionary to hold the part geometry, calculating the positions of each corner while keeping
        the crack tip at the origin.
        """

        self.part_geometry = {
            "top_left_corner_coords": (-self.crack_length_mm, self.part_height_mm / 2),
            "top_right_corner_coords": (self.part_width_mm - self.crack_length_mm, self.part_height_mm / 2),
            "bottom_left_corner_coords": (-self.crack_length_mm, -self.part_height_mm / 2),
            "bottom_right_corner_coords": (self.part_width_mm - self.crack_length_mm, -self.part_height_mm / 2),
            "crack_start_coords": (-self.crack_length_mm, self.crack_tip_coordinates[1]),
            "crack_end_coords": self.crack_tip_coordinates}

        return self

    def create_part(self):
        """
        Create the sketch for the overall part, and extrude it into a shell or a solid based on the dimension.
        """

        sketch = self.model.ConstrainedSketch(name='Part_Sketch', sheetSize=500.0)
        sketch.rectangle(point1=self.part_geometry["bottom_left_corner_coords"], point2=self.part_geometry["top_right_corner_coords"])

        if self.dimensions == "2d":
            self.part = self.model.Part(name="Part_" + self.analysis_name, dimensionality=TWO_D_PLANAR, type=DEFORMABLE_BODY)
            self.part.BaseShell(sketch=sketch)

        elif self.dimensions == "3d":
            self.part = self.model.Part(name="Part_" + self.analysis_name, dimensionality=THREE_D, type=DEFORMABLE_BODY)
            self.part.BaseSolidExtrude(sketch=sketch, depth=self.part_thickness_mm/2)

        del self.model.sketches['Part_Sketch']

        return self

    def calculate_horizontal_partition_heights(self):
        """
        The model is partitioned horizontally in a number of sections defined in the configuration. This method calculates
        the heights of each partition. The partitions may be biased so that they are closer together the closer they are to the
        crack tip.
        """

        base_height = self.crack_length_mm / 2.0
        self.partition_heights = []
        total_ratio = sum(self.horizontal_partition_bias**i for i in range(self.
```

```
        horizontal_partition_count))
57          step = (self.part_height_mm / 2.0 - base_height) / total_ratio
58
59          for i in range(self.horizontal_partition_count):
60              self.partition_heights.append(base_height + step * sum(self.
        horizontal_partition_bias**j for j in range(i + 1)))
61
62          return self
63
64      def partition_part_top_to_bottom(self):
65          """
66          Create the main horizontal partitions for the part, using horizontal lines. This
        partitions the face rather than the solid.
67          A separate method is used to partition the 3D solid once the face has been
        partitioned.
68          """
69
70          line_sketch = self.model.ConstrainedSketch(name='LineSketch', sheetSize=500.0,
        transform=self.transform)
71
72          line_sketch.Line(point1=(-self.crack_length_mm, 0.0), point2=(self.
        crack_length_mm, 0.0))
73
74          for partition in self.partition_heights:
75              line_sketch.Line(point1=(-self.crack_length_mm, partition), point2=(self.
        part_width_mm-self.crack_length_mm, partition))
76              line_sketch.Line(point1=(-self.crack_length_mm, -partition), point2=(self.
        part_width_mm-self.crack_length_mm, -partition))
77
78          checkPoint = (0.0, 0.0, self.z)
79          faceList = self.part.faces.findAt((checkPoint,))
80          if not faceList:
81              raise ValueError("No face found at point {}. Check geometry.".format(
        checkPoint))
82          main_face = faceList[0]
83
84          self.part.PartitionFaceBySketch(faces=main_face, sketch=line_sketch)
85          del self.model.sketches['LineSketch']
86
87          return self
88
89      def partition_edges(self):
90          """
91          Create partitions for the crack. One partition runs the length of the crack, and
        the other runs the length of the crack again,
92          starting at the crack tip. Therefore we have two line partitions, both starting
        at the crack tip, and moving in the direction of crack
93          growth and opposite (x and -x) for a length equal to the crack length.
94          """
95
96          # Create two datum planes - one at the crack tip and one ahead of the crack tip
        by one crack length.
97          datum_plane_crack_tip = self.part.DatumPlaneByPrincipalPlane(principalPlane=
        YZPLANE, offset=0.0)
98          datum_plane_forward_of_crack_tip = self.part.DatumPlaneByPrincipalPlane(
        principalPlane=YZPLANE, offset=self.crack_length_mm)
99
100         datum_planes = (datum_plane_crack_tip, datum_plane_forward_of_crack_tip)
101         x = self.crack_length_mm / 2.0
102         y_coords = (-self.partition_heights[0], 0.0, self.partition_heights[0])
103         z_coords = (self.tolerance_mm, -self.tolerance_mm)
104
105         for datum_plane in datum_planes:
106
107             entities_to_partition = []
108
109             if self.dimensions == "2d":
110                 for y in y_coords:
111                     entity_to_partition = self.part.edges.findAt(((x, y, 0.0),))[0]
112                     entities_to_partition.append(entity_to_partition)
113                 try:
114                     self.part.PartitionEdgeByDatumPlane(edges=entities_to_partition,
        datumPlane=self.part.datums[datum_plane.id])
```

```
                except:
                    continue

            elif self.dimensions == "3d":

                for y in y_coords:
                    for z in z_coords:
                        entity_to_partition = self.part.faces.findAt(((x, y, z),))[0]
                        entities_to_partition.append(entity_to_partition)
                try:
                    self.part.PartitionFaceByDatumPlane(faces=entities_to_partition, datumPlane=self.part.datums[datum_plane.id])
                except:
                    continue

        return self

    def create_sets_for_top_and_bottom_edges(self):
        """
        Create sets for the top and bottom edges / faces, and the bottom left corner, to be used to apply the loading and boundary conditions.
        """

        if self.dimensions == "2d":
            top_edge = self.part.edges.findAt(((0.0, self.part_height_mm / 2, 0.0),))
            bottom_left_corner = self.part.vertices.findAt(((-self.crack_length_mm, -self.part_height_mm / 2, 0.0),))
            bottom_edge = self.part.edges.findAt(((0.0, -self.part_height_mm / 2, 0.0),))
            top_surface = self.part.Surface(name='Top_Surface', side1Edges=top_edge)

            self.part.Set(name='Top_Edge', edges=top_edge)
            self.part.Set(name='Bottom_Left_Corner', vertices=bottom_left_corner)
            self.part.Set(name='Bottom_Edge', edges=bottom_edge)

        elif self.dimensions == "3d":
            top_face = self.part.faces.findAt(((0.0, self.part_height_mm / 2.0, 0.0),))
            bottom_left_edge = self.part.edges.findAt(((-self.crack_length_mm, -self.part_height_mm / 2, self.z / 2.0),))
            bottom_face = self.part.faces.findAt(((0.0, -self.part_height_mm / 2.0, self.z / 2.0),))
            top_surface = self.part.Surface(name='Top_Surface', side1Faces=top_face)

            self.part.Set(name='Top_Face', faces=top_face)
            self.part.Set(name='Bottom_Left_Edge', edges=bottom_left_edge)
            self.part.Set(name='Bottom_Face', faces=bottom_face)

        return self

    def create_crack_seam_set_geometry(self):
        """
        Define a crack seam within Abaqus, allowing the crack nodes to separate under load and allow the crack to open.
        """

        x_min, x_max = -self.crack_length_mm - self.tolerance_mm, 0.0 + self.tolerance_mm
        y_min, y_max = -self.tolerance_mm, self.tolerance_mm

        if self.dimensions == "2d":
            z_min, z_max = -self.tolerance_mm, self.tolerance_mm
            crack_edges = self.part.edges.getByBoundingBox(
                xMin=x_min, xMax=x_max,
                yMin=y_min, yMax=y_max,
                zMin=z_min, zMax=z_max)

            self.part.Set(name='CrackSeam', edges=crack_edges)
            crack_region = regionToolset.Region(edges=crack_edges)
            self.part.engineeringFeatures.assignSeam(regions=crack_region)

        elif self.dimensions == "3d":
            z_min, z_max = -self.part_thickness_mm - self.tolerance_mm, self.part_thickness_mm + self.tolerance_mm
            crack_edges = self.instance.faces.getByBoundingBox(
                xMin=x_min, xMax=x_max,
```

```
                yMin=y_min, yMax=y_max,
                zMin=z_min, zMax=z_max)

            self.assembly.Set(name='CrackSeam', faces=crack_edges)
            crack_region = regionToolset.Region(faces=crack_edges)
            self.assembly.engineeringFeatures.assignSeam(regions=crack_region)

        return self

    def create_crack_seam_set_nodes(self):
        """
        Create a set containing all of the nodes that make up the crack seam.
        """

        node_labels = []

        if self.dimensions == "2d":
            geom_set =  self.part.sets['CrackSeam']
            for edge in geom_set.edges:
                nodes = edge.getNodes()
                node_labels.extend([node.label for node in nodes])

        elif self.dimensions == "3d":
            geom_set =  self.assembly.sets['CrackSeam']
            nodes = geom_set.nodes
            self.assembly.Set(name="CrackSeam_Nodes",  nodes=nodes)

        # Create a node set on the part
        if node_labels:
            self.part.SetFromNodeLabels(name='CrackSeam_Nodes', nodeLabels=node_labels)
        else:
            print("No nodes found in 'CrackSeam'!")

        return self

    def apply_top_pressure_load(self):
        """
        Apply the negative pressure load (i.e. tensile stress) to the top face/edge of
    the part.
        """

        top_region = self.instance.surfaces["Top_Surface"]

        try:
            self.model.Pressure(name='Pressure_Load_Top',
                                createStepName=self.step_name,
                                region=top_region,
                                magnitude=self.applied_stress_mpa,
                                amplitude=UNSET)
        except Exception as e:
            self.print_abaqus("Failed to apply pressure load: " + str(e))

        return self

    def apply_bottom_fixed_boundary_condition(self):
        """
        Apply the boundary conditions to fix the lower left corner of the part, and
    prevent displacement of the lower
        surface in the y-direction (i.e. fixed and roller support respectively).
        """

        if self.dimensions == "2d":
            bottom_left = self.instance.sets['Bottom_Left_Corner']
            bottom_region = self.instance.sets['Bottom_Edge']
        elif self.dimensions == "3d":
            bottom_left = self.instance.sets['Bottom_Left_Edge']
            bottom_region = self.instance.sets['Bottom_Face']

        try:
            self.model.DisplacementBC(name='Pinned_Support_Bottom_Left',
                                      createStepName=self.step_name,
                                      region=bottom_left,
                                      u1=0.0, u2=0.0, u3=0.0, ur1=0.0, ur2=0.0, ur3=0.0)
```

