NASA/TM–2011-216879

Surrogate Modeling of High-Fidelity Fracture Simulations for Real-Time Residual Strength Predictions

Ashley D. Spear, Amanda R. Priest, Michael G. Veilleux, and Anthony R. Ingraffea
Cornell University, Ithaca, New York

Jacob D. Hochhalter
NASA Langley Research Center, Hampton, Virginia

January 2011

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include creating custom thesauri, building customized databases, and organizing and publishing research results.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at *http://www.sti.nasa.gov*

- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA STI Help Desk at 443-757-5803

- Phone the NASA STI Help Desk at 443-757-5802

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/TM–2011-216879

Surrogate Modeling of High-Fidelity Fracture Simulations for Real-Time Residual Strength Predictions

Ashley D. Spear, Amanda R. Priest, Michael G. Veilleux, and Anthony R. Ingraffea
Cornell University, Ithaca, New York

Jacob D. Hochhalter
NASA Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

January 2011

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

Surrogate Modeling of High-Fidelity Fracture Simulations for Real-time Residual Strength Predictions

Ashley D. Spear[1], Amanda R. Priest[2], Michael G. Veilleux[3] and Anthony R. Ingraffea[4]
Cornell University, Ithaca, NY 14853

Jacob D. Hochhalter[5]
NASA Langley Research Center, Hampton, VA 23681

A surrogate model methodology is described for predicting in real time the residual strength of flight structures with discrete-source damage. Starting with design of experiment, an artificial neural network is developed that takes as input discrete-source damage parameters and outputs a prediction of the structural residual strength. Target residual strength values used to train the artificial neural network are derived from 3D finite element-based fracture simulations. Two ductile fracture simulations are presented to show that crack growth and residual strength are determined more accurately in discrete-source damage cases by using an elastic-plastic fracture framework rather than a linear-elastic fracture mechanics-based method. Improving accuracy of the residual strength training data would, in turn, improve accuracy of the surrogate model. When combined, the surrogate model methodology and high fidelity fracture simulation framework provide useful tools for adaptive flight technology.

Nomenclature

---

[1] Graduate Student, School of Civil and Environmental Engineering, 640 Rhodes Hall, AIAA Student Member.
[2] Undergraduate Student, School of Civil and Environmental Engineering, 640 Rhodes Hall.
[3] Graduate Student, School of Civil and Environmental Engineering, 640 Rhodes Hall.
[4] Dwight C. Baum Professor of Engineering, School of Civil and Environmental Engineering, 643 Rhodes Hall.
[5] Research Materials Engineer, Durability & Damage Tolerance Branch, MS 188E, AIAA Member.

E = elastic modulus (GPa)

ν = Poisson's ratio (mm/mm)

σ_y = yield stress (MPa)

$STRI65$ = quadratic triangular shell element in ABAQUS [1]

$S8R$ = quadratic reduced-integration shell element in ABAQUS [1]

$C3D10$ = quadratic tetrahedral elements in ABAQUS [1]

$C3D15$ = quadratic wedge element in ABAQUS [1]

$C3D20(R)$ = quadratic brick element (reduced-integration) in ABAQUS [1]

a = crack length (cm)

n = number of cracks in discrete-source damage

θ = orientation of discrete-source damage, angle between positive x axis and nearest crack

d_x = distance from middle stiffener to center of discrete-source damage (cm)

K_I, K_{II}, K_{III} = mode I, II, and III plane strain stress intensity factors (MPa$\sqrt{m}$)

K_{Ic}, K_{IIc} = plane strain fracture toughness for modes I and II (MPa$\sqrt{m}$)

P_{max} = damage-dependent allowable traction, residual strength (MPa)

P = applied traction (MPa)

MSE = mean squared error as defined by Eq. (2)

c_v = correlation coefficient as defined by Eq. (3)

CTD = magnitude of relative displacement between upper and lower fracture surfaces (mm)

CTD_{crit} = critical value of CTD (mm)

$CTD_I, CTD_{II}, CTD_{III}$ = opening, in-plane sliding, out-of-plane shearing components of CTD (mm)

d = fixed characteristic distance behind crack front where CTD is monitored (mm)

da = crack extension (mm)

(n) = script to denote mesh at n^{th} crack increment

$(n+1)$ = script to denote mesh at $(n+1)^{th}$ crack increment

I. Introduction

Resilient aircraft control involves adaptive responses to off-nominal flight conditions, including the incurrence of structural discrete-source damage during flight. *Discrete-source damage* is typically manifested as a result of a structural impact event, including hail- and birdstrike. In 2003, an Airbus A300 operated by DHL was struck by a surface-to-air missile after takeoff from Baghdad, Iraq, causing discrete-source damage to crucial control surfaces of the left wing [2]. In 2008, a Boeing 747-438 operated by Qantas Airways incurred in-flight structural damage to the fuselage and right wing leading edge following the failure of an onboard oxygen cylinder [3]. Although the aircraft landed safely in both cases, these examples motivate a need for more resilient, adaptive control system responses.

In these types of cases, problems associated with in-flight discrete-source damage, for example inability to sustain original design loads, can be exacerbated by crack propagation from damaged regions. To avoid unstable crack propagation, load levels must be maintained below a reduced load-carrying capacity, or *residual strength*, of damaged flight structures. Adaptive control system responses might include automatic adjustment of certain flight parameters (e.g. velocity, maximum acceleration) to accommodate structural residual strength. This accommodation implies that accurate residual strength predictions of flight structures with complex damage configurations be made *in real time, during flight*; this capability currently does not exist for commercial aviation.

Challenges to developing an adaptive response technology include accurately predicting residual strength of discrete-source damaged structures both *offline* (i.e. during control system design) and *online* (i.e. in real time onboard the aircraft). In the offline context, researchers have developed various tools for determining residual strength of thin, damaged metallic structures using elastic-plastic fracture mechanics (EPFM)-based numerical methods. For example, two common finite element (FE) modeling techniques involve nodal release and adaptive remeshing. Both techniques represent cracks geometrically [4]. The former, however, prescribes possible crack trajectories, which introduces inherent mesh dependencies into fracture simulations and limits generality of crack path predictions. Nodal release techniques have been used in 2D [5–12] and in 3D [13–15] for studying crack growth parameters and predicting residual strength of structures where the crack path was

known *a priori* and where mesh refinement along the crack path sufficiently characterized growth increments. Adaptive remeshing techniques avoid such mesh dependencies and enable simulation of arbitrary crack propagation using evolutionary models or criteria [16–20]. Adaptive remeshing techniques have been implemented in both 2D [21] and in 3D [22, 23]. Of the described techniques, 3D, adaptively remeshed, elastic-plastic tearing simulations provide the most general prediction capabilities for crack growth and residual strength.

It is infeasible to perform a rigorous and computationally intensive crack growth simulation within the possible short time span following a discrete-source damage event. Thus, an approximation, or *surrogate model*, is needed for making online predictions of residual strength. Queipo *et al.* provided a complete description of surrogate modeling development and optimization [24]. With regard to surrogate construction, they described both parametric (e.g. polynomial regression and Kriging) and nonparametric (e.g. radial basis functions) approaches. In nonparametric approaches, a global functional form relating system input to system response is not assumed.

Artificial neural networks (NNs) are a nonparametric surrogate modeling approach and are trained to infer a nonlinear mapping from system input to system response, or output. The reader is referred to [25] for an extensive methodology overview of the most widely used types of NN. Different types of NNs have been applied extensively for damage detection [26–32] and, to a much lesser extent, for damage assessment. Ouenes *et al.* employed a NN methodology to predict fracture indicators (e.g. density of fractures) in naturally fractured rock reservoirs as a function of various geological and geophysical data [33]. Pidaparti *et al.* employed a NN to predict residual strength and corrosion rate of aging aircraft panels with collinear multi-site damage by training with experimental results and validating with both experimental results and analytical solutions [34]. Recently, Mohanty *et al.* used a Gaussian process (GP) approach to predict fatigue crack growth in aluminum 2024-T351 specimens by training two distinct models, one presented with experimental load parameters as input and another presented with piezoelectric sensor signals as input [35]. In that work, Mohanty *et al.* used observed fatigue crack lengths and growth rates as known output for training each model.

Alternatively, NNs can be trained using results from numerical experiments, or simulations [36]. For example, Sankararaman *et al.* recently used linear-elastic fracture parameters computed from

FE analyses to train a GP model as part of a method to statistically infer equivalent initial flaw size in fatigue applications [37]. High-fidelity numerical simulations can provide training data when analytically- and experimentally-derived data is limited due either to a lack of generally applicable analytical solutions or to prohibitive costs of obtaining sufficient experimental data.

The purpose of the work presented here is two-fold: (1) to illustrate a methodology for creating a surrogate model as a real-time residual strength prediction tool and (2) to describe and validate numerical tools for making accurate residual strength predictions offline using fully 3D, elastic-plastic, FE-based crack growth simulations. The high-fidelity, more computationally expensive tools described in (2) can provide training data that, when coupled with the surrogate model methodology described in (1), can be used in the design of adaptive response technology.

Consistent with our two-fold purpose, this paper is divided into two primary sections. Section II illustrates the methodology for developing a surrogate model (in particular, a NN) that predicts residual strength as a function of discrete-source damage parameters. The methodology is illustrated using a relatively simple proof-of-concept example. The procedure for gathering training data is described in II A and II B. Because an implementation-ready NN is beyond the scope of this paper, training data for the proof-of-concept example relies on reduced-order residual strength approximations. After collecting training data, a simple NN is constructed in II C by optimizing certain performance parameters. Finally, a sensitivity study is conducted in II D to understand the effect of each damage parameter on predicted residual strength specifically for the proof-of-concept structure.

Section III improves upon offline residual strength prediction tools used in Section II by simulating 3D, elastic-plastic tearing. The tools provide more general crack growth simulation capabilities and can be used to generate accurate residual strength training data. Simulation results are validated in III C for a mixed-mode I/II fracture test and for a relatively large, integrally-stiffened panel (ISP) that exhibits crack branching.

Results and discussions from the NN proof-of-concept example and from the elastic-plastic tearing simulations are provided in each respective section. Section IV offers a summary and conclusions for the entirety of this work.

II. Neural Network Development and Methodology

This section describes the development of a surrogate model for predicting residual strength of discrete-source damaged aircraft structures in real time. A global functional form is not assumed for the nonlinear relationship between residual strength and the damage parameters influencing it; thus, a nonparametric surrogate model is developed. In particular, a supervised NN is considered due to rapid prediction capabilities amenable to real-time applications. In Fig. 1, the upper dashed region shows the generalized procedure for developing a NN (surrogate model) that predicts residual strength as a function of parameterized discrete-source damage. The lower dashed region shows the functionality of the NN (surrogate model) in a real-time context.

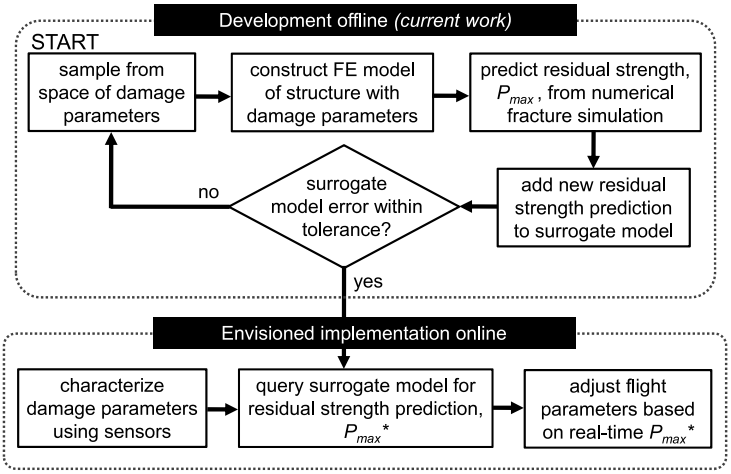


Fig. 1 Upper dashed box illustrates a general approach for developing a surrogate model to predict residual strength of damaged structures. Lower dashed box illustrates how the surrogate model would function onboard an aircraft for predicting residual strength of damaged structures in real time.

The first step in this type of surrogate model development is typically referred to as design of experiment (DOE) [24] and involves obtaining data points that will be used to train and test the NN. The DOE should be based on the intended application of the NN. For example, if the NN is intended to provide residual strength predictions in terms of maximum allowable bending moment in a damaged aircraft wing, then the data points should be gathered using an appropriate wing structure with applied boundary conditions of interest. Each data point includes sampled input variable(s) and corresponding known system response(s), called *target output*. Once the NN has been trained to map given input to target output, it becomes a useful tool for predicting system

response when presented with new input that is within the training range but does not necessarily correspond to data points used for training.

To illustrate the methodology, a simple NN is developed using a representative wing structure and reduced-order (linear-elastic) approximations for predicting residual strength. The representative wing structure is a 61.0 x 91.4 cm^2 integrally-stiffened panel (ISP) with three blade stiffeners each 5.1 cm in height, as shown in Fig. 2. The ISP skin and stiffeners are 2.3 mm thick. The panel is modeled as linear-elastic with $E = 71.0$ GPa and $\nu = 0.33$, similar to values for a $2XXX$ series lower wing skin aluminum alloy (AA).

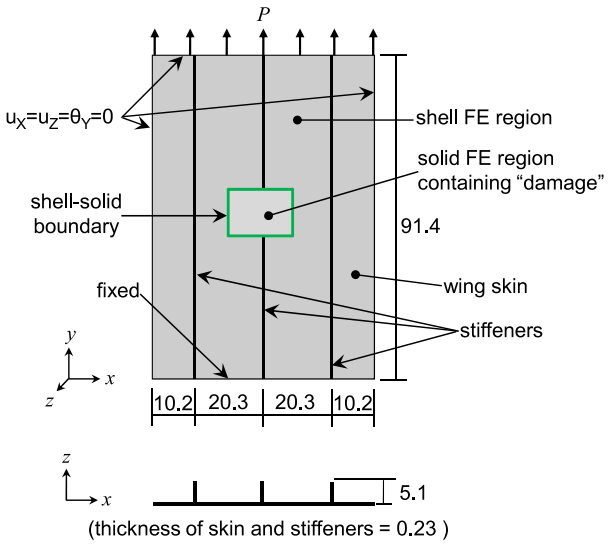


Fig. 2 Schematic of ISP model with dimensions similar to those in [38]. Plan view (top) and cross-section showing integral blade stiffeners (bottom). A damage-containing region is modeled using 3D solid elements (enclosed in shell-solid boundary) while remainder of panel is modeled with shell elements. All dimensions in cm.

Multiple FE models of the uncracked panel are constructed using ABAQUS® [1]. A shell-solid modeling technique is employed, where each panel is modeled using 3D solid elements in a region that will contain damage and shell elements elsewhere, as depicted schematically in Fig. 2. In this way, 3D constraint is inherently captured along crack fronts using fully 3D solid elements, while shell elements help maintain a level of computational efficiency without losing capability to capture out-of-plane deformation and possible buckling. The shell and solid element regions are joined using a coupling constraint, whereby resultant forces and moments acting at shell edge nodes on the shell-solid boundary are distributed as forces acting at nodes located in a region of influence on the

solid surface of the shell-solid boundary. A mesh refinement study is carried out to ensure adequate discretization of the panel models. *Uncracked* panels are modeled using approximately 50 *STRI65*, 2000 *S8R*, and between 1800 and 17,300 *C3D20R* elements, depending on the size of the damaged region. Boundary conditions for the ISP models are defined to emulate tensile loading conditions for a region of the lower wing surface and are shown schematically in Fig. 2.

A supplementary study was carried out to determine shell-solid boundary effects on nearby crack fronts in order to minimize the size of the solid region without affecting stress intensity factors (SIFs) computed along nearby crack fronts. Maintaining fracture parameter accuracy is especially important since fracture parameters are used to predict structural residual strength (described in II B), which is in turn used to train the NN (described in II C). The supplementary study considered a 61.0 x 91.4 cm^2 unstiffened panel of the same (linear-elastic) material and thickness as the ISP described above. The panel had a single, 12.7 cm long, centrally-located through-crack oriented in the x direction (normal to applied tensile load). Both tensile and bending conditions were considered in the study. The panel was modeled entirely with shell elements except for a region containing the crack, which was modeled with 3D solid elements. All model parameters remained constant while varying the size (both in-plane dimensions) of the square-shaped solid region, and therefore the distance from the shell-solid boundary to the crack front. The size of the solid region was initially slightly larger than the crack length and was increased until computed SIFs converged. Results from the supplementary study indicated that for a static, linear-elastic crack, the distance from shell-solid boundary to nearest crack front should be no less than 25% of the crack length. This ensures that the shell-solid boundary has negligible effect on computed SIFs. The same rule-of-thumb is applied to the example ISP models described in the NN study.

The following sections describe the generally applicable methodology for developing a NN as a real-time residual strength prediction tool.

A. Input Variables: Discrete-source Damage Parameters

Discrete-source damage in this work is represented by a symmetric, star-shaped, array of equi-length cracks, as depicted in Fig. 3(b). This representation of discrete-source damage is motivated

by observations of petaling caused by penetration damage to thin metallic structures, see Fig. 3(a). If all of the cracks in the star-shaped array of Fig. 3(b) separate under load (i.e. there are no crack closure effects), then the cracked region transfers no load and effectively represents a circular hole with petaling edges, similar to that shown in Fig. 3(a). The damage representation is parameterized by the four variables n, a, d_x, and θ, which are postulated to influence residual strength of the ISP.

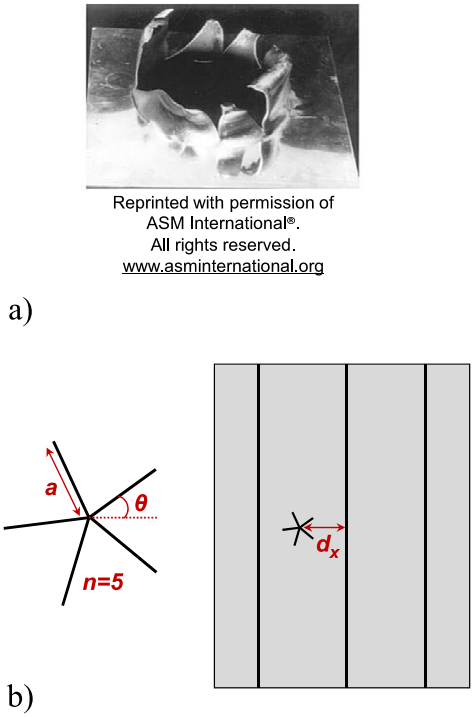


Fig. 3 (a) Petaling on the reverse side of a metallic sheet subject to explosive, discrete-source damage [39]. (b) Schematic showing the representation and parameterization of discrete-source in the NN example described in this work.

The sample space of damage configurations is defined by a range of values for each parameter. Ranges can be specified based on accident reports, photographic evidence, potential structural threats, design specifications, and so forth. Inherently, the NN predictions are valid only for input parameter values within the range of training data. Thus, it is necessary to define the sample space based on the particular NN application. In the example NN, ranges for each damage parameter are limited to some extent by the ISP geometry. Each range is given in Table 1. The parameter n, takes integer values ranging from two to six. The range of θ depends on n due to the definition of orientation and the symmetry of the star-shaped configuration. The range of a is defined in terms of ISP bay width, from $1/8 * baywidth$ to $1/4 * baywidth$. Due to symmetry of the ISP

model, the parameter d_x ranges from 10.2 cm (damage centered in mid-bay) to 0 (damage centered at middle stiffener). If the damage is located such that the damage-containing, solid FE region overlaps anywhere with the middle stiffener, the stiffener is assumed to be severed in the damaged region and is modeled explicitly as such.

Table 1 Range of values associated with each damage parameter in the example NN.

Damage Parameter	Range
n	2-6
θ: $n = 2$ (deg)	0-90
θ: $n = 3$ (deg)	0-60
θ: $n = 4$ (deg)	0-45
θ: $n = 5$ (deg)	0-36
θ: $n = 6$ (deg)	0-30
a (cm)	1.27-5.08
d_x (cm)	0-10.2

The damage parameter space is sampled to obtain damage configurations, each expressed as a combination of input parameters (n, θ, a, d_x). The space of variables can be sampled using a number of different sampling methods, including random, stratified, and Latin Hypercube [40]. Latin Hypercube Sampling (LHS) is a type of stratified sampling method that guarantees each partition, or stratum, of input variable space is sampled, though not necessarily uniformly. In this work, LHS is performed five times for each of the variables (θ, a, and d_x). Each of the five LHS runs corresponds to a different value of n (two, ..., six cracks) and requires the number of partitions to be specified. The MATLAB® implementation for LHS is used here [41], where output is provided in the range from zero to one. Each sample value is then scaled to the respective parameter range according to Table 1.

Table 2 shows all damage configurations (26 in total) that are modeled in the ISP NN example, where each configuration is expressed in terms of sampled input parameters. For each damage configuration, the x and y dimensions of the square, damage-containing, solid FE region are provided in the sixth column. The x and y dimensions are each 25% larger than the diameter of the star-shaped damage (i.e. $1.25 * 2a$), as suggested by the supplementary shell-solid boundary effect study described above. The last column specifies whether or not the solid, damaged region severs the middle stiffener. If so, the portion of the stiffener that intersects the solid model region is removed;

otherwise the stiffener remains intact.

Table 2 Damage configurations modeled in the ISP NN example. Each damage configuration is assigned an alphanumeric identification with number corresponding to n. The sixth column provides x and y dimensions of the square region in the shell-solid ISP.

Damage configuration ID	a (cm)	d_x (cm)	θ (deg)	n	Solid region x,y dimensions (cm)	Severs stiffener?
2A	3.8	7.1	21.8	2	9.39	NO
2B	4.2	2.1	87.8	2	10.48	YES
2C	3.0	3.6	2.2	2	7.53	YES
2D	1.4	0.2	35.7	2	3.49	YES
2E	2.3	9.9	36.5	2	5.66	NO
2F	4.5	6.1	44.9	2	11.25	NO
3A	3.7	8.3	5.0	3	9.36	NO
3B	1.5	0.6	25.2	3	3.72	YES
3C	2.9	9.6	23.2	3	7.15	NO
3D	4.0	3.7	32.9	3	9.88	YES
3E	4.6	1.7	10.4	3	11.56	YES
3F	2.0	6.3	38.6	3	4.92	NO
4A	1.7	6.5	6.6	4.	4.15	NO
4B	3.4	1.7	18.7	4	8.60	YES
4C	4.9	9.7	25.1	4	12.3	NO
4D	2.1	4.4	27.0	4	5.37	NO
4E	3.0	7.0	14.9	4	7.52	NO
5A	3.3	8.2	4.8	5	8.30	NO
5B	2.2	0.8	6.9	5	5.43	YES
5C	1.5	5.4	19.8	5	3.84	NO
5D	3.2	9.4	22.6	5	7.91	NO
6A	1.8	8.3	5.4	6	4.50	NO
6B	3.7	9.7	26.5	6	9.21	NO
6C	4.9	6.0	12.2	6	12.20	NO
6D	4.2	2.0	18.4	6	10.61	YES
6E	3.0	3.5	21.9	6	7.54	YES

B. Target Output: Residual Strength from Numerical Fracture Simulations

For each input damage configuration, a numerical fracture simulation is employed to determine residual strength, which provides target output used to train and test the NN. FRANC3D\NG [42] is used to insert each parameterized star-shaped crack configuration into the solid FE region of each panel. An ABAQUS® contact algorithm is employed to prevent crack surfaces from overlapping during the applied loading. Contact properties are defined as frictionless in the tangential direction with "hard" pressure-overclosure behavior normal to the contacting crack surfaces, which minimizes

interpenetration. The FE models are then analyzed using ABAQUS®, and FE analysis results are post-processed to determine residual strength.

For the sake of illustrating the NN methodology, two simplifying assumptions are made here to predict residual strength of the ISPs. First, the ISPs remain linear-elastic and can be analyzed using LEFM parameters (SIFs). Second, the residual strength can be predicted for a static crack configuration (i.e. crack growth is not modeled in this example).

In the ISP NN example, the LEFM approximation of residual strength is based on mixed-mode I/II fracture criteria [16, 18] to account for local mode mixity (in-plane) of angled cracks in the star-shaped damage array. In [43], Broek describes a practical mixed-mode I/II failure envelope, approximated by the equation of an ellipse:

$$(K_I/K_{Ic})^2 + (K_{II}/K_{IIc})^2 = 1. \tag{1}$$

For the AA $2XXX$ series material in the ISP example, $K_{Ic} = 32$ MPa$\sqrt{m}$, and K_{IIc} is assumed to be 10% less than K_{Ic} after results from the strain energy density criterion presented by Sih [18].

Using this mixed-mode LEFM-based approximation, residual strength is defined here as the applied traction load, Fig. 2, that first causes unstable crack growth for any point along any crack front of the star-shaped damage configuration. In other words, as soon as one point along one crack front reaches a critical combination $(K_I, K_{II})_c$ on the elliptical failure envelope, the entire panel is assumed to fail. The method for determining the residual strength for each damaged panel is shown in Fig. 4 and proceeds as follows: (1) analyze the ISP FE model with P; (2) compute K_I and K_{II} at each node along each crack front using FRANC3D\NG; (3) *for each crack front node*, find the intersection point $(K_I, K_{II})_c$ of the elliptical failure envelope with a straight line from the origin to the computed (K_I, K_{II}) and subsequently find the linear scaling factor, λ, that maps (K_I, K_{II}) to $(K_I, K_{II})_c$; (4) of all the computed scaling factors, select the most critical, λ_c; (5) calculate P_{max} as P scaled by λ_c.

To ensure that nonlinearity due to crack face contact does not invalidate the linear load scaling approach described above, each of the damaged ISPs is reanalyzed with the respective scaled load,

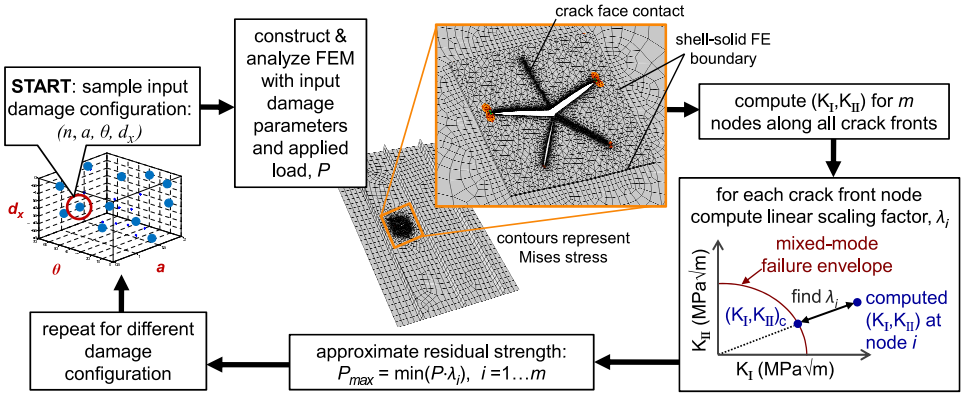


Fig. 4 LEFM-based procedure for approximating residual strength of the damaged ISPs in the NN example.

i.e. the approximated residual strength. In all cases, $(K_I, K_{II}) = (K_I, K_{II})_c$ at the predicted crack front failure point, indicating that the scaled loads indeed correspond to failure loads according to the LEFM-based failure criterion assumed for this example problem. Values of P_{max} provide the target outputs used to train the NN.

C. Neural Network Construction

The inputs (sampled damage parameters) and target outputs (residual strength predictions from numerical fracture simulations) are used to train and test a NN. For the ISP example, a feedforward NN with a backpropagation learning rule [25, 44], which is a commonly used type of supervised NN, is constructed using MATLAB® [41]. The NN consists of a single hidden layer mapping the four-parameter input vectors (n, θ, a, d_x) to the single-valued outputs (P_{max}). The reader is referred to [44] for a general discussion on NNs and details regarding specific implementation of the transfer functions and training algorithm described next. A tan-sigmoid transfer function is employed to map the weighted inputs plus bias to the interval (-1,1). A linear transfer function proportionally maps the weighted output plus bias from the hidden layer to the output layer. Data presented to the NN is divided into three sets—training, validation, and test. Weights and biases of the NN are adjusted at each iteration, or epoch, using the training set and a Levenberg-Marquardt optimization algorithm, as described in [45]. The algorithm seeks to improve performance of the NN by minimizing error between the NN outputs and the target outputs. Weights and biases from

training at any epoch are then used to check performance of the NN using the validation and test sets. The validation set prevents overtraining of the NN by ceasing training if performance degrades over a certain number of successive epochs. The test set is not used for training but is used to test NN accuracy following the current training epoch. The NN performance metric used here is the MSE, calculated as:

$$MSE^{(i)} = \frac{1}{Q^{(i)}} \sum_{k=1}^{Q^{(i)}} (t_k^{(i)} - p_k)^2, \tag{2}$$

where the superscript *(i)* corresponds to the training, validation, or test set, Q is the number of data points in the respective set, t_k is target output for the k^{th} input, and p_k is output predicted by the NN for the same k^{th} input.