```
252
253
254             self.model.DisplacementBC(name='Roller_Support_Bottom',
255                                        createStepName=self.step_name,
256                                        region=bottom_region,
257                                        u2=0.0)
258         except Exception as e:
259             self.print_abaqus("Failed to create boundary condition: " + str(e))
260
261         return self
262
263     def seed_crack_tip_region(self):
264         """
265         Seed the region around the crack tip - effectively four lines starting at the crack tip, in the x, -x, y, and -y directions.
266         This allows the mesh to be refined as it gets closer to the crack tip and maximises performance.
267         """
268
269         self.left_edges = []
270         self.right_edges = []
271         self.top_edges = []
272         self.bottom_edges = []
273
274         for x, left_right in (
275             (-self.crack_length_mm / 2.0, self.left_edges),
276             (self.crack_length_mm / 2.0, self.right_edges)):
277             for y in (-self.partition_heights[0], 0.0, self.partition_heights[0]):
278                 for z in (-self.z, 0.0, self.z):
279                     edge = self.part.edges.findAt(((x, y, z),))
280                     left_right.append(edge[0])
281
282         for y, top_bottom in (
283             (self.partition_heights[0] / 2.0, self.top_edges),
284             (-self.partition_heights[0] / 2.0, self.bottom_edges)):
285             for x in (-self.crack_length_mm, self.crack_length_mm):
286                 for z in (-self.z, 0.0, self.z):
287                     edge = self.part.edges.findAt(((x, y, z),))
288                     if edge:
289                         top_bottom.append(edge[0])
290
291         for edges in (self.right_edges, self.left_edges, self.top_edges, self.bottom_edges):
292             self.part.seedEdgeByBias(SINGLE, end1Edges=edges, ratio=self.crack_element_bias, number=self.crack_element_count)
293
294         return self
295
296     def seed_rest_of_part_regular_partitioning(self):
297         """
298         Seed the rest of the part with a coarse mesh, after excluding the edges near the crack tip that have already had the finer
299         mesh seeds applied.
300         """
301
302         allEdges = self.part.edges[:]
303
304         other_edges = []
305
306         if self.left_edges and self.right_edges and self.top_edges and self.bottom_edges:
307             for edge in allEdges:
308                 if edge not in self.left_edges and \
309                     edge not in self.right_edges and \
310                         edge not in self.top_edges and \
311                             edge not in self.bottom_edges:
312
313                     if self.dimensions == "2d":
314                         other_edges.append(edge)
315
316                     elif self.dimensions == "3d":
317                         if edge not in self.through_thickness_edges:
318                             other_edges.append(edge)
319         else:
```

```
320             other_edges = [ed for ed in allEdges]
321 
322         if other_edges:
323             self.part.seedEdgeBySize(edges=other_edges, size=self.coarse_seed_size_mm,
    deviationFactor=0.1)
324 
325         return self
326 
327     def move_crack_tip_nodes(self):
328         """
329         Move the nodes of the crack tip elements to the quarter-point position. This was
    only implemented for the
330         2D model due to time. Implementation of this for 3D is available in:
331         "On the use of quarter-point tetrahedral finite elements in linear elastic
    fracture mechanics"
332         (M. Nejati, A. Paluszny, R. W. Zimmerman, 2015) and can be implemented here in
    the future.
333 
334         """
335 
336         crack_tip_element_side_lengths = []
337         crack_tip_node = self.part.nodes.getClosest((self.crack_tip_coordinates,))[0]
338         crack_tip_elements = self.part.nodes.getFromLabel(crack_tip_node.label).
    getElements()
339 
340         crack_tip_edge_nodes = []
341         for elem in crack_tip_elements:
342             for edge in elem.getElemEdges():
343                 edge_nodes = edge.getNodes()
344                 if crack_tip_node in edge_nodes:
345                     nodes = [n.label for n in edge_nodes]
346                     crack_tip_edge_nodes.append(sorted(nodes))
347 
348         unique_label_sets = list(set(tuple(lst) for lst in crack_tip_edge_nodes))
349         unique_label_sets = [list(t) for t in unique_label_sets]
350 
351         for node_set in unique_label_sets:
352             nodes = [self.part.nodes.getFromLabel(label) for label in node_set]
353 
354             # Identify the crack tip node in this set.
355             A = None
356             for n in nodes:
357                 if n.label == crack_tip_node.label:
358                     A = n
359                     break
360             if A is None:
361                 # If for some reason the crack tip node isn't in the set, skip it.
362                 continue
363 
364             # The other two nodes in the set.
365             candidates = [n for n in nodes if n.label != crack_tip_node.label]
366 
367             # Determine which candidate is farther from the crack tip (A).
368             distances = [sum((n.coordinates[i] - A.coordinates[i])**2 for i in range(3))
    for n in candidates]
369             crack_tip_element_side_lengths.append(max(distances))
370 
371             far_node = candidates[distances.index(max(distances))]
372             mid_node = candidates[distances.index(min(distances))]
373 
374             # Compute the new mid node coordinates as the quarter point between A and the
     far node.
375             A_coords = A.coordinates
376             B_coords = far_node.coordinates
377             M_new = tuple(A_coords[i] + 0.25 * (B_coords[i] - A_coords[i]) for i in range
    (3))
378 
379             # Update only the mid node's coordinates.
380             self.part.editNode(nodes=(mid_node,),
381                                coordinate1=M_new[0],
382                                coordinate2=M_new[1],
383                                coordinate3=M_new[2])
384 
```

```
385             return self
```

6.3.3 edge_crack_2d.py

This class provides the model-creation methods that are specific to the 2D edge-crack model.

```
import math
from abaqusConstants import *
from src.model.edge_crack_general import CreateValidationModelEdgeCrack


class CreateValidationModelEdgeCrack2D(CreateValidationModelEdgeCrack):

    def __init__(self,
                 analysis_name = "Edge_Crack_2D",
                 dimensions = "2d"
                 ):

        super(CreateValidationModelEdgeCrack2D, self).__init__()
        self.analysis_name = analysis_name
        self.dimensions = dimensions

    def calculate_circle_radii(self):
        """
        Calculate the radii to use when using the circular partitioning scheme.
        """
        self.circle_radii = []
        self.circle_inner_radius = self.crack_length_mm * self.circle_inner_radius_factor
        self.circle_outer_radius = self.crack_length_mm * self.circle_outer_radius_factor

        if self.number_of_circles <= 1:
            self.circle_radii.append(self.circle_inner_radius)

        else:
            for i in range(self.number_of_circles):
                s = float(i) / (self.number_of_circles - 1)
                sbias = s ** self.circle_bias
                # Interpolate between inner and outer
                r = self.circle_inner_radius + (self.circle_outer_radius - self.
    circle_inner_radius) * sbias

                self.circle_radii.append(r)

        return self

    def partition_circle_around_crack(self):
        """
        Partition the set of circles around the crack tip, when using the circular
    partitioning scheme.
        """

        circleSketch = self.model.ConstrainedSketch(name='CrackTipCircleSketch',
    sheetSize=500.0, transform=self.transform)

        for radius in self.circle_radii:

            circleSketch.CircleByCenterPerimeter(
                center=(0.0, 0.0),
                point1=(radius, 0)
            )

        top_face = self.part.faces.getClosest(coordinates=((0.0, self.tolerance_mm, self.
    part_thickness_mm / 2.0), ))
        bottom_face = self.part.faces.getClosest(coordinates=((0.0, -self.tolerance_mm,
    self.part_thickness_mm / 2.0), ))

        partition_faces = (top_face[0][0], bottom_face[0][0])

        self.part.PartitionFaceBySketch(faces=partition_faces, sketch=circleSketch)

        del self.model.sketches['CrackTipCircleSketch']

```

```
62          return self
63
64      def partition_radially_around_crack(self):
65          """
66          Partition the radial spokes within the circles around the crack tip, when using
        the circular partitioning scheme.
67          """
68
69          spokeSketch = self.model.ConstrainedSketch(name='SpokeSketch', sheetSize=500.0,
        transform=self.transform)
70
71          for i in range(self.number_of_spokes):
72              angle = 2.0 * math.pi * (float(i) / self.number_of_spokes)
73              x_end = self.circle_outer_radius * math.cos(angle)
74              y_end = self.circle_outer_radius * math.sin(angle)
75
76              if y_end != 0:
77                  spokeSketch.Line(point1=(0.0, 0.0), point2=(x_end, y_end))
78                  spokeSketch.Line(point1=(0.0, 0.0), point2=(x_end, -y_end))
79
80          top_face = self.part.faces.getClosest(coordinates=((0.0, self.tolerance_mm, self.
        part_thickness_mm / 2.0), ))
81          bottom_face = self.part.faces.getClosest(coordinates=((0.0, -self.tolerance_mm,
        self.part_thickness_mm / 2.0), ))
82
83          partition_faces = (top_face[0][0], bottom_face[0][0])
84
85          self.part.PartitionFaceBySketch(faces=partition_faces, sketch=spokeSketch)
86          del self.model.sketches['SpokeSketch']
87
88          return self
89
90      def seed_arc_segments(self):
91          """
92          Seed the arc segments (the edges along each circle between two adjacent spokes).
93          """
94
95          self.arc_edges = []
96
97          for r in self.circle_radii:
98              # Consider each pair of consecutive spokes
99              for j in range(self.number_of_spokes):
100                 # Define an arc section swept along the circle
101                 angle_start = 2.0 * math.pi * j / self.number_of_spokes
102                 angle_end   = 2.0 * math.pi * (j + 1) / self.number_of_spokes
103                 mid_angle = 0.5 * (angle_start + angle_end)
104
105                 # Coordinates for midpoint on this arc
106                 x_mid = r * math.cos(mid_angle)
107                 y_mid = r * math.sin(mid_angle)
108
109                 try:
110                     found_edges = self.part.edges.findAt(((x_mid, y_mid, self.z),))
111                     for edge in found_edges:
112                         self.arc_edges.append(edge)
113                     if self.dimensions == "3d":
114                         found_edges_2 = self.part.edges.findAt(((x_mid, y_mid, -self.z),)
        )
115                         for edge in found_edges_2:
116                             self.arc_edges.append(edge)
117                 except Exception as e:
118                     print("Could not find arc edge at r={r:.3f}, angles=({angle_start:.3f
        },{angle_end:.3f}): {e}")
119
120         # Seed each arc edge with a fixed number of elements
121         if self.arc_edges:
122             self.part.seedEdgeByNumber(edges=self.arc_edges, number=self.
        circle_arc_elements)
123
124         return self
125
126     def seed_radial_segments(self):
127         """
```

```
128         Seed the radial segments (spokes) created when using the circular partitioning
    scheme.
129         """
130 
131         self.radial_edges = []
132 
133         for spoke_index in range(self.number_of_spokes):
134             angle = 2.0 * math.pi * spoke_index / self.number_of_spokes
135             cosA = math.cos(angle)
136             sinA = math.sin(angle)
137 
138             for i in range(len(self.circle_radii)):
139 
140                 if i == 0:
141                     r_i = 0.0
142                 else:
143                     r_i = self.circle_radii[i-1]
144                 r_j = self.circle_radii[i]
145 
146                 r_mid = 0.5 * (r_i + r_j)
147                 # Cartesian coordinates of that midpoint
148                 x_mid = r_mid * cosA
149                 y_mid = r_mid * sinA
150 
151                 try:
152                     found_edges = self.part.edges.findAt(((x_mid, y_mid, self.z),))
153                     for edge in found_edges:
154                         self.radial_edges.append(edge)
155 
156                     if self.dimensions == "3d":
157                         found_edges_2 = self.part.edges.findAt(((x_mid, y_mid, -self.z),)
    )
158                         for edge in found_edges_2:
159                             self.radial_edges.append(edge)
160                 except Exception as e:
161                     print("Could not find edge for spoke {spoke_index}, segment {i}: {e}"
    )
162 
163         # Seed each segment edge with a fixed number of elements
164         if self.radial_edges:
165             self.part.seedEdgeByNumber(edges=self.radial_edges, number=self.
    circle_radius_elements)
166 
167         return self
168 
169     def seed_rest_of_part_circular_partitioning(self):
170         """
171         When using the circular partitioning scheme, seed the rest of the part, excluding
     the arc segments and
172         radial segments which have already been seeded.
173         """
174 
175         allEdges = self.part.edges[:]
176 
177         if self.radial_edges and self.arc_edges:
178             otherEdges = [ed for ed in allEdges if ed not in self.radial_edges and ed not
     in self.arc_edges]
179 
180         if otherEdges:
181             self.part.seedEdgeBySize(edges=otherEdges, size=self.coarse_seed_size_mm,
    deviationFactor=0.1)
182 
183         return self
184 
185 if __name__ == '__main__':
186 
187     run = (
188         CreateValidationModelEdgeCrack2D()
189             .parse_arguments()
190             .read_configuration_data()
191             .calculate_horizontal_partition_heights()
192             .set_z_and_transform()
193             .create_model()
```

```
            .create_load_step()
            .set_field_outputs()
            .define_part_geometry()
            .create_part()
            .create_material()
            .partition_part_top_to_bottom()

            .create_section()
            .assign_section()
            .define_mesh_options()
    )


    if run.partitioning_method == "regular":
        (run
            .partition_edges()
            .seed_crack_tip_region()
            .seed_rest_of_part_regular_partitioning()
        )

    elif run.partitioning_method == "circular":
        (run
            .calculate_circle_radii()
            .partition_radially_around_crack()
            .partition_circle_around_crack()
            .seed_radial_segments()
            .seed_arc_segments()
            .seed_rest_of_part_circular_partitioning()
        )

    run.mesh_part()

    if run.use_quarter_point_elements:
        run.move_crack_tip_nodes()

    (run
        .create_crack_seam_set_geometry()
        .create_crack_seam_set_nodes()
        .create_sets_for_top_and_bottom_edges()
        .create_assembly()
        .create_instance()
        .apply_top_pressure_load()
        .apply_bottom_fixed_boundary_condition()
        .create_job()
        .submit_job()
        .save_model()
     )
```

6.3.4 edge_crack_3d.py

This class provides the model-creation methods that are specific to the 3D edge-crack model.

```
from abaqusConstants import *
from src.model.edge_crack_general import CreateValidationModelEdgeCrack


class CreateValidationModelEdgeCrack3D(CreateValidationModelEdgeCrack):

    def __init__(self,
                 analysis_name = "Edge_Crack_3D",
                 dimensions = "3d"):

        super(CreateValidationModelEdgeCrack3D, self).__init__()
        self.analysis_name = analysis_name
        self.dimensions = dimensions

    def mirror_geometry(self):
        """
        Mirror the geometry to create two cells. This is necessary to keep the crack tip at the origin when using a 3D model,
        as Abaqus does not allow for a mid-point extrusion.
        """

        checkPoint = (0.0, 0.0, 0.0)
        faceList = self.part.faces.findAt((checkPoint,))
        if not faceList:
            raise ValueError("No face found at point {}. Check geometry.".format(checkPoint))
        main_face = faceList[0]
        self.part.Mirror(mirrorPlane=main_face, keepOriginal=True)

        return self

    def partition_model(self, mirror=False):
        """
        Finds radial and arc edges, filters out boundary edges, and sweeps them downward through the full thickness
        to create a pie-slice-shaped partitioned structure. This is performed on the full model, after mirroring.
        """

        x_min, x_max = -self.crack_length_mm - self.tolerance_mm, self.part_width_mm - self.crack_length_mm + self.tolerance_mm
        y_min, y_max = -self.part_height_mm / 2.0 - self.tolerance_mm, self.part_height_mm / 2.0 + self.tolerance_mm

        if mirror:
            z_min = -self.z - self.tolerance_mm
            z_max = 0.0
            z_gen = -self.z
        else:
            z_min = 0.0
            z_max = self.z + self.tolerance_mm
            z_gen = self.z

        # Select candidate edges
        candidate_edges = self.part.edges.getByBoundingBox(
            xMin=x_min, xMax=x_max,
            yMin=y_min, yMax=y_max,
            zMin=z_min, zMax=z_max
        )

        cellMidPoint = (0.0, 0.0, z_gen)
        cell = self.part.cells.findAt((cellMidPoint,))

        internal_edges = []

        for edge in candidate_edges:
```

```
61              x, y, z = edge.pointOn[0]
62              if x != -self.crack_length_mm and x != self.part_width_mm - self.
    crack_length_mm and \
63                  y != self.part_height_mm / 2.0 and y != -self.part_height_mm / 2.0 and \
64                  z == z_gen:
65                   internal_edges.append(edge)
66
67          direction_edge = self.part.edges.findAt(
68              ((-self.crack_length_mm, self.part_height_mm / 2.0, z_gen / 2.0),)
69          )
70
71          self.part.PartitionCellBySweepEdge(
72              cells=cell,
73              edges=tuple(internal_edges),
74              sweepPath=direction_edge[0]
75          )
76
77          return self
78
79      def seed_through_thickness_edges(self):
80          """
81          Seed the through-thickness edges near the crack tip, in the z-direction.
82          """
83
84          self.through_thickness_edges = []
85
86          for edge in self.part.edges[:]:
87
88              # Get the vertices of each edge of the part, along with their starts and ends
    .
89              vertex_1 = edge.getVertices()[0]
90              vertex_2 = edge.getVertices()[1]
91
92              vertex_1_coords_x = self.part.vertices[vertex_1].pointOn[0][0]
93              vertex_1_coords_y = self.part.vertices[vertex_1].pointOn[0][1]
94
95              vertex_2_coords_x = self.part.vertices[vertex_2].pointOn[0][0]
96              vertex_2_coords_y = self.part.vertices[vertex_2].pointOn[0][1]
97
98              # If the edge is within the central partition of the model, and is directly
    in the z-direction,
99              # register it as a through-thickness edge to seed.
100             if abs(vertex_1_coords_y) <= abs(self.partition_heights[1]):
101                 if abs(vertex_1_coords_x) <= self.crack_length_mm:
102                     if vertex_1_coords_x == vertex_2_coords_x and vertex_1_coords_y ==
    vertex_2_coords_y:
103                         self.through_thickness_edges.append(edge)
104
105         self.part.seedEdgeByNumber(self.through_thickness_edges, self.
    through_thickness_element_count)
106
107         return self
108
109 if __name__ == '__main__':
110
111     run = CreateValidationModelEdgeCrack3D()
112
113     (run
114         .parse_arguments()
115         .read_configuration_data()
116         .calculate_horizontal_partition_heights()
117         .set_z_and_transform()
118         .create_model()
119         .create_load_step()
120         .set_field_outputs()
121         .define_part_geometry()
122         .create_part()
123         .create_material()
124         .partition_part_top_to_bottom()
125         .mirror_geometry()
126         .partition_model(mirror=False)
127         .partition_model(mirror=True)
128         .partition_edges()
```

```
        .create_section()
        .assign_section()
        .define_mesh_options()
        .seed_crack_tip_region()
        .seed_through_thickness_edges()
        .seed_rest_of_part_regular_partitioning()
        .create_sets_for_top_and_bottom_edges()
        .create_assembly()
        .create_instance()
        .create_crack_seam_set_geometry()
        .mesh_part()
        .create_crack_seam_set_nodes()
        .apply_top_pressure_load()
        .apply_bottom_fixed_boundary_condition()
        .create_job()
        .submit_job()
        .save_model()
    )
```

6.4 Results Export

6.4.1 export_results.py

This class provides the methods used to export the results from the Abaqus ODB to a JSON file, to be used for further analysis.