The NN can be optimized by adjusting any number of parameters, including transfer functions between layers, number of hidden layers, various performance metrics, and so forth. In the ISP example, the NN is optimized by varying the number of neurons in the hidden layer (4,5,6) and by increasing size of the training set from 60%, to 70%, to 80% of the available data (with the balance equally divided between validation and test sets). Further, the performance metrics are optimized by minimizing MSE for the training and testing sets and by specifying that the correlation coefficient between NN output and targets should be at least 0.95 over the entire data set.

D. Parametric Sensitivity Studies

The trained NN is then employed to conduct parametric sensitivity studies, whereby sensitivity of residual strength to each postulated damage parameter is gauged. The sensitivity studies are carried out for configuration 4E, Table 2, as it represents an average damage configuration in terms of n, a, and d_x as compared to the other configurations.

A sensitivity study is carried out for each of the four damage parameters. In each study, three damage parameters of configuration 4E are held constant while one is varied. The variable parameter in each study takes values in the range of the corresponding variable on which the NN was trained. For example, the longest a considered in the sensitivity study is no longer than the longest a used to train the NN, which is $a = 4.9$ cm in the ISP example (damage configurations

4C and 6C). Further, the variable parameter takes values that are equally incremented within the respective range. Results from the study are presented in the following subsection.

E. Results and Discussion from Neural Network Example

Table 3 shows the approximated residual strengths of all damaged ISPs based on numerical fracture analyses and LEFM assumptions outlined in II B. The table is sorted in order of increasing residual strength, and the corresponding damage parameters are provided to help draw preliminary conclusions. One immediate observation is that panels with severed stiffeners have lower ($\approx 50 - 80$ MPa) residual strengths, as expected. The single exception is damage configuration 2B. Though it severs the stiffener, configuration 2B is less critical than all other stiffener-severing cases and some intact-stiffener cases because it is an $n = 2$ configuration (straight crack) aligned with the loading direction. Overall, the correlation between severed stiffener and reduced residual strength highlights the effect of the load carrying stiffener on crack criticality.

The ISPs with the lowest computed residual strength (configuration 3D) and highest computed residual strength (configuration 5C) are presented in Fig. 5. Each ISP is depicted with its respective residual strength, or failure load, applied. The predicted point of first-failure lies along the crack front indicated. Configuration 5C has *more* cracks and is 62.5% smaller than configuration 3D, though it is not the smallest of all configurations. More importantly, configuration 5C leaves the stiffener intact while configuration 3D results in a severed stiffener. For configuration 3D, the crack front that lies within the severed region and near the geometric discontinuity of the stiffener junction is subjected to higher stresses and is predicted to be critical.

The optimal NN consists of four neurons in a single hidden layer with 80% of available data (i.e. twenty damage configurations) allocated to training. NN performance metric (MSE) as a function of training epochs is plotted in Fig. 6 for training, validation, and test sets. The NN is best trained at epoch 141, beyond which the MSE in the validation set continually increases and overtraining is said to occur. At this epoch, MSE of the three sets are $MSE^{train} = 0.001$, $MSE^{val} = 0.87$, and $MSE^{test} = 0.30$. Weights and biases connecting the input layer (damage parameters) to the hidden layer and the hidden layer to the output (residual strength) at training epoch 141 are presented in

Table 3 LEFM-based residual strength approximations for all damage configurations considered in the ISP example, sorted by increasing residual strength.

Damage configuration ID	a (cm)	d_x (cm)	Severs stiffener?	P_{max} (MPa)
3D	4.0	3.7	YES	54.2
6D	4.2	2.0	YES	56.8
3E	4.6	1.7	YES	56.8
4B	3.4	1.7	YES	58.1
6E	3.0	3.5	YES	63.5
2C	3.0	3.6	YES	67.6
5B	2.2	0.8	YES	71.2
3B	1.5	0.6	YES	73.8
2D	1.4	0.2	YES	80.7
6C	4.9	6.0	NO	82.7
4C	4.9	9.7	NO	83.9
2A	3.8	7.1	NO	89.1
3A	3.7	8.3	NO	90.3
2B	4.2	2.1	YES	93.8
5A	3.3	8.2	NO	94.3
4E	3.0	7.0	NO	98.4
5D	3.2	9.4	NO	101.5
2F	4.5	6.1	NO	102.1
6B	3.7	9.7	NO	109.2
3C	2.9	9.6	NO	110.2
4A	1.7	6.5	NO	124.9
3F	2.0	6.3	NO	128.2
2E	2.3	9.9	NO	129.5
6A	1.8	8.3	NO	130.7
4D	2.1	4.4	NO	132.8
5C	1.5	5.4	NO	146.7

Tables 4 and 5.

Considering the entire set of damage configurations, the NN predictions correlate well with the target outputs at epoch 141, as depicted in Fig. 6. Despite the good overall correlation and small MSE for the training set, the MSE in the validation and testing sets (which include only three damage configurations each) may be too large for actual implementation onboard an aircraft, depending on design specifications. It is suspected that adding more samples to the entire set of damaged ISPs would further reduce these errors in the NN.

The influence of each damage parameter on predicted residual strength can be visualized graphically by plotting predicted residual strength as a function of each damage parameter (see Fig. 7). Sensitivity can be quantified by a number of different metrics, many of which yield comparable

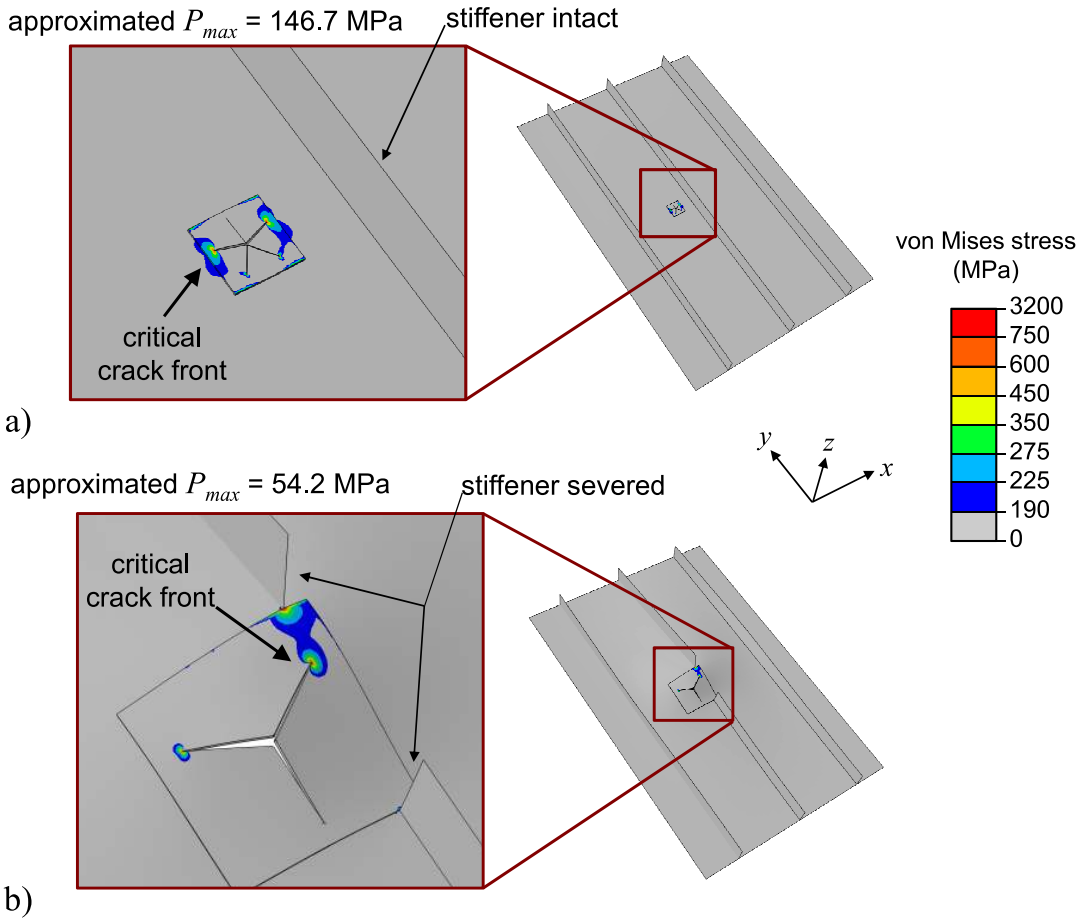


Fig. 5 Two different damaged panels (ID 5C and ID 3D) shown with respective P_{max} applied. Panels represent damage configurations that are least critical (a) and most critical (b) of all configurations considered. Predicted failure point lies along the indicated crack front. Deformation is scaled by factor of 10. FE mesh is not shown for better contour visualization.

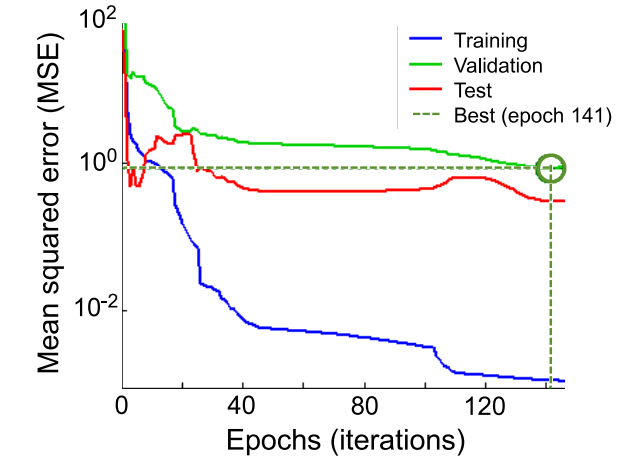
Table 4 NN weights and biases used to map input layer to hidden layer for the optimized NN at training epoch 141.

	Input parameter	Hidden layer neuron 1	2	3	4
Weights	n	-0.43	-2.98	-1.11	1.83
	a	0.69	-9.17	1.61	-2.27
	x	3.58	-0.15	1.50	-8.74
	θ	-1.53	2.87	-2.96	5.58
Biases		-0.42	0.45	1.37	2.94

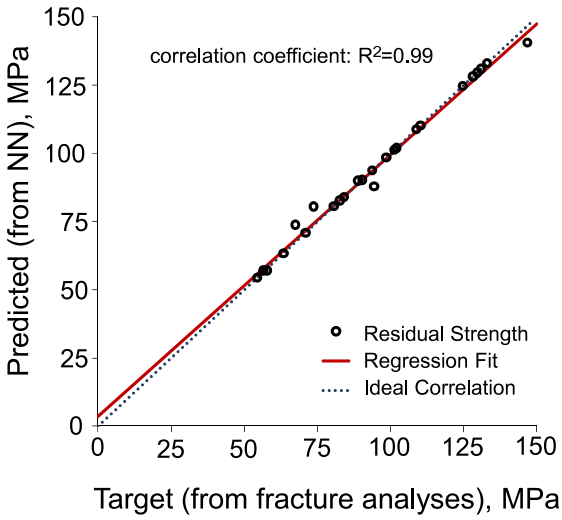
results [46]. Here, sensitivity is quantified by the c_v, expressed as a percentage:

$$c_v = \frac{\sigma}{\bar{P}_{max}}, \text{ where } \sigma = \sqrt{\frac{\sum_{i=1}^{N}(P_{max,i} - \bar{P}_{max})^2}{N-1}}. \tag{3}$$

For any given sensitivity subset, i corresponds to the i^{th} sample configuration, $\bar{P}_{max}$ is the average



a)



b)

Fig. 6 (a) NN performance as a function of training epochs for the optimized NN; overtraining occurs after epoch 141. (b) Correlation between predicted and target residual strength values considering all damage configurations.

Table 5 NN weights and bias used to map hidden layer to output for the optimized NN at training epoch 141.

	Hidden layer neuron	Output
Weights	1	0.78
	2	0.25
	3	-1.70
	4	1.05
Bias		1.29

residual strength of the subset, and N is the total number of damage configurations in the respective

subset. Sensitivities to each damage parameter are calculated as $c_v^{(x)} = 24.8\%$, $c_v^{(a)} = 16.6\%$, $c_v^{(n)} = 6.0\%$, and $c_v^{(\theta)} = 1.2\%$.

Orientation and number of cracks are found to have relatively minor influences on predicted residual strength, which is apparent both by their sensitivity metrics and by the plots (b) and (d) of Fig. 7. Crack length, on the other hand, has a more significant influence and causes a reduction in predicted residual strength as crack length increases, which is expected. What is unexpected, however, is the step-like behavior depicted in Fig. 7(c). This behavior is caused by binary modeling of the stiffener (explicitly modeling as severed or intact), a feature that is inherently implicit in both crack size and location. The stiffener effect is also apparent in Fig. 7(a) of damage location sensitivity. Predicted residual strength is lowest (and relatively insensitive to damage location) if the damage is located such that it severs the stiffener. As the damage location moves away from the stiffener and is no longer severing it, there is a linear increase in residual strength until the damage is located within the middle quarter of the bay.

In general, the importance of this kind of sensitivity study is (1) to gain a better intuition of how and why certain damage characteristics influence residual strength and (2) to potentially decrease the dimensionality of the NN by neglecting parameters deemed insignificant.