```
from collections import defaultdict
import ast
import ConfigParser
import json
import os
import sys
from odbAccess import *


class ExportResults:
    """
    The purpose of this class is to export results from an Abaqus run. The end result is a JSON file with a key for each element.
    Each element then contains data on the nodes which connect to it, including the coordinates and displacements of those nodes.
    The stress data and strain energy data for each element is also stored.
    """
    def __init__(self,
                 elements = defaultdict(dict),
                 nodes = defaultdict(dict),
                 crack_tip_nodes = defaultdict(dict),
                 shear_stress_factor = 1.0, # Abaqus provides shear stresses already in tensorial form.
                 shear_strain_factor = 2.0 # Abaqus provides shear strains in engineering form. We need to divide by 2 to convert to tensorial form.
                 ):
        self.print_abaqus("Exporting Data from Abaqus", spacers=True)
        self.elements = elements
        self.nodes = nodes
        self.crack_tip_nodes = crack_tip_nodes
        self.shear_stress_factor = shear_stress_factor
        self.shear_strain_factor = shear_strain_factor

    def parse_arguments(self):

        self.print_abaqus("Parsing command line arguments")

        self.dimensions = sys.argv[-4]
        self.model_name = sys.argv[-3]
        self.run_dir = sys.argv[-2]
        self.odb_path = sys.argv[-1]

        os.chdir(self.run_dir)

        return self

    def read_configuration_data(self):

        config_path = os.path.join(self.run_dir, "config.ini")

        config = ConfigParser.ConfigParser()
        config.read(config_path)

        self.crack_tip_coordinates = ast.literal_eval(config.get("Geometry", "crack_tip_coordinates"))

        if self.dimensions == "2d":
            self.crack_tip_coordinates = self.crack_tip_coordinates[:2]

        self.tolerance_mm = float(config.get("Geometry", "tolerance_mm"))

```

```
57          return self
58
59      def load_model_parameters(self):
60
61          self.print_abaqus("Loading model parameters")
62
63          self.instance_name = "INSTANCE_" + self.model_name.upper()
64
65          self.odb = openOdb(self.odb_path)
66          self.assembly = self.odb.rootAssembly
67          self.instance = self.odb.rootAssembly.instances[self.instance_name]
68          self.frame = self.odb.steps["Load_Step"].frames[-1]
69
70          return self
71
72      def get_crack_tip_nodes(self):
73          """
74          The modelling script creates a set for the crack tip nodes. This is retrieved
        here.
75          """
76
77          self.print_abaqus("Retrieving crack tip nodes")
78
79          self.crack_tip_nodes = []
80
81          if self.dimensions == "2d":
82              crack_seam = self.instance.nodeSets["CRACKSEAM_NODES"]
83              for node in crack_seam.nodes:
84                  self.crack_tip_nodes.append(node.label)
85
86          elif self.dimensions == "3d":
87              crack_seam_nodes = self.assembly.nodeSets["CRACKSEAM_NODES"]
88              for node_set in crack_seam_nodes.nodes:
89                  for node in node_set:
90                      self.crack_tip_nodes.append(node.label)
91
92          return self
93
94      def get_element_stresses_and_strains(self):
95          """
96          This function gets the stress and strain energy field outputs from the Abaqus ODB
         file,
97          and saves them to the element data dictionary.
98          """
99
100         self.print_abaqus("Getting element stress and strain data")
101
102         fields = [
103             ("S", "stress_tensor", self.shear_stress_factor),
104             ("EE", "strain_tensor", self.shear_strain_factor)]
105
106         for field_name, storage_key, factor in fields:
107
108             field = self.frame.fieldOutputs[field_name]
109             field_values = list(field.getSubset(position=INTEGRATION_POINT).values)
110
111             for value in field_values:
112                 elem_id = int(value.elementLabel)
113                 if storage_key not in self.elements[elem_id]:
114                     self.elements[elem_id][storage_key] = {}
115
116                 if factor:
117                     self.elements[elem_id][storage_key][int(value.integrationPoint)] =
        self._convert_stress_strain_data(value.data, factor)
118                 else:
119                     self.elements[elem_id][storage_key][int(value.integrationPoint)] =
        value.data
120
121         return self
122
123     def get_node_coordinates_and_displacements(self):
124         """
125         This function gets the coordinates of each node, along with the displacements,
```

```
        from the Abaqus ODB file. They are saved to the node data dictionary.
126         """
127
128         self.print_abaqus("Getting node coordinate and displacement data")
129
130         displacement_field = list(self.frame.fieldOutputs["U"].values)
131
132         for node in self.instance.nodes:
133             self.nodes[int(node.label)]["coordinates_original"] = self.
        _convert_coordinate_displacement_data(node.coordinates)
134
135         for displacement in displacement_field:
136             self.nodes[int(displacement.nodeLabel)]["displacement"] = self.
        _convert_coordinate_displacement_data(displacement.data)
137
138         return self
139
140     def get_element_and_node_connectivity(self):
141         """
142         This function creates a list for each element, containing the nodes which are
        connected to that element.
143         """
144
145         self.print_abaqus("Getting element and node connectivity")
146
147         # N1, N2, and N3 are the corner nodes.
148         # N4 is the node between N1 and N2, N5 is the node between N2 and N3, and N6 is
        the node between N1 and N3.
149         for element in self.instance.elements:
150             self.elements[int(element.label)]["connected_nodes"] = list(element.
        connectivity)
151
152         for element in self.elements:
153             for node in self.elements[element]["connected_nodes"]:
154                 if "connected_elements" not in self.nodes[node]:
155                     self.nodes[node]["connected_elements"] = []
156                 self.nodes[node]["connected_elements"].append(element)
157
158         for node in self.nodes:
159             self.nodes[node]["connected_elements"] = sorted(self.nodes[node]["
        connected_elements"])
160
161         return self
162
163     def save_output_data_to_json(self):
164         """
165         Save the processed results data to a JSON file inside the run directory.
166         """
167
168         self.print_abaqus("Saving exported data to JSON file", spacers=True)
169
170         self.output_data = {
171             "crack_tip_nodes": self.crack_tip_nodes,
172             "element_data": self.elements,
173             "node_data": self.nodes
174         }
175         filename = "Export_" + self.model_name + ".json"
176
177         output_data_path = os.path.join(self.run_dir, filename)
178
179         with open(output_data_path, "w") as f:
180             json.dump(self.output_data, f, indent=4)
181
182         return self
183
184     def print_abaqus(self, title=None, spacers=False, spacer_line = "-" * 120):
185
186         if spacers == True:
187             print >> sys.__stdout__, spacer_line
188
189         if title is not None:
190             print >> sys.__stdout__, title
191
```

```
            if spacers == True:
                print >> sys.__stdout__, spacer_line

        return self

    def _convert_stress_strain_data(self, data, factor):
        """
        Convert stresses and strains from the 1D array provided by Abaqus into a 2D array.
        Convert the strains from engineering to tensorial form.
        """

        if self.dimensions == "2d":
            return [
                [float(data[0]),          float(data[3] / factor)],
                [float(data[3] / factor), float(data[1])         ]]

        elif self.dimensions == "3d":
            return [
                [float(data[0]),          float(data[3] / factor), float(data[4] / factor)],
                [float(data[3] / factor), float(data[1]),          float(data[5] / factor)],
                [float(data[4] / factor), float(data[5] / factor), float(data[2])
                 ]]

        return 0

    def _convert_coordinate_displacement_data(self, data):
        """
        Get the coordinates and displacements relative to the crack tip. Usually the crack tip is at the origin in any case.
        """

        if self.dimensions == "2d":
            return [float(data[0] - self.crack_tip_coordinates[0]),
                    float(data[1] - self.crack_tip_coordinates[1])]

        elif self.dimensions == "3d":
            return [float(data[0] - self.crack_tip_coordinates[0]),
                    float(data[1] - self.crack_tip_coordinates[1]),
                    float(data[2] - self.crack_tip_coordinates[2])]

        return 0

if __name__ == "__main__":

    export = (
        ExportResults()
            .parse_arguments()
            .read_configuration_data()
            .load_model_parameters()
            .get_crack_tip_nodes()
            .get_element_stresses_and_strains()
            .get_node_coordinates_and_displacements()
            .get_element_and_node_connectivity()
            .save_output_data_to_json()
    )
```

6.5 Results Analysis

6.5.1 analyse_results.py

General class to perform the analysis step – calls the rest of the analysis classes in the correct order to perform the J-integral calculations.

```
import logging
logging.basicConfig(level=logging.INFO, format='%(message)s')

import numpy as np

from .mixins.domain_methods_mixin import DomainMethodsMixin
from .mixins.element_methods_mixin import ElementMethodsMixin
from .mixins.node_methods_mixin import NodeMethodsMixin
from .mixins.shape_function_methods_mixin import ShapeFunctionMethodsMixin

class AnalyseResults(DomainMethodsMixin,
                     ElementMethodsMixin,
                     NodeMethodsMixin,
                     ShapeFunctionMethodsMixin):

    def __init__(self,
                 dimensions,
                 input_data,
                 run_dir
):
        self.dimensions = dimensions
        self.input_data = input_data
        self.run_dir = run_dir

        if dimensions == "2d":
            self.integration_point_count = 3
            self.integration_point_weights = [1/6, 1/6, 1/6]

        elif dimensions == "3d":
            self.integration_point_count = 4
            self.integration_point_weights = [1/24, 1/24, 1/24, 1/24]


    def analyse_results(self):

        self.elements = self.input_data["element_data"].copy()
        self.nodes = self.input_data["node_data"].copy()

        logging.info("Collecting Parameters")
        (self
             .read_configuration_data()
             .filter_elements_and_nodes_by_integral_domain()
             .calculate_minimum_element_size()
             .calculate_displaced_nodal_coordinates()
             .get_element_nodal_coordinates()
             .convert_element_stress_and_strain_tensors()
             .define_integration_domains())

        logging.info("Computing Shape Functions")
        self.compute_shape_functions_and_derivatives()

        logging.info("Calculating Element Values")
        (self.calculate_element_integration_point_coordinates()
             .calculate_integration_point_jacobians()
             .calculate_element_strain_energy_densities()
             .calculate_element_displacement_gradients())

        logging.info("Performing Calculations")
        (self
             .calculate_element_weights()
             .calculate_domain_j_integrals()
```

```
            .calculate_domain_stress_intensity_factors()
            .calculate_analytical_values())

        self.output_data_dict = {
            "configuration": self.configuration,
            "stress_state": self.stress_state,
            "approx_min_element_size_mm": self.min_element_size_mm,
            "weight_function_type": self.weight_function_type,
            "applied_stress_mpa": np.abs(self.applied_stress_mpa),
            "crack_length_mm": self.crack_length_mm,
            "number_of_domains": self.number_of_domains,
            "minimum_domain_mm": np.round(self.domain_r_min, 4),
            "maximum_domain_mm": np.round(self.domain_r_max, 4),
            "results": {}
        }

        for domain_id, domain_data in self.domains.items():
            self.output_data_dict["results"][domain_id] = {}
            self.output_data_dict["results"][domain_id]["domain_r_min"] = domain_data["
    r_min"]
            self.output_data_dict["results"][domain_id]["domain_r_max"] = domain_data["
    r_max"]

            for param, unit, solution in ["j_integral", "", self.j_analytical], ["
    stress_intensity_factor", "MPa mm^1/2", self.sif_analytical]:
                val = domain_data[f'{param}']
                error = abs(((val - solution) / solution) * 100)
                self.output_data_dict["results"][domain_id][f"{param}_value_analytical"]
    = np.round(solution, 4)
                self.output_data_dict["results"][domain_id][f"{param}_value_calculated"]
    = np.round(val, 4)
                self.output_data_dict["results"][domain_id][f"{param}_error_percentage"]
    = np.round(error, 4)

        print("-" * 100)

        self.convert_results_to_dataframe()

        print("Analysis complete.")
        return self
```

6.5.2 domain_methods_mixin.py

This class provides the methods used to define the integration domain, and filter the elements and nodes to only capture those that fall within an integration domain, in order to increase performance.

```
import logging
import numpy as np
from .shared_methods_mixin import *
logging.basicConfig(level=logging.INFO, format='%(message)s')

class DomainMethodsMixin(SharedMethodsMixin):

    @SharedMethodsMixin.timeit
    def define_integration_domains(self):
        """
        Defines multiple annular integration domains based on the distance from the crack tip.
        Each domain is stored in a separate dictionary, with elements allowed in multiple domains.
        """
        logging.info("Defining integration domains around the crack tip.")

        crack_tip = np.array(self.crack_tip_coordinates)

        # Create an array containing the outer radii of each annular domain.
        radii = np.linspace(self.domain_r_min, self.domain_r_max, self.number_of_domains + 1)

        # Build the domains, storing the inner and outer radii of each domain.
        self.domains = {}
        for i in range(self.number_of_domains):
            r_min = radii[i]
            r_max = radii[i + 1]
            self.domains[i + 1] = {
                "r_min": r_min,
                "r_max": r_max,
                "elements": []
            }

        # For each element, if any nodal coordinate lies in the domain, include that element in the domain.
        for elem_id, elem_data in self.elements.items():
            for node_coords in elem_data["nodal_coordinates_original"]:
                distance = np.linalg.norm(node_coords[:2] - crack_tip[:2])
                for i in range(self.number_of_domains):
                    r_min = radii[i]
                    r_max = radii[i + 1]
                    if r_min <= distance <= r_max:
                        self.domains[i + 1]["elements"].append(elem_id)

        # Remove duplicate elements to ensure each element only appears once in each domain.
        for domain in self.domains:
            self.domains[domain]["elements"] = list(set(self.domains[domain]["elements"]))

        return self

    @SharedMethodsMixin.timeit
    def filter_elements_and_nodes_by_integral_domain(self, extension_factor=2.0):
        """
        Filters elements that are within a specified distance from the crack tip.

        args:
            extension_factor (float): The extension of the maximum domain size, for which elements are captured.
                                      i.e. an extension factor of 2.0 means all nodes/elements within double the radius
                                      of the domain are captured.
        """
```

```
58          logging.info("Filtering elements and nodes to keep only those within the maximum
    integral domain radius.")
59
60          # Get the actual minimum and maximum domain radii, rather than relative to the
    crack tip.
61          self.domain_r_min = self.domain_r_min_factor * self.crack_length_mm
62          self.domain_r_max = self.domain_r_max_factor * self.crack_length_mm
63
64          domain_elements, domain_nodes = [], []
65          filtered_elements, filtered_nodes = {}, {}
66
67          # Extend beyond the actual maximum integration domain by some factor to make sure
     that all nodes are captured.
68          r_max = self.domain_r_max * extension_factor
69
70          # If a node falls within the maximum domain, capture it, along with any elements
    it is connected to.
71          for node_id, node_data in self.nodes.items():
72              node_r = np.linalg.norm(np.array(node_data["coordinates_original"]) - self.
    crack_tip_coordinates)
73              if  node_r <= r_max:
74                  domain_nodes.append(int(node_id))
75                  for element in node_data["connected_elements"]:
76                      domain_elements.append(int(element))
77
78          domain_elements = sorted(list(set(domain_elements)))
79          domain_nodes = sorted(list(set(domain_nodes)))
80
81          for element in domain_elements:
82              filtered_elements[element] = self.elements[str(element)].copy()
83
84          for node in domain_nodes:
85              filtered_nodes[node] = self.nodes[str(node)].copy()
86
87          self.elements = filtered_elements
88          self.nodes = filtered_nodes
89
90          return self
```

6.5.3 element_methods_mixin.py

This class provides the methods used to operate on elements and integration points. It contains the bulk of the analysis performed.

```
import logging
import numpy as np
from .shared_methods_mixin import *
logging.basicConfig(level=logging.INFO, format='%(message)s')

class ElementMethodsMixin(SharedMethodsMixin):

    @SharedMethodsMixin.timeit
    def get_element_nodal_coordinates(self):
        """
        Get a set of all the nodal coordinates for each element.
        """
        logging.info("Retrieving the nodal coordinates for each element.")

        failed_elements = []

        for element_id, element_data in self.elements.items():

            # Store both the original and displaced nodal coordinates.
            nodal_coords = []
            nodal_coords_deformed = []

            for node in element_data["connected_nodes"]:
                try:
                    nodal_coords.append(self.nodes[node]["coordinates_original"])
                    nodal_coords_deformed.append(self.nodes[node]["coordinates_deformed"
    ])
                except:
                    failed_elements.append(element_id)

            element_data["nodal_coordinates_original"] = nodal_coords
            element_data["nodal_coordinates_deformed"] = nodal_coords_deformed

        for element in failed_elements:
            self.elements.pop(element, None)

        return self

    @SharedMethodsMixin.timeit
    def convert_element_stress_and_strain_tensors(self):
        """
        The stress and strain tensors in the export JSON file are stored as dictionaries
    due to the format.
        However, for calculations, it is more convenient to have them as arrays so they
    can be used with NumPy.
        """

        logging.info("Converting the element stress and strain tensors from dictionaries
    to arrays.")

        # Number of integration points per element.
        n_ip = len(self.integration_point_weights)

        for elem_id, elem_data in self.elements.items():

            for param in ["stress_tensor", "strain_tensor"]:
                tensor = []

                for ip in range(n_ip):
                    tensor.append(elem_data[param][str(ip+1)])

                elem_data[param] = tensor

        return self

    @SharedMethodsMixin.timeit
```

```
63      def calculate_element_integration_point_coordinates(self):
64          """
65          Get the coordinates of the integration points for each element, as an array.
66          """
67          logging.info("Calculating the integration point coordinates for each element.")
68
69          for element in self.elements:
70              nodal_coords = np.array(self.elements[element][f"nodal_coordinates_{self.
    configuration}"])
71              self.elements[element][f"integration_point_coordinates"] = np.dot(self.
    shape_functions_num, nodal_coords).tolist()
72
73          return self
74
75      @SharedMethodsMixin.timeit
76      def calculate_element_strain_energy_densities(self):
77          """
78          Computes the strain energy density W for each element at each integration point.
79          """
80          logging.info("Calculating the strain energy density for each element.")
81
82          for elem_id, elem_data in self.elements.items():
83
84              # Get the lists of stress and strain tensors, one per integration point.
85              stress_list = elem_data["stress_tensor"]
86              strain_list = elem_data["strain_tensor"]
87              n_ip = len(stress_list)
88              W_list = []
89
90              for ip in range(n_ip):
91                  stress_arr = np.array(stress_list[ip])
92                  strain_arr = np.array(strain_list[ip])
93
94                  # There is the option available here to convert from engineering to
    tensorial strain.
95                  # However, this is done during the data export.
96                  strain_factor = 1.0
97
98                  if self.dimensions == "2d":
99                      strain_arr_factored = np.array([
100                         [strain_arr[0][0]                 , strain_arr[0][1]/2 /
    strain_factor],
101                         [strain_arr[1][0] / strain_factor, strain_arr[1][1]
    ]
102                     ])
103                 elif self.dimensions == "3d":
104                     strain_arr_factored = np.array([
105                         [strain_arr[0][0]                 , strain_arr[0][1] /
    strain_factor, strain_arr[0][2] / strain_factor],
106                         [strain_arr[1][0] / strain_factor, strain_arr[1][1]
  , strain_arr[1][2] / strain_factor],
107                         [strain_arr[2][0] / strain_factor, strain_arr[2][1] /
    strain_factor, strain_arr[2][2]                 ]
108                     ])
109                 # Compute the strain energy density at this integration point.
110                 W = 0.5 * (np.tensordot(stress_arr, strain_arr_factored))
111                 W_list.append(W)
112
113             self.elements[elem_id]["strain_energy_density"] = W_list
114
115         return self
116
117     @SharedMethodsMixin.timeit
118     def calculate_integration_point_jacobians(self):
119         """
120         Compute the Jacobian matrix and determinant for each integration point of each
    element
121         using the shape function derivatives.
122         """
123         logging.info("Calculating the integration point Jacobians for each element.")
124
125         for element_id, element_data in self.elements.items():
126             nodal_coords = np.array(element_data[f"nodal_coordinates_{self.configuration}
```