III. 3D Elastic-plastic Fracture Simulations for Improved Neural Network Training

For discrete-source damage cases involving significant ductile tearing, a generally applicable 3D EPFM framework may be used for improving residual strength training data. An elastic-plastic crack growth simulation procedure, as implemented in this section, is illustrated in Fig. 8 and proceeds as follows:

1. define an uncracked FE model and boundary conditions;
2. extract a sub-region of the mesh for crack insertion, remeshing, and reconnection with the global mesh;
3. map previous deformation and material state onto the remeshed model;
4. perform nonlinear FE analysis and monitor the crack growth criterion;

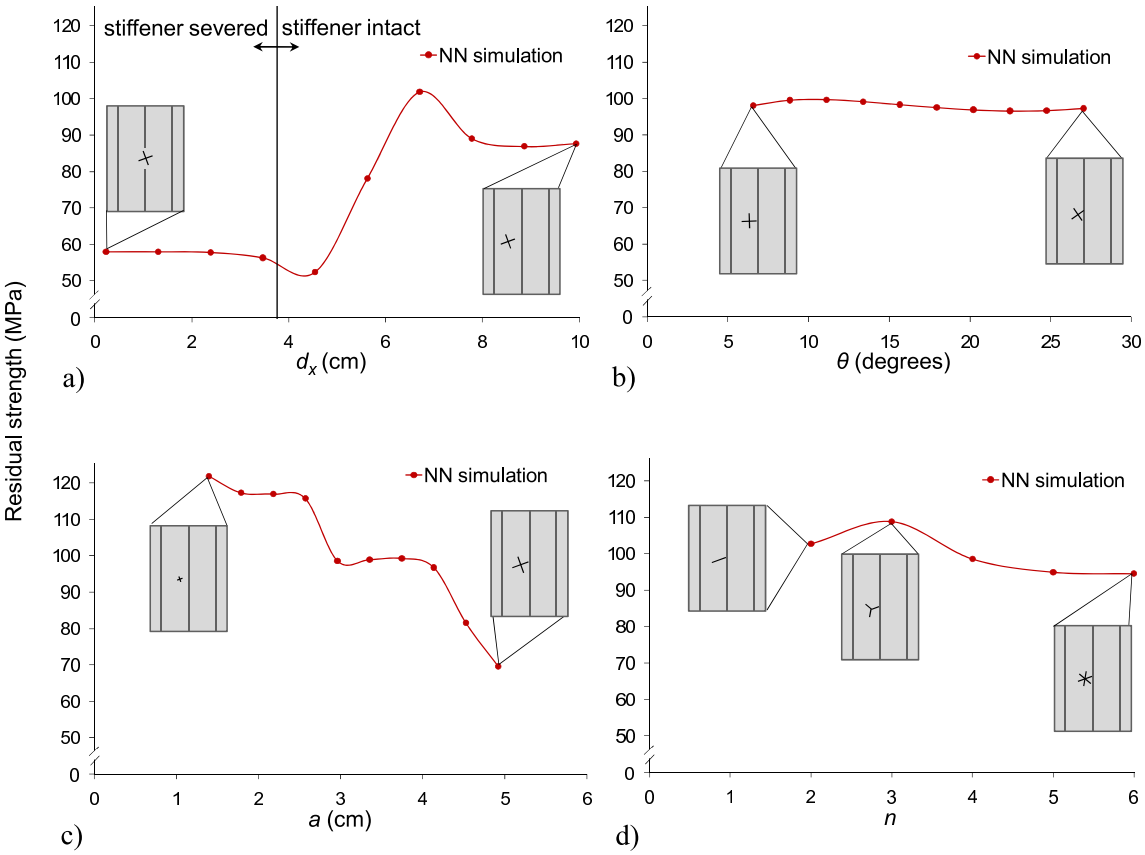


Fig. 7 Sensitivity of predicted residual strength to each damage parameter.

5. once criterion is satisfied, stop the current FE analysis to update crack configuration, remesh sub-region, and reconnect sub-region mesh with global mesh;

6. repeat from step 3 until critical crack length is achieved or until residual strength is attained.

The simulation procedure allows for prediction of curvilinear crack paths and arbitrary crack front evolution. The EPFM framework was overviewed in [47] and is described here for completeness. Scripts used for implementation are found in the Appendix.

A. Nonlinear Fracture Parameter: Crack-tip Displacement

In elastic-plastic tearing simulations, especially of thin metallic structures, crack growth should be characterized by an appropriate nonlinear parameter. One such parameter arises from correlation between crack growth and a critical amount of opening or displacement behind the crack tip (see [48] for details). A criterion based on this parameter, which is called the crack-tip displacement

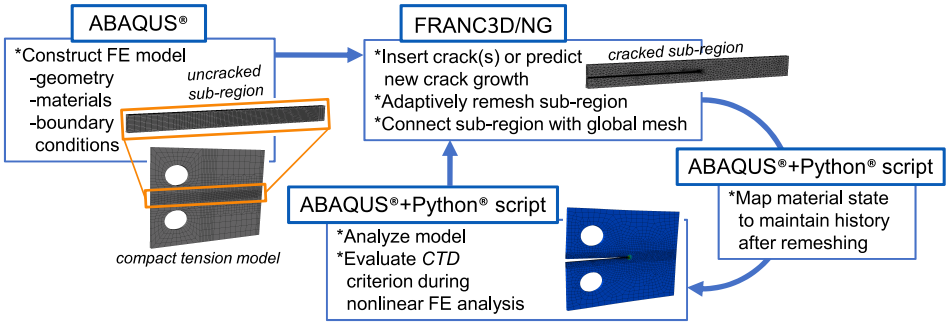


Fig. 8 Elastic-plastic crack growth simulation algorithm using FRANC3D\NG. Contributions from this work include evaluation of crack-tip displacement (CTD) criterion during nonlinear FE analysis and implementation of material state mapping algorithm following adaptive remeshing.

(CTD) or sometimes referred to as the generalized crack (tip) opening displacement [22, 49], is implemented here. Notably in simulation, once a critical value, CTD_{crit} , has been determined for a specific material and thickness through a calibration procedure, the same CTD_{crit} is applicable over a range of structural configurations comprising the same material and thickness under similar loading. In the EPFM simulations here, CTD is computed as:

$$CTD = \sqrt{{CTD_I}^2 + {CTD_{II}}^2 + {CTD_{III}}^2} \tag{4a}$$

$$CTD_I = v_1 - v_2 \tag{4b}$$

$$CTD_{II} = u_1 - u_2 \tag{4c}$$

$$CTD_{III} = w_1 - w_2, \tag{4d}$$

where u, v, and w correspond to displacements in the x, y, and z directions, respectively, and subscripts 1 and 2 denote the two points used to compute CTD. CTD is computed between two points that are initially coincident (one on each crack face) on the undeformed crack surface at a distance, d, behind a crack front node (i.e. in the direction normal to the crack front at the particular

crack front node). This is illustrated schematically in 2D in Fig. 9. In 3D, CTD values are computed behind multiple crack front nodes. The pair of initially coincident points where CTD is calculated is called a CTD point. Element shape functions are used to interpolate displacements (u, v, w) such that the CTD points need not correspond to nodal locations.

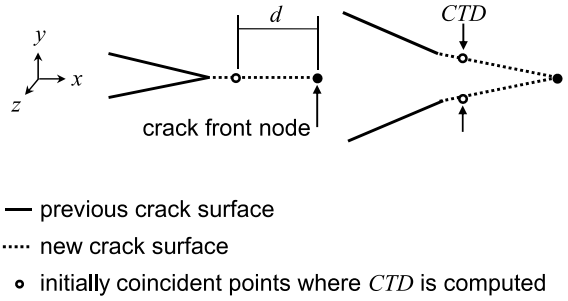


Fig. 9 Simplified schematic of CTD implementation illustrated on crack profile.

Crack growth occurs when CTD attains a critical value, CTD_{crit}, within a specified tolerance. There are several ways to evaluate the CTD criterion when modeling a 3D crack front, including evaluation at a single CTD point either midway along the crack front or on the specimen's free surface. Alternatively, the CTD criterion may be evaluated by comparing CTD_{crit} to an average CTD value calculated for multiple CTD points. Because CTD_{crit} is known to depend on 3D constraint at any point along a crack front, using a single CTD_{crit} to predict the advance of an entire crack front might not be valid for all cases. A more rigorous and computationally expensive evaluation technique would be to compare CTD at each CTD point to a constraint-dependent CTD_{crit}. While some work has been done to resolve a relationship between 3D constraint and CTD_{crit} [20, 50], a constraint-dependent fracture criterion is not evaluated in the simulations described in this work.

B. Material State Mapping Algorithm

Following crack growth and remeshing, state variables are mapped from the previous mesh to the current mesh using an inverse isoparametric mapping routine, as in [51]. Here, the scripts (n) and $(n+1)$ generically denote previous and current increments of crack growth, respectively. Lim *et al.* described the inverse isoparametric mapping technique for 2D elastic-plastic fracture simulations [52]. Implementation of the mapping algorithm consists of two high-level steps: (1) in the (n) mesh,

state variables stored at integration points are extrapolated to nodes using element shape functions and (2) displacements and state variables are transferred to either nodes or integration points in the $(n+1)$ mesh. The second step involves finding, for each point in the $(n+1)$ mesh, the natural coordinates (ξ, η) of that point with respect to the element from the undeformed (n) mesh in which that point would spatially reside. The inverse problem becomes finding the natural coordinates (ξ, η) that satisfy the known global coordinates:

$$X^{(n+1)} = \Sigma N_i(\xi, \eta) X_i^{(n)}, \tag{5}$$

where the subscript i ranges from one to the number of element nodes, $X^{(n)}$ are global nodal coordinates in the (n) mesh, $X^{(n+1)}$ are point or nodal coordinates in the $(n+1)$ mesh, and N are element shape functions evaluated at (ξ, η). Once (ξ, η) are known, nodal displacements and state variables, U, can be transferred from the (n) mesh to the $(n+1)$ mesh in a forward manner, again using the element shape functions, N:

$$U^{(n+1)} = \Sigma N_i(\xi, \eta) U_i^{(n)}. \tag{6}$$

Two levels of mapping are incorporated into the extended FRANC3D\NG and ABAQUS® software framework. First, displacements are mapped onto the undeformed mesh following crack growth and remeshing. Second, a mapping function available in ABAQUS® is invoked to map the remaining state variables (e.g. stress, strain, plastic strain) onto the deformed mesh. When mapping material state between successive cracked configurations, it is critical that mesh refinement in regions of high gradients (e.g. near crack fronts) is sufficient to minimize solution diffusion, which occurs as a result of extrapolation, interpolation, and nodal averaging (if employed). This effect can become compounded as the crack growth simulation continues. After growing the crack and remeshing, the updated mesh model contains additional surface area due to crack extension, and equilibrium must be re-established before additional load is applied. During the equilibration procedure, global boundaries are held fixed and the new, traction-free crack surfaces are allowed to displace in response to surrounding fields, as shown in Fig. 10.

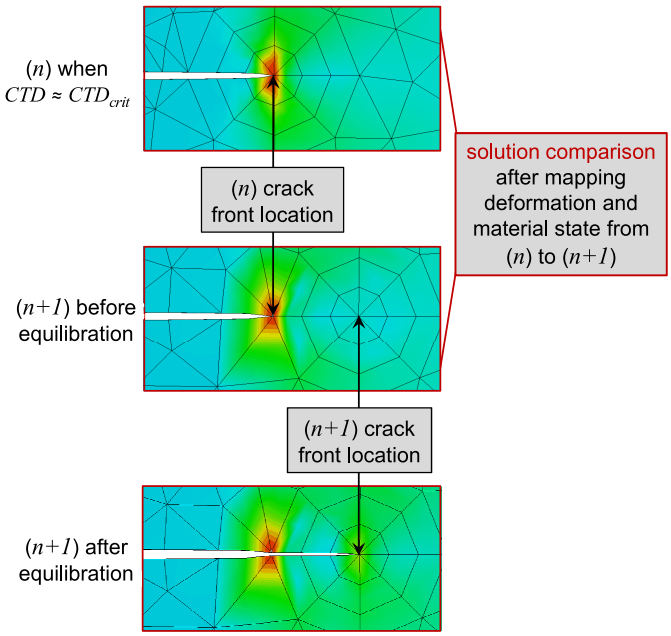


Fig. 10 Qualitative comparison of deformation and equivalent plastic strain field after mapping and subsequent equilibration. Images show face of 3D mesh. Deformation is not scaled.

C. Validation Simulations

Two stable tearing experiments are simulated to validate the EPFM framework for predicting crack growth and residual strength of relatively thin metallic structures. To illustrate the necessity of using an EPFM framework for predicting crack propagation and residual strength in such cases, the experiments are also simulated using an LEFM-based methodology. In the LEFM simulations, the material is modeled as linear-elastic, and crack growth is assumed to occur when an average value of K_I (and K_{II} for mixed-mode crack growth) along a crack front approximately equals fracture toughness of the material for any increment of crack length.

1. Arcan Specimen

A (modified) mixed-mode I/II Arcan fracture test [53] is simulated. Experimental details and results were provided in [54] and [55]. Drawings of the fracture specimen and modified load fixture are shown in Fig. 11. Curvilinear crack growth was induced in an AA 2024-T3 fracture specimen by applying monotonic load at a 30° angle relative to the fatigue precrack plane, as depicted in Fig. 11.

The FE model of the load fixture and fracture specimen is constructed using ABAQUS® and is

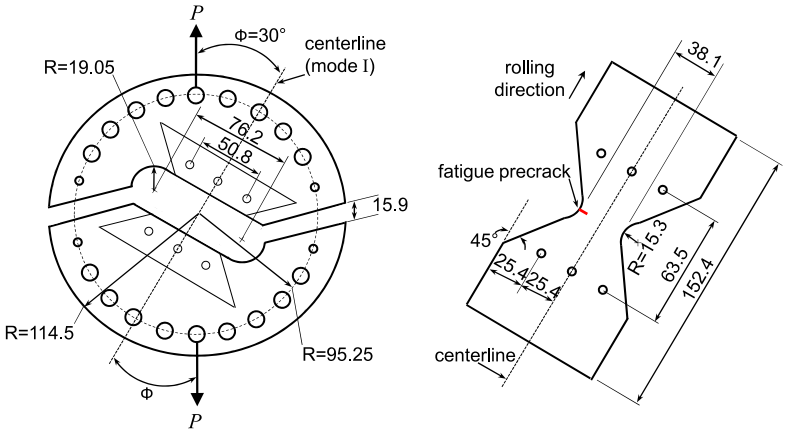


Fig. 11 Schematic of Arcan test fixture (left) and 2.3 mm thick fracture specimen (right). Load, P, can be applied at different pinholes in the fixture to induce mode I/II crack growth in the specimen. Drawings not to scale. All length dimensions in mm.

depicted in Fig. 12. The FE model contains an initial crack of length 6.35 mm and does not simulate the fatigue precracking process. A mesh refinement study is first conducted by simulating a pure mode I Arcan test (0° loading angle). Results from the study reveal that, for the initial crack, the applied load at CTD_{crit} varies by less than 0.25% when the element length nearest the crack front is 0.33 mm or smaller. This converged level of mesh refinement is used for the mixed-mode Arcan model presented here (30° loading angle). The specimen and fixture are assumed to be perfectly bonded such that coincident nodes are merged at the specimen-fixture interface. The fixture is modeled using 6000 $C3D10$ elements and remains unchanged throughout the simulation. The fracture specimen sub-region is subject to geometry and mesh updating within FRANC3D\NG. Depending on the crack length, the specimen comprises between 9000 and 25,000 quadratic elements, which include a standard rosette of $C3D15$, $C3D20$, and pyramid (collapsed $C3D20$) elements surrounding the crack front (see [42] for details). The bulk of the sub-region mesh comprises $C3D10$ elements.

Material properties for the 15-5PH stainless steel fixture are $E = 207$ GPa and $\nu = 0.3$. The fixture is assumed to remain elastic during loading. Material properties for the AA 2024-T3 fracture specimen are $E = 71.4$ GPa, $\nu = 0.33$, and $\sigma_y = 345$ MPa. The strain hardening curve for the specimen is provided in Fig. 13. A von Mises yield criterion with isotropic hardening is assumed. For the LEFM simulation, the specimen is modeled as linear-elastic with $K_{Ic} = 37$ MPa$\sqrt{m}$ for AA 2024-T3 in the LT orientation [56].

Crack growth occurs in the LEFM simulation when the average (K_I, K_{II}) along the crack front

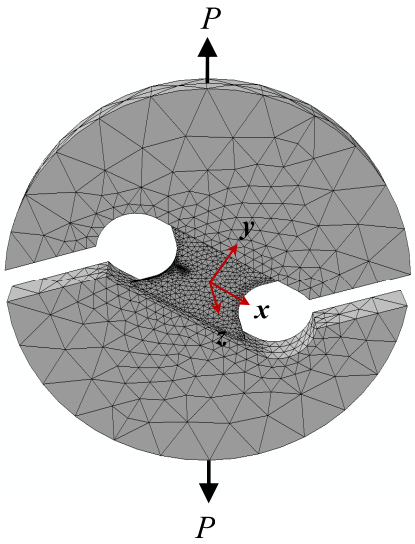


Fig. 12 3D FE model of modified Arcan test set-up, including load fixture and fracture specimen. Line traction, P, is applied at 30° from mode I axis (y-axis).

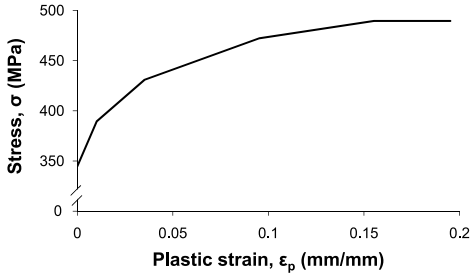


Fig. 13 Strain hardening curve determined from uniaxial tension tests for AA 2024-T3 in LT orientation. Courtesy NASA Langley Research Center. Similar curves were used in [22] and [23].

reaches a critical combination. The critical combination is determined using the mixed-mode failure envelope described in IIB, where K_{IIc} is taken to be 10% less than K_{Ic}.

In the EPFM simulation, important observations from a number of studies inform the selection of CTD_{crit} and d used to predict crack growth. First, from [6, 54, 55], for 2.3 mm thick AA 2024-T3 specimens precracked in the L-T orientation, an average constant critical value of CTD_{crit}= 0.1 mm was observed, where CTD was measured on the specimen face at $d = 1$ mm. This critical value was observed for a range of specimens exhibiting mode I dominant crack growth, which includes the 30° Arcan test [54, 55]. Scatter among the measurements was typically ±0.02 mm. Second, significant crack front tunneling was observed in the fracture specimens, especially during initial crack extension [6]. Third, a recent study by Lan *et al.* suggested that modeling tunneled cracks using straight crack

fronts can lead to over-estimation of load versus crack extension predictions [50]. This is because higher levels of constraint along a crack front, which can induce crack tunneling, effectively decrease fracture toughness. Lan *et al.* noted that surface-measured CTD values can be 24% larger for cases with crack tunneling than cases without. In the Arcan simulation, crack front tunneling and slant crack growth are not modeled. Thus, using the surface-measured value of CTD_{crit}= 0.1 mm from [6, 54, 55] as the crack growth criterion in the Arcan simulation, an over-prediction of load versus crack extension is expected. Assuming that the over-prediction is proportional to the difference in surface-measured CTD_{crit} values between straight and tunneled crack fronts, CTD_{crit} is taken to be 0.08 mm, which is $\approx$ 24% less than the observed average surface-measured value of CTD_{crit}= 0.1 mm and also corresponds to the lower bound of experimental scatter. This adjusted critical value used in simulation should account for some expected over-prediction of residual strength.

Simulated crack growth occurs in increments of approximately 1 mm, which satisfies convergence results from similar simulations [22]. Propagation direction is predicted according to the maximum tangential stress theory [16], though a 2D CTD-based directional criterion [19] could also be used.

Simulation proceeds by applying displacement in the direction indicated in Fig. 12, which corresponds to a 30° loading angle. In the EPFM simulation, after each increment of, inclusively, crack growth, remeshing, and material state mapping (see subsection III B), all nodes with applied boundary conditions are held fixed while the model is brought into equilibrium. Then, additional displacement is applied, and the simulation continues until maximum load, or residual strength, is attained.

Applied load versus crack extension is plotted in Fig. 14 from the Arcan experiment and simulation. Both LEFM and EPFM simulation results are plotted. Compared to experiment, residual strength is 6.8% higher using the EPFM framework and 10.8% lower using the LEFM method. In the LEFM simulation, maximum load occurs at the first increment of crack growth; whereas, maximum load occurs in the EPFM simulation after a small amount of crack growth, consistent with experiment. The load to initiate crack extension is approximately the same in both EPFM and LEFM simulations since there are no residual stresses in the model at da=0. However, unlike the LEFM simulation, the EPFM simulation is able to capture the shape of the load versus crack exten-

sion curve. In other words, including plasticity effects in the model requires an increase in applied load, initially, to drive crack extension before reaching the residual strength of the specimen.

Although the EPFM framework captures the shape of the load versus crack extension curve, it slightly over-predicts the maximum load. The over-prediction is likely related to modeling the crack front, which tunnels or thumbnails in reality, as remaining straight throughout the simulation. Given a constraint-dependent CTD_{crit}, curvature along the crack front could be predicted using a point-by-point evaluation technique, in which case it is suspected that the load versus crack extension curve would be predicted even more accurately. Another potential contribution to the discrepancy is the numerical tolerance for satisfying CTD_{crit}, which in this case is set at 2%.

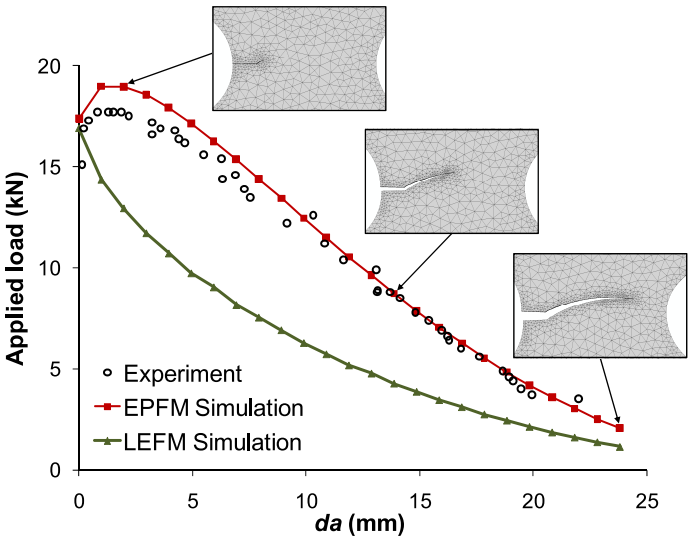


Fig. 14 Applied load versus crack extension, *da*, from the 30° Arcan experiment and simulations. Insets show snapshots of deformed mesh at various crack increments. Complete simulation can be viewed at *www.cfg.cornell.edu*

Figure 15 shows the simulated and experimental curvilinear crack trajectory as viewed from the specimen free surface. There is a slight deviation in the actual crack trajectory during the first 6.35 mm of crack length due to fatigue precracking processes, which are not modeled in the simulation. Rather, the fatigue precrack is simply modeled as a perfectly planar initial crack in the simulation. The deviation appears to have little effect on the predicted crack path thereafter.

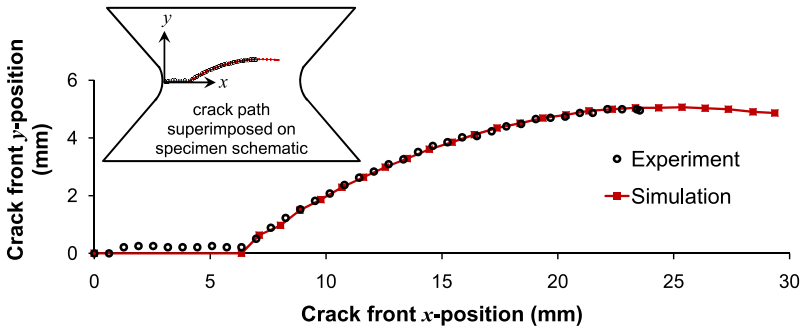


Fig. 15 Comparison of experimental and simulated curvilinear crack paths in the 30° Arcan fracture test. Inset shows reference coordinate system.

2. Integrally-stiffened Panel

A stable tearing test of an ISP machined from a lower wing-skin aluminum alloy, C433-T39, is also simulated. The test was conducted at Alcoa Technical Center. Test details, data, and results were overviewed in [57] and have also been provided to the authors by Alcoa Technical Center. Relevant details are described next, and additional details from the test program can be found in the Appendix.

Dimensions of the panel are shown in Fig. 16(a). An initial two-bay saw cut of length $\approx$ 2.54 cm was made at mid-height to completely sever the middle stiffener. The initial cut was then propagated under fatigue loading until both crack fronts were 2.54 cm short of reaching the intact stiffeners ($2a \approx$ 24.1 cm). The panel was then loaded monotonically in uniaxial tension until failure occurred by unstable crack growth. Crack front branching was observed, where an initial crack propagating toward an intact stiffener eventually split into two distinct cracks, one continuing into the adjacent bay and one propagating in the z direction within the stiffener. Photographs of the test panel with views of crack branching are provided in Fig. 17.

A 3D FE model of the entire panel is constructed using ABAQUS® [1]. The FE model contains an initial crack of total length 24.1 cm, which corresponds to the fatigue crack length just prior to conducting the residual strength test. The FE model with initial crack is shown in Fig. 16(b). The mesh region that remains unchanged throughout the tearing simulation is modeled using 56 $C3D15$ and 9,400 $C3D20R$ elements. A 38.5 x 12.7 cm^2 sub-region centered in the panel is subject to geometry and mesh updating within FRANC3D\NG. Depending on crack length, the sub-region

comprises between 27,000 and 95,000 quadratic elements, including a bulk of $C3D10$ elements and a standard rosette of $C3D15$, $C3D20$, and pyramid (collapsed $C3D20$) elements surrounding the crack front (see [42] for details). The mesh interface between the sub- and global regions is coherent, obviating the enforcement of a constraint.

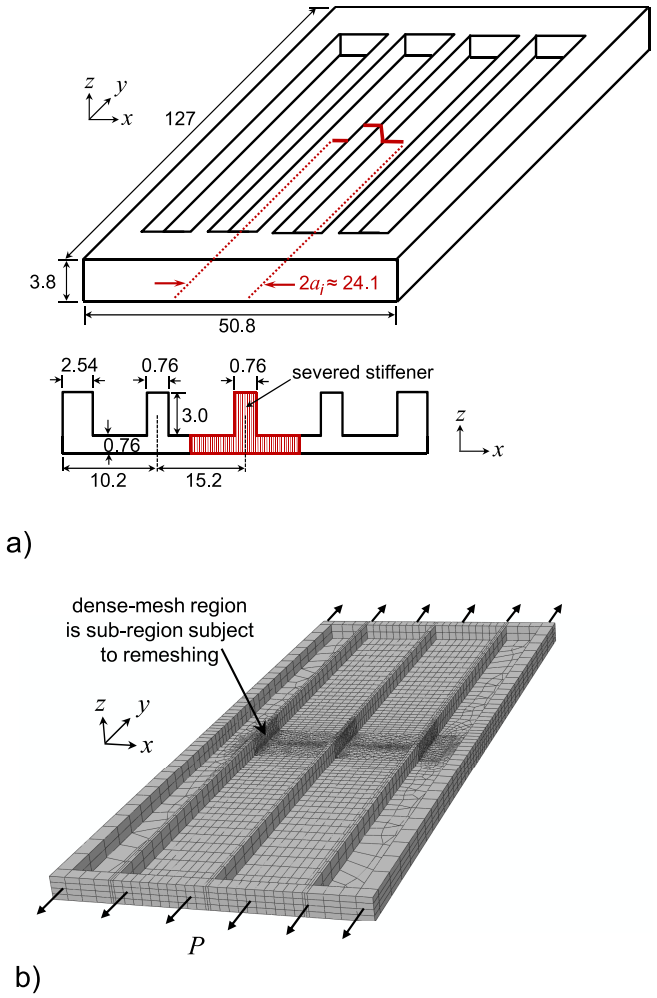


Fig. 16 (a) Schematic (not to scale) from [57] showing dimensions of symmetric ISP tested at Alcoa Technical Center. Isoparametric view of full panel indicates final fatigue crack length $2a_i$, and cross-section view in plane of the crack shows where stiffener is completely severed. All dimensions are in cm. (b) Corresponding 3D FE model of ISP. Initial crack severs middle stiffener. Traction, P, is applied uniaxially in the y direction.