```
        "])
127             jacobians = np.tensordot(self.shape_function_derivatives_num, nodal_coords,
        axes=([1], [0]))
128             det_jacobians = np.linalg.det(jacobians)
129
130             element_data[f"jacobian"] = jacobians.tolist()
131             element_data[f"jacobian_determinant"] = det_jacobians.tolist()
132
133         return self
134
135     @SharedMethodsMixin.timeit
136     def calculate_element_displacement_gradients(self):
137         """
138         Calculate the displacement gradients for each integration point of each element,
        using the
139         shape function derivatives, Jacobians, and nodal displacements.
140         """
141         logging.info("Calculating the displacement gradient for each element.")
142
143         for elem_id, elem_data in self.elements.items():
144             n_ip = len(self.integration_point_weights)
145             gradients_list = []
146             displacements = np.array([self.nodes[n][f"displacement"] for n in elem_data["
        connected_nodes"]])
147             jacobians = np.array(elem_data[f"jacobian"])
148
149             for ip in range(n_ip):
150                 invJ = np.linalg.inv(jacobians[ip])
151                 global_derivs = (invJ @ np.array(self.shape_function_derivatives_num)[ip
        ].T).T
152                 gradients_list.append((displacements.T @ global_derivs).tolist())
153
154             elem_data[f"displacement_gradients"] = gradients_list
155
156         return self
157
158     @SharedMethodsMixin.timeit
159     def calculate_element_weights(self):
160         """
161         Calculate the element weight function value and derivative for each integration
        point of each element,
162         for each annular domain.
163         """
164         logging.info("Calculating weights for each element based on distance to the crack
         tip.")
165         crack_tip_coordinates = np.array(self.crack_tip_coordinates)
166         dim = int(self.dimensions[0])
167
168         for elem_id, elem_data in self.elements.items():
169             ip_coords = elem_data.get(f"integration_point_coordinates", [])
170             weight_vals = {domain: [] for domain in self.domains}
171             weight_grads = {domain: [] for domain in self.domains}
172
173             for ip_coord in ip_coords:
174                 ip_coord = np.array(ip_coord)
175                 # Calculate the distance from the crack tip to the integration point, in
        the x-y plane only.
176                 r_vec = ip_coord[:2] - crack_tip_coordinates[:2]
177                 r = np.linalg.norm(r_vec)
178
179                 for domain in self.domains:
180                     r_min = self.domains[domain]["r_min"]
181                     r_max = self.domains[domain]["r_max"]
182
183                     # If the integration point is inside or outside the domain, gradient
        is zero, so no contribution.
184                     if r <= r_min:
185                         q = 1.0
186                         grad_q = np.zeros(dim)
187                     elif r >= r_max:
188                         q = 0.0
189                         grad_q = np.zeros(dim)
190                     else:
```

```
                        # Normalize the distance through the domain. s=0 is at the inside edge of the domain, s=1 is at
                        # the outside edge of the domain.
                        s = (r - r_min) / (r_max - r_min)
                        if self.weight_function_type == "linear":
                            q = 1 - s
                            dq_dr = -1.0 / (r_max - r_min)
                        elif self.weight_function_type == "polynomial":
                            q = 1 - 3 * s**2 + 2 * s**3
                            dq_dr = (-6 * s + 6 * s**2) / (r_max - r_min)
                        elif self.weight_function_type == "gaussian":
                            sigma = 0.4
                            q = np.exp(-(s / sigma) ** 2)
                            dq_dr = (-2 * s / (sigma ** 2) * np.exp(-(s / sigma) ** 2)) / (r_max - r_min)
                        else:
                            raise ValueError(f"Unknown weight_function_type: {self.weight_function_type}")

                        # Get the gradient of the weight function at the integration point.
                        grad_q = dq_dr * (r_vec / r)

                    weight_vals[domain].append(q)
                    weight_grads[domain].append(grad_q.tolist())

            elem_data[f"weight_function_values"] = weight_vals
            elem_data[f"weight_function_gradients"] = weight_grads

        return self

    @SharedMethodsMixin.timeit
    def calculate_domain_j_integrals(self):
        """
        Calculate the J-integral for each domain.
        """
        logging.info("Calculating J-integral for each integral domain.")

        for domain_id, domain_data in self.domains.items():

            domain_elements = domain_data["elements"]

            # Total J-integral for the domain.
            j_value = 0

            for elem_id in domain_elements:
                elem_data = self.elements[elem_id]
                stress_tensor = np.array(elem_data["stress_tensor"])
                displacement_grads = np.array(elem_data[f"displacement_gradients"])
                strain_energy_density = np.array(elem_data["strain_energy_density"])
                weight_grads = elem_data[f"weight_function_gradients"][domain_id]
                n_ip = len(elem_data[f"integration_point_coordinates"])

                for ip in range(n_ip):

                    # Given as 2x2 arrays. xx = [0, 0], xy = [0, 1], yx = [1, 0], yy = [1, 1]
                    sigma = stress_tensor[ip]
                    grad_u = displacement_grads[ip]
                    W = strain_energy_density[ip]
                    dq_dx, dq_dy = weight_grads[ip][0], weight_grads[ip][1]
                    detJ_ip = elem_data[f"jacobian_determinant"][ip]
                    w_ip = self.integration_point_weights[ip]

                    term_x = sigma[0, 0] * grad_u[0, 0] + sigma[0, 1] * grad_u[1, 0]
                    term_y = sigma[1, 0] * grad_u[0, 0] + sigma[1, 1] * grad_u[1, 0]

                    # Integrand for the individual integration point.
                    integrand = ((term_x - W) * dq_dx) + (term_y * dq_dy)

                    # Multiply by detJ and integration point weight to account for the relative area/volume.
                    j_value += integrand * detJ_ip * w_ip
```

```
258
259             domain_data[f"j_integral"] = j_value
260
261         return self
262
263     @SharedMethodsMixin.timeit
264     def calculate_domain_stress_intensity_factors(self):
265         """
266         Converts the J-integral value into the Stress Intensity Factor (SIF).
267         """
268         logging.info("Calculating the stress intensity factor for each integral domain.")
269
270         # Compute effective modulus E'
271         if self.stress_state == "plane_stress":
272             self.E_prime = self.material_E
273         elif self.stress_state == "plane_strain":
274             self.E_prime = self.material_E / (1 - self.material_nu**2)
275
276         for domain in self.domains:
277             self.domains[domain][f"stress_intensity_factor"] = np.sqrt((self.domains[
    domain][f"j_integral"]) * self.E_prime)
278
279         return self
```

6.5.4 node_methods_mixin.py

This class provides the methods used to operate on the nodes.

```
import logging
import numpy as np
from .shared_methods_mixin import *
logging.basicConfig(level=logging.INFO, format='%(message)s')

class NodeMethodsMixin(SharedMethodsMixin):

    @SharedMethodsMixin.timeit
    def calculate_displaced_nodal_coordinates(self):
        """
        Calculate the displaced coordinates for each node. Abaqus only gives the original coordinates along with the displacement vectors.
        """
        logging.info("Calculating the displaced coordinates for each node.")

        for node in self.nodes:
            self.nodes[node]["coordinates_deformed"] = (np.array(self.nodes[node]["coordinates_original"]) + np.array(self.nodes[node]["displacement"])).tolist()

        return self
```

6.5.5 shape_function_methods_mixin.py

This class provides the methods used to determine the shape functions and shape function derivatives for the elements.

```
import logging
import numpy as np
import sympy as sp
from .shared_methods_mixin import *
logging.basicConfig(level=logging.INFO, format='%(message)s')

class ShapeFunctionMethodsMixin(SharedMethodsMixin):

    @SharedMethodsMixin.timeit
    def compute_shape_functions_symbolically_cps6(self):
        """
        Generate the symbolic shape functions for a CPS6 (6-node triangular) element,
        as defined in the Abaqus theory manual, section 3.2.6.
        https://classes.engineering.wustl.edu/2009/spring/mase5513/abaqus/docs/v6.6/books/stm/default.htm?startat=ch03s02ath64.html
        """
        logging.info("Computing symbolic shape functions for triangular CPS6 elements.")

        x, y = sp.symbols('x y')
        L1 = x
        L2 = y
        L3 = (1 - x - y)

        N1 = -(L3 * (1 - (2 * L3)))
        N2 = -(L1 * (1 - (2 * L1)))
        N3 = -(L2 * (1 - (2 * L2)))
        N4 = 4 * L1 * L3
        N5 = 4 * L1 * L2
        N6 = 4 * L2 * L3

        self.shape_functions_sym = [N1, N2, N3, N4, N5, N6]

        return self

    @SharedMethodsMixin.timeit
    def compute_shape_functions_symbolically_c3d10(self):

        """
        Generate the symbolic shape functions for a C3D10 (10-node tetrahedral) element,
        as defined in the Abaqus theory manual, section 3.2.6.
        https://classes.engineering.wustl.edu/2009/spring/mase5513/abaqus/docs/v6.6/books/stm/default.htm?startat=ch03s02ath64.html
        """
        logging.info("Computing symbolic shape functions for tetrahedral C3D10 elements.")

        x, y, z = sp.symbols('x y z')

        w1 = y * ((2 * y) - 1)
        w2 = z * ((2 * z) - 1)
        w3 = (1 - x - y - z) * (1 - (2 * x) - (2 * y) - (2 * z))
        w4 = x * ((2 * x) - 1)
        w5 = 4 * y * z
        w6 = (4 * z) * (1 - x - y - z)
        w7 = (4 * y) * (1 - x - y - z)
        w8 = 4 * x * y
        w9 = 4 * x * z
        w10 = (4 * x) * (1 - x - y - z)

        self.shape_functions_sym = [w1, w2, w3, w4, w5, w6, w7, w8, w9, w10]

        return self

    @SharedMethodsMixin.timeit
    def compute_shape_function_derivatives_symbolically(self):
        """
```