Mesh refinement near each crack front is dictated, to some extent, by crack front location with respect to stiffener. As the crack initially propagates through the skin, element lengths nearest both crack fronts are 0.5 mm. When the crack fronts are within 1.5 mm of the 90° skin-stiffener junctions, however, near-crack front elements must decrease in size. This is because the rosette

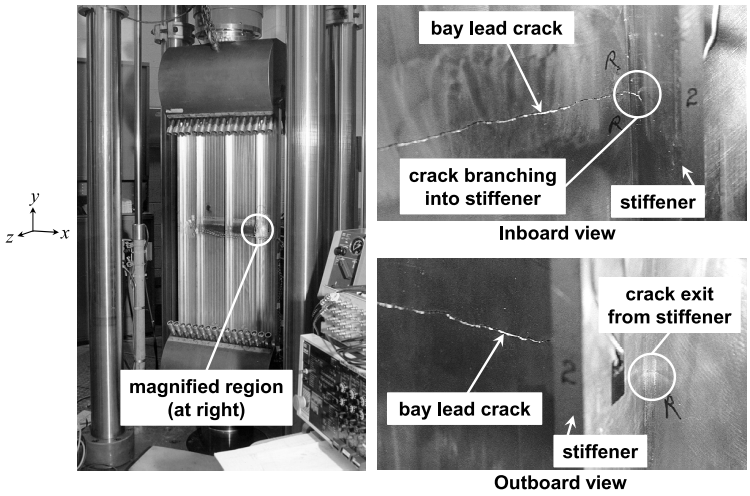


Fig. 17 Photographs of ISP with central two-bay crack from the Alcoa test program [57]. Full panel in load frame (left) and angled views of crack front branching into stiffener (top right) then exiting the stiffener (bottom right).

template of elements surrounding either crack front consists of three rings of equi-length elements. In order to accommodate (i.e. facilitate remeshing) a full rosette of elements within the proximity of the discontinuous geometry, the size of the template elements must decrease.

The thickened grip ends of the panel are modeled as linear-elastic with an elastic modulus approximately five times greater than that of C433-T39. The rest of the panel is assigned C433-T39 material properties: $E = 71.4$ GPa, $\nu = 0.3$, and $\sigma_y = 455$ MPa [58]. The strain hardening curve used for C433-T39 is provided in Fig. 18. A von Mises yield criterion with isotropic hardening is assumed. For the LEFM simulation, the panel is modeled as linear-elastic with $K_{Ic} = 50$ MPa$\sqrt{m}$ for C433-T39 [58].

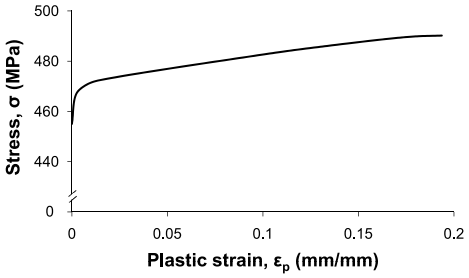


Fig. 18 Strain hardening curve determined from uniaxial tension tests for C433-T39 in LT orientation [57].

Boundary conditions are applied to simulate actual loading in the panel. Nodes on the bottom face of the lower grip end are fixed in the y direction. Displacement is applied in the y direction at

nodes on the top face of the upper grip end. Additionally, nodes along the same top and bottom grip end faces are fixed in the x and z directions. For the EPFM simulation, after each increment of, inclusively, crack growth, remeshing, and material state mapping (see subsection III B), all nodes with applied boundary conditions are held fixed while the model is brought into equilibrium before applying additional displacement. Additionally, based on preliminary simulation results, the entire back (z_{min}) face is artificially fixed after mapping and during the equilibration procedure. This is because resonance in the z direction is observed with increased crack growth otherwise. The resonance occurs when mapped tensile and compressive stresses in the faces of the panel are artificially reversed during equilibration of the mapped solution. The additional boundary condition is, however, removed after the equilibration procedure so that z displacement is allowed during the subsequent loading step.

Crack growth occurs in the LEFM simulation when the average K_I value along either crack front reaches K_{Ic}. A mixed-mode failure criterion is unnecessary, as K_{II} and K_{III} are negligible (i.e. $<$2.5% of K_I for all crack growth increments).

For the EPFM simulation, CTD_{crit} was calibrated at NASA Langley Research Center from a middle-crack tension (MT) test of the same material (C433-T39) and thickness as the ISP [59]. In that work, 3D FE simulations of the MT test revealed that simulated load versus crack extension matched experimental data when the mode I opening angle midway along the crack front at $d =$ 1.02 mm reached a critical value of 6.5°. This angle corresponds to CTD_{crit} through the relation

$$tan(6.5^\circ) = \frac{CTD_{crit}}{1.02}. \tag{7}$$

The same criterion is applied in the EPFM simulation by specifying for both crack fronts that CTD_{crit} must attain a value of 0.116 mm at $d = 1.02$ mm behind the crack front and that the criterion be evaluated midway along either crack front.

Crack growth occurs in increments of 1.15 mm (about 15% of the skin thickness), which is selected to be approximately the same as that implemented by Seshadri *et al.* in a similar simulation [59]. Straight crack fronts are enforced during crack growth in the skin of ISP. Upon entering the

intact stiffener, a crack front is evolved such that (1) a realistic, arbitrary crack front profile is represented (though the actual evolving crack front profile was not monitored during experiment) and (2) the new crack front profile has relatively smooth curvature to facilitate remeshing. A cross-section view of the panel in Fig. 19 shows different stages of simulated crack front evolution, from lead crack growth in the skin, to transition crack growth within the stiffener, to complete branching. The simulation proceeds as depicted in Fig. 8 until both initial crack fronts completely branch and P_{max} is attained.

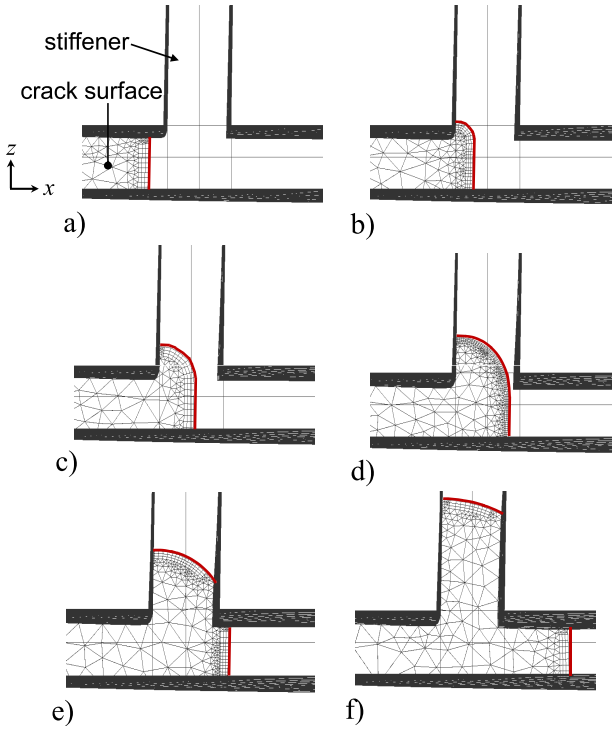


Fig. 19 Cross-sectional views of ISP mesh model taken at the crack plane and magnified at one stiffener. A thickened red line is overlaid along the crack front(s) at each step of crack growth. Views show lead bay crack before entering stiffener(a); transition crack evolution within stiffener (b,c,d); and complete branching into two distinct crack fronts (e,f).

Evaluation of the CTD criterion becomes nontrivial as a crack front transitions within the stiffener (i.e. while a crack front is within the stiffener but has not completely branched). Constraint effects introduced by the stiffener on the unsymmetric crack front profile, along with slight z displacement near the cracked region, lead to nonuniform and unsymmetric CTD values along the crack front. Evaluating the CTD criterion at only one point along the crack front becomes ambiguous to implement numerically and less representative, physically, of 3D crack growth behavior. A simple and efficient approach to address these issues is to compare CTD_{crit} to an average of CTD

values along the crack front. For transition crack growth in the stiffener, the middle third section of CTD points along the crack front are averaged and evaluated to predict crack propagation. Once the crack fully branches, the CTD criterion is again evaluated midway along each crack front.

Predicting crack front evolution for transition crack growth within the stiffener is not currently a fully-automated process. In this work, the straight portion of the crack front (see Fig. 19) is predicted using FRANC3D\NG and the curved portion of the front is specified by manually adding crack front points according to the two considerations described in the previous paragraph. Given a constraint-dependent CTD_{crit} relationship, a point-wise CTD criterion could be evaluated to evolve the arbitrarily-shaped crack front automatically.

The consequence of using an LEFM versus EPFM simulation to determine residual strength is quite obvious in the ISP case. Figure 20 shows load versus crack extension for both LEFM and EPFM simulations. Experimental load versus crack extension was not recorded during the tests; however, maximum applied load is plotted for two ISP tests of the same material and loading conditions. As in the Arcan simulation, the load to initiate crack extension is similar using either EPFM or LEFM methods since there are no residual stresses in the model at da=0. Following initiation, however, the EPFM simulation predicts that the applied load must be increased to maintain crack propagation. The necessary increase in applied load occurs since a significant amount of energy in the system is dissipated through plastic deformation. This effect cannot be predicted by the LEFM simulation since plasticity effects are not modeled. As a result, much of the energy in the ISP for the LEFM simulation must be dissipated through creation of new fracture surface area, which means less load is required to drive crack growth in the LEFM simulation than in the EPFM simulation. Using the EPFM framework, residual strength is determined within 2% of experimental average of the two tests. On the other hand, the LEFM method underpredicts the average residual strength by 64%. Although the LEFM simulation does predict an increase in applied load at the stiffener junction due to geometrical effects, the increase is negligible compared to that due to plastic deformation.

From the EPFM simulation, equivalent plastic strain field evolution in the ISP is depicted in Fig. 21 for the first and final crack steps. Accumulation of plastic strain in the wakes of the advancing crack fronts is relatively significant, extending from initial to final crack front locations.

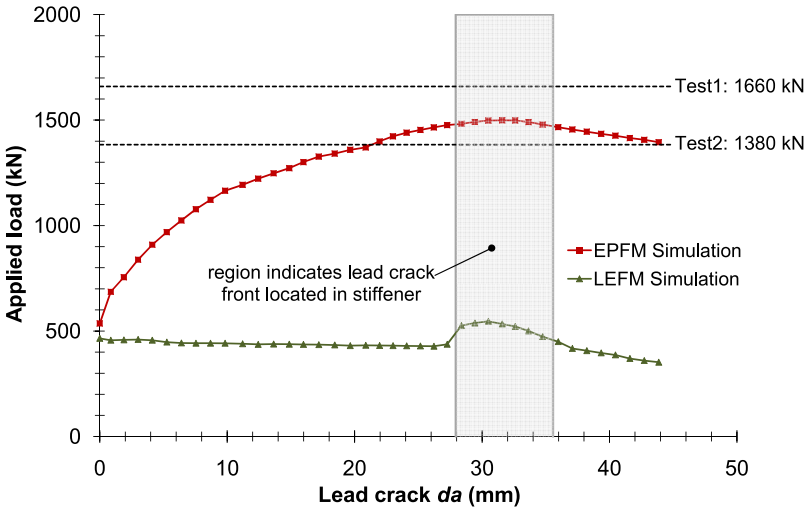


Fig. 20 Applied load (traction P, Fig. 16(b), integrated over applied area) versus half-crack extension, da, from ISP simulation. Maximum applied load is indicated for two corresponding tests conducted at Alcoa Technical Center. Shaded region indicates initially intact stiffener.

The general shape of 45° contour lobes at da=0 mm is maintained at da=44 mm both for the lead crack front extending into adjacent bay and for the crack front propagating in the z direction within the stiffener. At da=44 mm, the equivalent plastic strain in each stiffener extends in the direction of both contour lobes to the stiffener boundary. The consistent contour lobe shapes indicate that, despite increased z displacement as the crack propagates and severs initially intact stiffeners, both lead and branched crack fronts remain locally mode I dominant throughout tearing.

Finally, as evident in Fig. 21 for da=44 mm, the mapping procedure inevitably leads to imperfections in the fields due to diffusion of the FE solution, which occurs in regions of high gradients from repeated extrapolation and interpolation procedures, see III B. If mapping errors significantly affect crack growth predictions, mesh refinement should mitigate this effect.

IV. Conclusions

A surrogate model methodology and 3D elastic-plastic fracture simulation toolset have been presented, which enable accurate residual strength prediction of damaged structures in real time. The methodology and toolset are particularly useful for scenarios involving metallic aircraft structures subject to discrete-source damage during flight. An accurate prediction of structural residual strength in these scenarios could aid in avoidance of catastrophic crack growth and subsequent

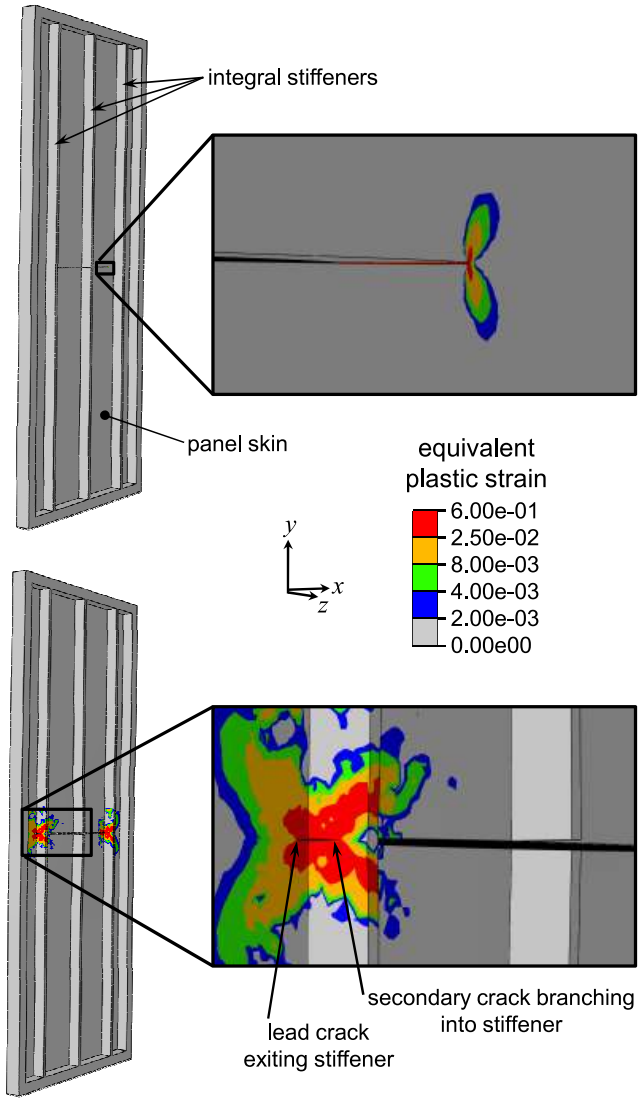


Fig. 21 Magnified views of simulated crack growth in ISP at half-crack extensions da=0 mm (top) and da=44 mm (bottom). Contours show evolution of equivalent plastic strain fields with crack growth. Deformation is not scaled. FE mesh is not shown for better contour visualization. Complete simulation can be viewed at *www.cfg.cornell.edu*.

structural failure.

The surrogate model methodology relies on offline numerical fracture simulations to obtain a set of data points describing residual strength as a function of discrete-source damage parameters. Strictly for illustration, a NN has been constructed as a surrogate model for predicting residual strength of a representative wing sub-structure subject to discrete-source damage. In the illustration, offline residual strength values have been determined using computationally efficient LEFM approximations. Subsequently, the consequences of using LEFM approximations for determining

residual strength of damaged metallic structures have been shown, and an EPFM framework to accurately determine residual strength using high-fidelity, 3D, elastic-plastic tearing simulations has been described. For an aluminum-alloy, integrally-stiffened panel exhibiting crack branching, residual strength is predicted within 2% of experiment using an EPFM simulation and is underpredicted by 64% using an LEFM simulation.

The more general and rigorous elastic-plastic tearing framework should be used to generate accurate residual strength training data, especially for cases involving discrete-source damage. Also, the FE model for the structure of interest should include enough detail to fully capture the relationship between a particular global loading state and onset of unstable crack growth. Furthermore, damage should be parameterized by taking into account onboard sensor characterization capability and resolution. With these considerations in mind, the general surrogate model methodology coupled with the EPFM simulation framework presented in this work provides a means of achieving more resilient and adaptive aircraft control.

V. Acknowledgments

The authors express gratitude to Drs. Robert Bucci and Mark James of Alcoa for providing valuable discussions and experimental details from the integrally-stiffened panel test program. Thank you also to Dr. Wilkins Aquino for providing guidance in the surrogate modeling aspect of this work. Funding was provided by NASA under contract NNX08AC50A with technical oversight provided by Drs. Edward Glaessgen and Thiagarajan Krishnamurthy of NASA Langley Research Center.

VI. References

[1] ABAQUS 6.8 Documentation, Dassault Systèmes Simulia Corp., 2008.

[2] Hughes, D. and Dornheim, M., "No Flight Controls," *Aviation Week & Space Technology*, Vol. 159, No. 23, 2003, pp. 42 – 43.

[3] "Depressurisation - 475 km north-west of Manila, Philippines - 25 July 2008," ATSB Transportation Safety Report 2, Australian Transport Safety Bureau, Nov 2009.

[4] Ingraffea, A., *Computational Fracture Mechanics*, Vol. 2 of *Encyclopedia of Computational Methods in Mechanics*, John Wiley and Sons, 2nd ed., 2007.

[5] Newman, J.C., Jr., "Finite-element analysis of crack growth under monotonic and cyclic loading," *ASTM STP 637*, 1977, pp. 56–80.

[6] Dawicke, D., Sutton, M., Newman, J.C., Jr., and Bigelow, C., "Measurement and analysis of critical CTOA for an aluminum alloy sheet," *ASTM STP 1220*, 1995.

[7] Deng, X. and Newman, J. C., "A study of some issues in stable tearing crack growth simulations," *Engineering Fracture Mechanics*, Vol. 64, No. 3, 1999, pp. 291 – 304.

[8] Seshadri, B., Newman, J.C., Jr., Dawicke, D., and Young, R., "Fracture analysis of the FAA/NASA wide stiffened panels," Tech. Rep. 208982, NASACP, 1999.

[9] Sutton, M. A., Boone, M. L., Ma, F., and Helm, J. D., "A combined modeling-experimental study of the crack opening displacement fracture criterion for characterization of stable crack growth under mixed mode I/II loading in thin sheet materials," *Engineering Fracture Mechanics*, Vol. 66, No. 2, 2000, pp. 171 – 185.

[10] Chen, C., Ingraffea, A., and Wawrzynek, P., "Prediction of Residual Strength and Curvilinear Crack Growth in Aircraft Fuselages," *AIAA Journal*, Vol. 40, aug 2002, pp. 1644–1652.

[11] Newman, J. C., Dawicke, D. S., and Seshadri, B. R., "Residual strength analyses of stiffened and unstiffened panels–Part I: laboratory specimens," *Engineering Fracture Mechanics*, Vol. 70, No. 3-4, 2003, pp. 493 – 507.

[12] Seshadri, B. R., Newman, J. C., and Dawicke, D. S., "Residual strength analyses of stiffened and unstiffened panels–Part II: wide panels," *Engineering Fracture Mechanics*, Vol. 70, No. 3-4, 2003, pp. 509 – 524.

[13] Dawicke, D., Newman, J.C., Jr., and Bigelow, C., "Three-dimensional CTOA and constraint effects during stable tearing in a thin-sheet material," *ASTM STP 1256*, 1995, pp. 358–379.

[14] Gullerud, A., Dodds, R.H., Jr., Hampton, R., and Dawicke, D., "Three-dimensional modeling of ductile crack growth in thin sheet metals: computational aspects and validation," *Engineering Fracture Mechanics*, Vol. 63, No. 4, 1999, pp. 347 – 374.

[15] Seshadri, B., Forth, S., Johnston, W.M., Jr., and Domack, M., "Application of CTOA/CTOD in the residual strength analysis of built-up and integral structures," *11th International Conference on Fracture*, Turin, Italy, March 2005.

[16] Erdogan, F. and Sih, G., "On the crack extension of plates under plane loading and transverse shear," *Journal of Basic Engineering*, Vol. 85, No. 4, 1963, pp. 519–527.

[17] Hussain, M., Pu, S., and Underwood, J., "Strain energy release rate for a crack under combined mode I and mode II," *ASTM STP 560*, 1974.

[18] Sih, G., "Strain-energy-density factor applied to mixed-mode crack problems," *International Journal of Fracture*, , No. 10, 1974, pp. 305–321.

[19] Sutton, M., Deng, X., Ma, F., Newman, J.C., Jr., and James, M., "Development and application of a crack tip opening displacement-based mixed mode fracture criterion," *International Journal of Solids and Structures*, Vol. 37, No. 26, 2000, pp. 3591 – 3618.

[20] Zuo, J., Deng, X., Sutton, M. A., and Cheng, C.-S., "Three-Dimensional Crack Growth in Ductile Materials: Effect of Stress Constraint on Crack Tunneling," *Journal of Pressure Vessel Technology*, Vol. 130, No. 3, 2008, pp. 031401.

[21] James, M. and Swenson, D., "A software framework for two-dimensional mixed mode I/II elastic-plastic fracture," *ASTM STP 1359*, 1999.

[22] Lan, W., Deng, X., and Sutton, M. A., "Three-dimensional finite element simulations of mixed-mode stable tearing crack growth experiments," *Engineering Fracture Mechanics*, Vol. 74, No. 16, 2007, pp. 2498 – 2517.

[23] Zuo, J., Deng, X., and Sutton, M. A., "Computational Aspects of Three-Dimensional Crack Growth Simulations," *ASME Conference Proceedings*, Vol. 2004, No. 47020, 2004, pp. 385–391.

[24] Queipo, N. V., Haftka, R. T., Shyy, W., Goel, T., Vaidyanathan, R., and Tucker, P. K., "Surrogate-based analysis and optimization," *Progress in Aerospace Sciences*, Vol. 41, No. 1, 2005, pp. 1 – 28.

[25] Reed, R. D. and Marks, R. J., *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, MIT Press, Cambridge, MA, USA, 1998.

[26] Kudva, J., Munir, N., and Tan, P., "Damage detection in smart structures using neural networks and finite-element analyses," *Smart Materials and Structures*, Vol. 1, No. 2, 1992.

[27] Ceravolo, R., Stefano, A. D., and Sabia, D., "Hierarchical use of neural techniques in structural damage recognition," *Smart Materials and Structures*, Vol. 4, No. 4, 1995, pp. 270–280.

[28] Liang, Y. C. and Hwu, C., "On-line identification of holes/cracks in composite structures," *Smart Materials and Structures*, Vol. 10, No. 4, 2001, pp. 599–609.

[29] Liu, S. W., Huang, J. H., Sung, J. C., and Lee, C. C., "Detection of cracks using neural networks and computational mechanics," *Computer Methods in Applied Mechanics and Engineering*, Vol. 191, No. 25-26, 2002, pp. 2831 – 2845.

[30] Ni, Y. Q., Wang, B. S., and Ko, J. M., "Constructing input vectors to neural networks for structural damage identification," *Smart Materials and Structures*, Vol. 11, No. 6, 2002, pp. 825–833.

[31] Lu, Y., Ye, L., Su, Z., Zhou, L., and Cheng, L., "Artificial Neural Network (ANN)-based Crack Identification in Aluminum Plates with Lamb Wave Signals," *Journal of Intelligent Material Systems and*

Structures, Vol. 20, No. 1, 2009, pp. 39–49.

[32] Su, Z. and Ye, L., *Application of Algorithms for Identifying Structural Damage - Case Studies*, Vol. 48 of *Lecture Notes in Applied and Computational Mechanics*, Springer Berlin / Heidelberg, 2009.

[33] Ouenes, A., "Practical application of fuzzy logic and neural networks to fractured reservoir characterization," *Computers & Geosciences*, Vol. 26, No. 8, 2000, pp. 953 – 962.

[34] Pidaparti, R., Jayanti, S., and Palakal, M., "Residual Strength and Corrosion Rate Predictions of Aging Aircraft Panels: Neural Network Study," *Journal of Aircraft*, Vol. 39, No. 1, 2002, pp. 175 – 180.

[35] Mohanty, S., Chattopadhyay, A., Peralta, P., and Das, S., "Bayesian Statistic Based Multivariate Gaussian Process Approach for Offline/Online Fatigue Crack Growth Prediction," *Experimental Mechanics*, 2010, pp. 1–11.

[36] Hambli, R., Chamekh, A., and H. Bel Hadj Salah, "Real-time deformation of structure using finite element and neural networks in virtual reality applications," *Finite Elements in Analysis and Design*, Vol. 42, No. 11, 2006, pp. 985 – 991.

[37] Sankararaman, S., Ling, Y., and Mahadevan, S., "Statistical inference of equivalent initial flaw size with complicated structural geometry and multi-axial variable amplitude loading," *International Journal of Fatigue*, Vol. 32, No. 10, 2010, pp. 1689 – 1700.

[38] Hinrichsen, R., Kurtz, A., Wang, J., Belcastro, C., and Parks, J., "Modeling Projectile Damage in Transport Aircraft Wing Structures," *AIAA Journal*, Vol. 46, Feb. 2008, pp. 328–335.

[39] Ramachandran, V., Raghuram, A., Krishnan, R., and Bhaumik, S., *Failure Analysis of Engineering Structures: Methodology and Case Histories*, chap. 6, ASM International, Materials Park, OH 44073-0002, 2005, p. 45.

[40] Mckay, M. D., Beckman, R. J., and Conover, W. J., "A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code," *Technometrics*, Vol. 42, No. 1, 2000, pp. 55–61.

[41] MATLAB R2009a Documentation, The MathWorks, Inc., 2009.

[42] Wawrzynek, P., Carter, B., and Ingraffea, A., "Advances in simulation of arbitrary 3D crack growth using FRANC3D/NG," *12th International Conference on Fracture*, Ottawa, Canada, July 2009.

[43] Broek, D., *Elementary Engineering Fracture Mechanics*, chap. 14, Martinus Nijhoff Publishers, 4th ed., 1986.

[44] Demuth, H. and Beale, M., *Neural Network Toolbox For Use with MATLAB (Version 4)*, The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098, 2004.

[45] Hagan, M. and Menhaj, M., "Training feedforward networks with the Marquardt algorithm," *IEEE*

Transactions on Neural Networks, Vol. 5, No. 6, Nov. 1994, pp. 989 –993.

[46] Hamby, D. M., "A review of techniques for parameter sensitivity analysis of environmental models," *Environmental Monitoring and Assessment*, Vol. 32, 1994, pp. 135–154, 10.1007/BF00547132.

[47] Spear, A., Veilleux, M., Bozek, J., and Ingraffea, A., "Structural Assessment and Prognosis Using a Multi-scale, Fracture Mechanics-based Approach," Austin, TX, May 2010.

[48] Newman, J. C., James, M. A., and Zerbst, U., "A review of the CTOA/CTOD fracture criterion," *Engineering Fracture Mechanics*, Vol. 70, No. 3-4, 2003, pp. 371 – 385.

[49] Sutton, M., Yan, J., Deng, X., Cheng, C., and Zavattieri, P., "Three-dimensional digital image correlation to quantify deformation and crack-opening displacement in ductile aluminum under mixed-mode I/III loading," *Optical Engineering*, Vol. 46, No. 5, 2007, pp. 051003.

[50] Lan, W., Deng, X., and Sutton, M. A., "Investigation of Crack Tunneling in Ductile Materials," *Engineering Fracture Mechanics*, Vol. In Press, Accepted Manuscript, 2010, pp. –.

[51] James, M., *A plane stress finite element model for elastic-plastic mode* I/II *crack growth*, Ph.D. thesis, Kansas State University, 1998.

[52] Lim, I., Johnston, I., Choi, S., and Murti, V., "An improved numerical inverse isoparametric mapping technique for 2D mesh rezoning," *Engineering Fracture Mechanics*, Vol. 41, No. 3, 1992, pp. 417 – 435.

[53] Arcan, M., Hashin, Z., and Voloshin, A., "A Method to Produce Uniform Plane-stress States with Applications to Fiber-reinforced Materials," *Experimental Mechanics*, , No. 141, 1977.

[54] Amstutz, B., Sutton, M., Dawicke, D., and Newman, J.C., Jr., "An experimental study of CTOD for mode I/mode II stable crack growth in thin 2024-T3 aluminum specimens," *ASTM STP 1256*, 1995, pp. 256–271.