```
64          Compute the symbolic derivatives of the shape functions with respect to the
    natural coordinates.
65          For each shape function w, this function computes the partial derivatives dw/dx
    and dw/dy (and dw/dz for 3D elements).
66          """
67          logging.info("Computing symbolic shape function derivatives.")
68
69          x, y, z = sp.symbols('x y z')
70          derivative_vars = (x, y) if self.dimensions == "2d" else (x, y, z)
71
72          self.shape_function_derivatives_sym = []
73
74          for w in self.shape_functions_sym:
75              w_derivs = []
76              for var in derivative_vars:
77                  w_derivs.append(sp.diff(w, var))
78              self.shape_function_derivatives_sym.append(w_derivs)
79
80          return self
81
82      @SharedMethodsMixin.timeit
83      def compute_shape_functions_and_derivatives_numerically(self):
84          """
85          Numerically evaluate the symbolic shape functions and their derivatives for an
    element
86          at given natural coordinate points.
87          """
88          logging.info("Performing numerical evaluation of shape functions and shape
    function derivatives.")
89
90          if self.dimensions == "2d":
91              get_sub_dict = lambda point: {'x': point[0], 'y': point[1]}
92              dim = 2
93          elif self.dimensions == "3d":
94              get_sub_dict = lambda point: {'x': point[0], 'y': point[1], 'z': point[2]}
95              dim = 3
96          else:
97              raise ValueError("Unsupported dimensions: {}".format(self.dimensions))
98
99          self.shape_functions_num = []
100         self.shape_function_derivatives_num = []
101
102         for point in self.natural_points:
103             sub_dict = get_sub_dict(point)
104
105             shape_function_single = [float(function.subs(sub_dict)) for function in self.
    shape_functions_sym]
106             shape_function_derivative_single = [
107                 [float(derivative[i].subs(sub_dict)) for i in range(dim)]
108                 for derivative in self.shape_function_derivatives_sym]
109
110             self.shape_functions_num.append(shape_function_single)
111             self.shape_function_derivatives_num.append(shape_function_derivative_single)
112
113         return self
114
115     def compute_shape_functions_and_derivatives(self):
116
117         """
118         Compute the shape function values and derivatives at predefined natural
    integration points,
119         as defined in section 4.1 of the Code_Aster documentation (https://code-aster.org
    /doc/v12/en/man_r/r3/r3.01.01.pdf).
120         """
121
122         if self.dimensions == "2d":
123
124             a = 1.0 / 6.0
125             b = 2.0 / 3.0
126
127             self.natural_points = np.array([
128                 (a, a),
129                 (b, a),
```

```
                (a, b)
            ])
            self.compute_shape_functions_symbolically_cps6()

        elif self.dimensions == "3d":

            a = (5 - np.sqrt(5)) / 20.0
            b = (5 + 3 * np.sqrt(5)) / 20.0

            self.natural_points = np.array([
                (a, a, a),
                (a, a, b),
                (a, b, a),
                (b, a, a)
            ])
            self.compute_shape_functions_symbolically_c3d10()

        self.compute_shape_function_derivatives_symbolically()
        self.compute_shape_functions_and_derivatives_numerically()

        return self
```

6.5.6 shared_methods_mixin.py

This class provides general helper methods used to read the configuration, print results to the terminal, etc.

```
import ast
import configparser
import logging
import os
import time
import numpy as np
import pandas as pd
from functools import wraps
logging.basicConfig(level=logging.INFO, format='%(message)s')

class SharedMethodsMixin:

    @staticmethod
    def timeit(func):
        """
        Function to give the time taken to perform each step of the calculation.
        """
        @wraps(func)
        def wrapper(*args, **kwargs):
            start_time = time.perf_counter()
            result = func(*args, **kwargs)
            elapsed_time = time.perf_counter() - start_time
            print(f"COMPLETED: {elapsed_time:.3f} seconds")
            print()
            return result
        return wrapper

    def print_spacers(self, title=None, spacers=False, spacer_line = "-" * 120):
            """
            Pretty print using spacers.
            """
            if spacers:
                print(spacer_line)
            if title:
                print(title)
                if spacers:
                    print(spacer_line)
            return 0

    def read_configuration_data(self):
        """
        Read in the configuration data from the config.ini file.
        """

        config_path = os.path.join(self.run_dir, "config.ini")

        config = configparser.ConfigParser()
        config.read(config_path)

        # Read analysis data.
        self.configuration = config.get("Analysis", "configuration")
        self.stress_state = config.get("Analysis", "stress_state")
        self.weight_function_type = config.get("Analysis", "weight_function_type")
        self.number_of_domains = int(config.get("Analysis", "number_of_domains"))
        self.domain_r_min_factor = float(config.get("Analysis", "domain_r_min_factor"))
        self.domain_r_max_factor = float(config.get("Analysis", "domain_r_max_factor"))

        # Read geometry data.
        self.part_height_mm = float(config.get("Geometry", "part_height_mm"))
        self.part_width_mm = float(config.get("Geometry", "part_width_mm"))
        self.part_thickness_mm = float(config.get("Geometry", "part_thickness_mm"))
        self.crack_length_mm = float(config.get("Geometry", "crack_length_mm"))
        self.crack_tip_coordinates = ast.literal_eval(config.get("Geometry", "
    crack_tip_coordinates"))

        if self.dimensions == "2d":
```

```
            self.crack_tip_coordinates = self.crack_tip_coordinates[:2]

        self.tolerance_mm = float(config.get("Geometry", "tolerance_mm"))

        # Read loading data.
        self.applied_stress_mpa = -abs(float(config.get("Loading", "applied_stress_mpa"))
    )

        # Read material data.
        self.material_name = config.get("Material", "material_name")
        self.material_E = float(config.get("Material", "material_E"))
        self.material_nu = float(config.get("Material", "material_nu"))

        # Read modelling data.
        self.partitioning_method = config.get("Modelling", "partitioning_method")
        self.step_name = config.get("Modelling", "step_name")
        self.run_job = config.getboolean("Modelling", "run_job")
        self.use_quarter_point_elements = config.getboolean("Modelling", "
    use_quarter_point_elements")

        # Read partitioning data.
        self.horizontal_partition_count = int(config.get("Partitioning", "
    horizontal_partition_count"))
        self.horizontal_partition_bias = float(config.get("Partitioning", "
    horizontal_partition_bias"))

        if self.dimensions == "2d" and self.partitioning_method == "circular":
            self.circular_partition_count = int(config.get("Partitioning", "
    circular_partition_count"))
            self.spoke_partition_count = int(config.get("Partitioning", "
    spoke_partition_count"))
            self.circular_partition_bias = float(config.get("Partitioning", "
    circular_partition_bias"))
            self.circle_inner_radius_factor = float(config.get("Partitioning", "
    circle_inner_radius_factor"))
            self.circle_outer_radius_factor = float(config.get("Partitioning", "
    circle_outer_radius_factor"))

        # Read meshing data.
        self.crack_element_count = int(config.get("Meshing", "crack_element_count"))
        self.crack_element_bias = float(config.get("Meshing", "crack_element_bias"))
        self.coarse_seed_size_mm = float(config.get("Meshing", "coarse_seed_size_mm"))

        if self.dimensions == "2d" and self.partitioning_method == "circular":
            self.circle_arc_element_count = int(config.get("Meshing", "
    circle_arc_element_count"))
            self.circle_spoke_element_count = int(config.get("Meshing", "
    circle_spoke_element_count"))

        elif self.dimensions == "3d":
            self.through_thickness_element_count = int(config.get("Meshing", "
    through_thickness_element_count"))

        return self

    @timeit
    def calculate_analytical_values(self):
        """
        Calculate an analytical value for the stress intensity factor using the formula
    obtained from
        'Research on the stress intensity factor of crack on 'the finite width plate with
     edge damage based on the finite element method'
        (Hong Yuan & Hao Zhang, 2023).
        Use the stress intensity factor to calculate a J-integral for the configuration.
        """
        logging.info("Calculating analytical values for the SIF and the J-integral.")

        a = self.crack_length_mm
        b = self.part_width_mm

        # Shape factor formula Beta for an edge-crack in a finite plate.
        F = 1.12 - (0.23 * a / b) + (10.6 * (a / b) ** 2) - (21.7 * (a / b) ** 3) + (30.4
     * (a / b) ** 4)
```

```
124
125             self.sif_analytical = F * np.abs(self.applied_stress_mpa) * np.sqrt(np.pi * a)
126             self.j_analytical = (self.sif_analytical ** 2) / self.E_prime
127
128             return self
129
130         @timeit
131         def calculate_minimum_element_size(self):
132             """
133             Calculate the minimum element size near the crack tip. Required for the mesh
        sensitivity study.
134             """
135             # Uniform mesh case: bias of 1 means no growth.
136             if self.crack_element_bias == 1:
137                 return self.crack_length_mm / float(self.crack_element_count)
138
139             # Calculate growth factor r from the bias ratio and number of elements.
140             r = self.crack_element_bias ** (1.0 / (self.crack_element_count - 1))
141
142             # Calculate the minimum element size using the geometric series sum formula.
143             self.min_element_size_mm = np.round(self.crack_length_mm * (r - 1) / (r**self.
        crack_element_count - 1), 3)
144
145             return self
146
147         def convert_results_to_dataframe(self):
148             """
149             Convert the results of the tool to a pandas DataFrame, so that charts can be more
         easily created.
150             """
151             rows = []
152             for domain_id_str, domain_data in self.output_data_dict["results"].items():
153
154                 row = {
155                     "configuration": self.output_data_dict["configuration"],
156                     "stress_state": self.output_data_dict["stress_state"],
157                     "weight_function_type": self.output_data_dict["weight_function_type"],
158                     "crack_length_mm": self.output_data_dict["crack_length_mm"],
159                     "minimum_element_size_mm": self.min_element_size_mm,
160                     "domain_id": int(domain_id_str),
161                     "domain_r_min": domain_data["domain_r_min"],
162                     "domain_r_max": domain_data["domain_r_max"],
163                     "j_integral_value_analytical": domain_data["j_integral_value_analytical"
        ],
164                     "j_integral_value_calculated": domain_data["j_integral_value_calculated"
        ],
165                     "j_integral_error_percentage": domain_data["j_integral_error_percentage"
        ],
166                     "stress_intensity_factor_value_analytical": domain_data["
        stress_intensity_factor_value_analytical"],
167                     "stress_intensity_factor_value_calculated": domain_data["
        stress_intensity_factor_value_calculated"],
168                     "stress_intensity_factor_error_percentage": domain_data["
        stress_intensity_factor_error_percentage"],
169                 }
170                 rows.append(row)
171
172             self.output_data_df = pd.DataFrame(rows)
173             self.output_data_df.sort_values(by="domain_id", inplace=True)
174             self.output_data_df.reset_index(drop=True, inplace=True)
175
176             return self
```

6.6 Run Management

6.6.1 runner.py

General class to manage the run of the tool. Reads in the arguments, creates the directories to hold the results, and then calls the correct functions to perform the analysis.

```
import argparse
import json
import os
import pathlib
import shutil
import subprocess
from analysis.analyse_results import *

class Runner:

    def __init__(self, run_type):
        self.run_type = run_type
        self.base_dir = pathlib.Path(__file__).resolve().parent.parent.parent

    def print_spacers(self, title=None, spacers=False, spacer_line = "-" * 120):
            if spacers:
                print(spacer_line)
            if title:
                print(title)
                if spacers:
                    print(spacer_line)
            return 0

    def parse_arguments(self):

        parser = argparse.ArgumentParser(
            description="Run export and/or analysis for an Abaqus job."
        )
        parser.add_argument(
            "--dimensions",
            help="Perform the export step (process the Abaqus output file).")

        parser.add_argument(
            "--model",
            help="The model to be used for the export or analysis.")

        parser.add_argument(
            "--input",
            help="Directory number to use for existing results.")

        parser.add_argument(
            "--config",
            help="Filename of the config file to use"
        )

        args = parser.parse_args()

        self.dimensions = args.dimensions
        self.model_name = "_".join(word.capitalize() for word in args.model.split("_")) +
     f"_{self.dimensions.upper()}"

        if args.input:
            self.input = args.input
        else:
            self.input = None

        if args.config:
            self.config_path = args.config
        else:
            self.config_path = os.path.join(self.base_dir, "config", "config.ini")

```

```
61          return self
62
63      def get_last_directory_number(self, dir, prefix):
64          """
65          Get the number of the last directory. Directories are incremented progressively
    for each analysis,
66          i.e. 001, 002 etc. We need the last number to create the new directory.
67          """
68
69          existing_dirs = [
70                  d for d in os.listdir(dir)
71                  if d.startswith(prefix.title()) and d[len(prefix.title())+1:].isdigit()
72              ]
73
74          existing_numbers = [int(d[-3:]) for d in existing_dirs] if existing_dirs else []
75          last_number = max(existing_numbers, default=0)  # Increment highest number
76
77          # Format the new directory name with zero padding (e.g., "001", "002", etc.)
78          return last_number
79
80      def create_run_directories(self):
81          """
82          Create a directory for this specific run of the tool, and populate with
    subdirectories.
83          """
84
85          self.print_spacers(f"Creating Directories for Job: {self.model_name}", spacers=
    True)
86
87          self.data_dir = os.path.join(self.base_dir, "data")
88
89          if self.run_type == "analysis":
90              self.analysis_dir = os.path.join(self.data_dir, "analysis")
91          elif self.run_type == "export":
92              self.analysis_dir = os.path.join(self.data_dir, "exports")
93          elif self.run_type == "model":
94              self.analysis_dir = os.path.join(self.data_dir, "models")
95
96          self.model_dir = os.path.join(self.analysis_dir, self.model_name)
97
98          dirs = [
99              self.data_dir,
100             self.analysis_dir,
101             self.model_dir,
102         ]
103
104         for dir in dirs:
105             try:
106                 os.makedirs(dir, exist_ok=False)
107                 print(f"Created directory -> {dir}")
108             except OSError:
109                 print(f"Directory exists  -> {dir}")
110                 os.makedirs(dir, exist_ok=True)
111
112         next_number = self.get_last_directory_number(self.model_dir, self.run_type) + 1
113
114         run_dir_name = f"{self.run_type.title()}_{next_number:03d}"
115         self.run_dir = os.path.join(self.model_dir, run_dir_name)
116
117         try:
118             os.makedirs(self.run_dir, exist_ok=False)
119             print(f"Created directory -> {self.run_dir}")
120         except OSError:
121             print(f"Directory exists  -> {self.run_dir}")
122             os.makedirs(self.run_dir, exist_ok=True)
123
124         return self
125
126     def copy_config_data(self):
127         """
128         Copy the configuration data from the old directory to the new. We want to keep
    track of which configuration was used for each analysis.
129         """
```

```
130
131         src_config_file = self.config_path
132         dest_config_file = os.path.join(self.run_dir, "config.ini")
133
134         if os.path.isfile(src_config_file):  # Ensure it's a file, not a directory
135                 shutil.copy2(src_config_file, dest_config_file)
136                 print(f"Copied: {src_config_file} -> {dest_config_file}")
137
138         return self
139
140
141     def copy_and_read_input_data(self):
142         """
143         If the run is being performed with some already existing exported data as an
    input, this function copies that data
144         to the run directory, to be used as the input to the analysis.
145         """
146
147         self.print_spacers("Copying Exported Data to the Run Directory", spacers=True)
148
149         if self.run_type == "analysis":
150             input_data_dir = "exports"
151             prefix = "Export"
152         elif self.run_type == "export":
153             input_data_dir = "models"
154             prefix = "Model"
155
156         if self.input:
157             last_number = str(self.input).zfill(3)
158         else:
159             last_number = str(self.get_last_directory_number(os.path.join(self.data_dir,
    input_data_dir, self.model_name), prefix + "_")).zfill(3)
160             print(last_number)
161
162         data_dir_name = prefix + "_" + last_number
163
164         input_data_path = os.path.join(
165             self.data_dir,
166             input_data_dir,
167             self.model_name,
168             data_dir_name,
169         )
170
171         output_data_path = os.path.join(
172             self.run_dir,
173             data_dir_name
174         )
175
176         try:
177             os.makedirs(output_data_path, exist_ok=False)
178             print(f"Created directory -> {output_data_path}")
179         except OSError:
180             print(f"Directory exists  -> {output_data_path}")
181             os.makedirs(output_data_path, exist_ok=True)
182
183         print("Copying exported data to the run directory...")
184         for file in os.listdir(input_data_path):
185
186             src_file = os.path.join(input_data_path, file)
187
188             if file == "config.ini":
189                 dest_file = os.path.join(self.run_dir, file)
190             else:
191                 dest_file = os.path.join(output_data_path, file)
192
193             if os.path.isfile(src_file):  # Ensure it's a file, not a directory
194                 shutil.copy2(src_file, dest_file)
195                 print(f"Copied: {src_file} -> {dest_file}")
196
197             elif os.path.isdir(src_file):  # Copy directories properly
198                 dest_file = os.path.join(self.run_dir, file)
199                 shutil.copytree(src_file, dest_file, dirs_exist_ok=True)
200                 print(f"Copied directory: {src_file} -> {dest_file}")
```

```
201
202         if self.run_type == "analysis":
203             print("Reading the exported data into memory...")
204             json_file = os.path.join(output_data_path, prefix + "_" + self.model_name + "
    .json")
205             with open(json_file, "r") as f:
206                 self.input_data = json.load(f)
207         elif self.run_type == "export":
208             self.odb_path = os.path.join(output_data_path, self.model_name + ".odb")
209
210         return self
211
212     def submit_abaqus_command(self):
213         """
214         This method submits the command to call the other script and run Abaqus. The
    export script is called with no GUI, the modelling script is called
215         with the GUI.
216         """
217
218         self.print_spacers(f"Connecting to Abaqus", spacers=True)
219
220         self.src_dir = os.path.join(self.base_dir, "src")
221
222         if self.run_type == "export":
223             script_path = os.path.join(self.src_dir, "export", "export_results.py")
224             self.abaqus_command = f"cd {self.base_dir} && abaqus cae noGUI={script_path}
    -- {self.dimensions} {self.model_name} {self.run_dir} {self.odb_path}"
225
226         elif self.run_type == "model":
227             script_path = os.path.join(self.src_dir, "model", self.model_name.lower() + "
    .py")
228             self.model_path = os.path.join(self.run_dir, self.model_name + ".cae")
229             self.abaqus_command = f"cd {self.base_dir} && abaqus cae script={script_path}
     -- {self.dimensions} {self.model_name} {self.run_dir} {self.model_path}"
230
231         subprocess.run(self.abaqus_command, shell=True, check=True)
232
233         return self
234
235     def run_analysis(self):
236
237         self.print_spacers(f"Running Analysis for {self.dimensions.upper()} Scenario",
    spacers=True)
238         self.analysis = AnalyseResults(self.dimensions, self.input_data, self.run_dir).
    analyse_results()
239
240         return self
241
242     def save_output_data_to_json(self):
243         """
244         Save the processed results data to a JSON file inside the run directory.
245         """
246
247         self.results_filename = "Analysis_" + self.model_name + ".json"
248         self.results_filename_csv = "Analysis_" + self.model_name + ".csv"
249
250         output_data_path = os.path.join(self.run_dir, self.results_filename)
251         output_data_path_csv = os.path.join(self.run_dir, self.results_filename_csv)
252
253         with open(output_data_path, "w") as f:
254             json.dump(self.analysis.output_data_dict, f, indent=4)
255
256         import pandas as pd
257
258         self.analysis.output_data_df.to_csv(output_data_path_csv)
259
260         self.print_spacers(f"Results data saved to -> {output_data_path}")
261         self.print_spacers(spacers=True)
262
263         return self
```

6.6.2 `analyse.py`

Simple module to perform the analysis step. Called directly using `python analyse.py`.

```
from runner.runner import Runner

if __name__ == "__main__":

    run = (
        Runner(run_type = "analysis")
            .parse_arguments()
            .create_run_directories()
            .copy_and_read_input_data()
            .run_analysis()
            .save_output_data_to_json()
    )
```

6.6.3 `export.py`

Simple module to perform the export step. Called directly using `python export.py`.

```
from runner.runner import Runner

if __name__ == "__main__":

    run = (
        Runner(run_type = "export")
            .parse_arguments()
            .create_run_directories()
            .copy_and_read_input_data()
            .submit_abaqus_command()
    )
```

6.6.4 `model.py`

Simple module to perform the modelling step. Called directly using `python model.py`.

```
from runner.runner import Runner

if __name__ == "__main__":

    run = (
        Runner(run_type = "model")
            .parse_arguments()
            .create_run_directories()
            .copy_config_data()
            .submit_abaqus_command()
    )
```