[55] Amstutz, B., Sutton, M., Dawicke, D., and Boone, M., "Effects of mixed mode I/II loading and grain orientation on crack initiation and stable tearing in 2024-T3 aluminum," *ASTM STP 1296*, 1997, pp. 105–125.

[56] NASGRO Material Database, Southwest Research Institute®.

[57] Kulak, M., Bucci, R., Skyut, H., Bray, G., and Newman, J.C., Jr., "Fatigue crack growth and residual strength analysis of TWIST spectrum loaded integrally stiffened panels simulating a lower wing skin two bay crack scenario," *ICAF 20, 1999 - Structural Integrity for the Next Millenium*, edited by J. Rudd and R. Bader, Vol. I, International Committee on Aeronautical Fatigue, EMAS Publishing, Bellevue, Washington, USA, July 1999.

[58] Bray, G., Bucci, R., and Brazill, R., "Lessons Neglected: Effects of Moist Air on Fatigue and Fatigue Crack Growth in Aluminum Alloys," *7th International Conference on Aluminum Alloys*, Charlottesville,

VA, 2000.

[59] Seshadri, B., James, M., Johnston, W.M., Jr., Young, R., and Newman, J.C., Jr., "Recent developments in the analysis of monolithic structures at NASA Langley," *Sixth Joint FAA/DoD/NASA Conference on Aging Aircraft*, San Francisco, CA, 2002.

[60] de Jonge, J., Schutz, D., Lowak, H., and Schijve, J., "A standardized load sequence for flight simulation tests on transport aircraft wing structures," NLR Technical Report TR 73029 U, National Aerospace Laboratory NLR, Amsterdam, The Netherlands, 1973.

VII. Appendix

A. Python® Script to Evaluate *CTD* Criterion During Nonlinear Finite Element Analysis Using ABAQUS®

The script *CTDjobControlTerminal.py* executes an FE analysis using ABAQUS® and periodically interrupts the analysis to compute CTD values along one or multiple crack fronts and to subsequently evaluate a user-specified CTD_{crit} criterion. If CTD_{crit} is satisfied within the given tolerance, the script will terminate the FE analysis and echo to the command window the load increment when the criterion is satisfied. If there are multiple cracks in the model, the critical crack front will be identified and echoed to the command window. The script will also print the file *CTDResults.txt* to the current working directory. The file includes CTD values (decoupled into three modes) at each point along each crack front for every FE analysis interruption. The script includes function calls to the object libraries MeshTools and Vec3D, which are not included here. However, the functionality of each routine should be somewhat clear from the name of the function and comments in the script.

For the first step of crack growth, assuming the initial state is undeformed, *CTDjobControl.py* should be located in a current working directory and should be executed from the MS-DOS command window using a command like: "...>abaqus python CTDjobControl.py". The user will then be prompted for a series of input. Successful execution requires that the following files be located in the current working directory along with the script: (1) *.fdb (generated by FRANC3D\NG following crack insertion and remeshing) and (2) the deformed *.inp (only the model to be submitted for FE analysis). Additionally, the undeformed *.inp file (the undeformed local mesh model in a global-

local analysis) should be included in a sub-directory called *Undeformed* within the current working directory.

For subsequent steps of crack growth, a similar script, *CTDjobControlCall.py*, is called automatically following deformation mapping. Unlike *CTDjobControl.py*, *CTDjobControlCall.py* does not require the user to relocate files or execute a script to start the next FE analysis. While the two scripts differ slightly in how they are executed, the main functions are the same. Only *CTDjobControlCall.py* is included here, and electronic versions of both scripts are available for download at *www.cfg.cornell.edu*.

```
'''
CTDjobControlCall.py

Python script to control ABAQUS nonlinear FEA while
monitoring CTD at points located distance d behind crack.
written by ADS (October, 2009)
modified March, 2010

***if executing manually, use: "...>abaqus python CTDjobControlCall.py --
defInpFilePath undefInpFilePath fdbFilePath oldJobodbPath"***
Otherwsie the script will be executed automatically after mapping
deformation using runMapScript.py

'''

# *************************
import os, sys, time
from sys import argv, exit
import odbAccess as oa
from abaqusConstants import *
import Vec3D, MeshTools
import math
from math import sqrt
# *************************

# ************************* DEFINE HELPER FUNCTIONS ******************************
# ********************************************************************************
def askForFloat(number):
    # Request float from user
    response = float(raw_input(number))
    if response < 0 :
        print "Requested value must be positive.  Exiting."
        exit(0)
    else:
        return response

# ***************************************
def askForInt(number):
```

```
    # Request int from user
    try:
        response = int(raw_input(number))
    except:
        print "Response must be a positive integer!"
        exit(0)
    if response < 0 :
        print "Requested values must be a positive integer. Exiting."
        exit(0)
    else:
        return response

# ****************************************
def askForPath(prompt,extension):
    # Request path from user
    response = raw_input(prompt)
    while not os.path.exists(response):
        print "Specified path could not be located."
        print "Check that path exists and try again."
        response = raw_input(prompt)
    while not response.split('.')[-1] == extension:
        print "\n**File path should have extension '%s'**" % extension
        response = raw_input(prompt)
    return response

# ****************************************
def changeDirectory(pathToFile):
    path = str()
    tmp = pathToFile.split("/")
    for i in range(len(tmp)-1):
        path+=tmp[i]
        path+="/"
    print "\nThe cwd has been changed to: \n\t%s" % path
    os.chdir(path)

# ****************************************
def getFdb(path):
    fdb = str()
    # First try splitting on "."
    tmp = path.split(".")
    tmp[-1] = '.fdb'
    for i in range(len(tmp)):
        fdb+=tmp[i]
    # If doesn't exist, try splitting on "_"
    if not os.path.exists(fdb):
        fdb = str()
        tmp = path.split("_")
        tmp[-1] = '.fdb'
        for i in range(len(tmp)):
            fdb+=tmp[i]
        if not os.path.exists(fdb):
            print "The file %s cannot be located." % fdb
            exit(0)
    print "\nGetting %s" % fdb
    return fdb
```

```
# ****************************************
def askYesNo(question):
    # Request yes/no response from user
    # (Returns "Y" or "N")
    response = raw_input(question)
    while response.upper()[0] not in ("Y","N"):
        response = raw_input(question)
        if response.upper()[0] not in ("Y","N"):
            print "Wrong answer!  Try again"
    return response.upper()[0]

# ****************************************
def isInt(value):
    # Check if value is an integer
    try:
        int(value)
        return True
    except:
        return False

# ****************************************
def getGroupList(meshObj,listName,type):
    list = meshObj.GetGroupInfo(listName,type)
    if len(list) == 0:
        print "***No members found in group: %s ***" % listName
    return list

# ****************************************
def sortDictValsByKey(dict):
    # Returns a list of values sorted by keys
    sorted_list = []
    keys = dict.keys()
    keys.sort()

    for i in keys:
        sorted_list.append(dict[i])

    return sorted_list

# ****************************************
def printTimeStamp(start_time):
    # Print time stamp
    end_time = time.time()
    mins = int((end_time-start_time)/60)
    secs = round((end_time-start_time)%60,3)
    resultsFile = 'CTDResults.txt'
    file = open(resultsFile,"a")
    file.write('\n*************************************************')
    file.write('\nTotal time for run: %i:%g' % (mins,secs))
    file.write('\n*************************************************')
    file.close()

# *************** DEFINE JOB/ANALYSIS INTERACTION FUNCTIONS **********************
# ********************************************************************************
def runAnalysis(d,CTDc,eval,oldJob,mapSoln,undefInpFile,inpFile,fdbFile):
```

```
# ************************ runAnalyis SUB-FUNCTIONS ***************************
def submitJob():
    # Submits ABAQUS job and suspends it after data written .sta file
    print "\nSubmitting ABAQUS job '%s'..." % jobName
    if oldJob in "None":
        os.system('abaqus job='+jobName)
    else:
        print "Old job specified"
        os.system('abaqus job='+jobName+' oldjob='+oldJob)
    while not os.path.isfile(staFile):  # Wait for .sta file to be created
        continue
    # A hack at monitoring the sta file...
    fileSize = os.path.getsize(staFile)
    print fileSize
    fileSize_update = fileSize
    while fileSize == fileSize_update:
        fileSize_update = os.path.getsize(staFile)
    else:
        print fileSize_update
        os.system('abaqus job='+jobName+' suspend')

# ****************************************
def resumeJob():
    # Resume ABAQUS job
    os.system('abaqus job='+jobName+' resume')
    # Monitor the sta file
    fileSize = os.path.getsize(staFile)
    print fileSize
    fileSize_update = fileSize
    while fileSize == fileSize_update:
        fileSize_update = os.path.getsize(staFile)
    else:
        print fileSize_update
        os.system('abaqus job='+jobName+' suspend')

# ****************************************
def openOdb(jobName):
    odbFile = jobName+'.odb'
    # Try opening the odb file
    ct = 1
    while ct < 11:
        try:
            odb=oa.openOdb(path=odbFile,readOnly=TRUE)
            print "Opening the odb file..."
            return odb
        except:
            print "ERROR: Unable to open the specified odb %s!" % odbFile
        ct+=1
    exit(0)

# ****************************************
def getCurrentStatus():
    fd = open(staFile,"r")
    lines = fd.readlines()
    i = 1
    if len(lines[-i].split()) == 0:
```

```
        i+=1
    while isInt(lines[-i].split()[0]) == False :
        i+=1
        while len(lines[-i].split()) == 0:
            i+=1
    curr_step = int(lines[-i].split()[0])
    curr_inc = int(lines[-i].split()[1])

    return curr_step, curr_inc

# ****************************************
def previousIncrement(frac_status_per_front,front,eval,curr_inc,curr_step,\
                      interesting_nodes,mapped_disp,pts_per_front,\
                      main_side_elems,mate_side_elems,odbStep):
    # Evaluate CTD values at successive previous increments
    # until no values are supercritical.
    while frac_status_per_front[front] == 2:
        if curr_inc == 1:
            print "\nCTD(A) exceeds critical at crack front %i on load \
                  increment 1" % front
            print "Decrease initial inc size! See CTD results file for details"
            os.system('abaqus job='+jobName+' terminate')
            exit(0)
        else:
            print "\nStepping back one increment..."
            curr_inc-=1
            relative_disp = disps_per_inc[curr_inc]
            if not mapped_disp:
                abs_disp = relative_disp
            else:
                abs_disp = getAbsDisps(mapped_disp,relative_disp)
            updateDisplacements(mesh,abs_disp)
            flags_per_front = computeAndWriteCTDResults(mesh,tol,d,CTDc,\
                          inpFile,fdbFile,jobName,odbStep,curr_inc,\
                          pts_per_front,main_side_elems,mate_side_elems)
            frac_status_per_front = evaluateFractureCriterion(flags_per_front,eval)
    print "\nCrack front %i was supercritical at\n%s increment %i" \
           % (front,odbStep,curr_inc+1)
    print "May need to decrease FEA increment!  See results file."

# ***************************** runAnalysis MAIN ******************************
# Initializing required inputs...
jobName = inpFile.split("/")[-1].split(".")[0]   # Job name corresponds to .inp file
staFile = jobName+".sta"              # Status file corresponds to job
base = undefInpFile.split(".")[0]     # Undeformed .inp file in \Undeformed subdir.
mesh = MeshTools.MeshTools(base,"INP")
inc_interval = 1
i = inc_interval                      # Access the odb at every ith FEA increment
disps_per_inc = {}                    # A dict containing incremental displacements
disps_per_inc[0] = 0                  # Initialize the dictionary
flags_per_front = {}
eq_flag = 1                           # Flag to indicate equilibration step

# Calling CTD initializing functions...
    # Get mapped displacements from mapped_disps.txt (if any)
mapped_disp = getInitialDisps(mapSoln,inpFile)
```

```
    # Get main/mate side node groups
(main_side_nodes, mate_side_nodes) = getCrackFaceNodes(mesh)
    # Main/mate side elements along crack faces
(main_side_elems, mate_side_elems) = getCrackFaceElems(mesh,main_side_nodes,\
                                     mate_side_nodes)
    # Nodes belonging to main/mate side elements (only need to update disps
    # for these nodes during the analysis for computing CTD values)
interesting_nodes = getCrackFaceElemNodes(mesh,main_side_elems,mate_side_elems)
    # Determine points behind crack front(s) where CTDs computed ("CTD points")
pts_per_front = getPoints(main_side_elems,mate_side_elems,main_side_nodes,\
                          fdbFile,mesh,d)
    # Set tolerance distance
tol = setTol(mesh,main_side_nodes)

# JOB CONTROL:
# Initial job submission, suspension, and CTD computation
start_time = time.time()
submitJob()
(curr_step, curr_inc) = getCurrentStatus()
# Keep track of current step for disps_per_inc dictionary
# (clear the dictionary when we start a new step)
step = curr_step

# In some cases, increment info may not be written to odb yet,
# in which case disps_per_inc (and flags_per_front) remains empty.
while not flags_per_front:
    print "Waiting for displacements to be written to odb..."
    odb = openOdb(jobName)      # The odb must be closed then re-opened if
                                # we need information that has not been written
                                # to the odb at the current time of access
    (curr_step, curr_inc) = getCurrentStatus()
    (disps_per_inc, odbStep) = getRelDispsAtInterestingNodes(disps_per_inc,\
                               interesting_nodes,jobName,curr_step,curr_inc,odb)

    # Raise flag if done with equilibration step
    if odbStep.upper()[0:3] != "EQU":
        eq_flag = 0

    try:
        relative_disp = disps_per_inc[curr_inc] # Disps available at current inc?
        if not mapped_disp:     # If no initial mapped disps,
                                # then relative_disp=abs_disp
            abs_disp = relative_disp
        else:
            abs_disp = getAbsDisps(mapped_disp,relative_disp)

        updateDisplacements(mesh,abs_disp)
        flags_per_front = computeAndWriteCTDResults(mesh,tol,d,CTDc,\
                                                    inpFile,fdbFile,jobName,\
                                                    odbStep,curr_inc,pts_per_front,\
                                                    main_side_elems,mate_side_elems)
    except:
        odb.close()
        resumeJob()
        continue
```

```
# While the script is running, perform actions based on CTD flags returned.
# frac_status_per_front --> A dict where key is crack front id and value is:
#                           0 --> 'subcritical', continue
#                           1 --> critical, extend crack
#                           2 --> 'supercritical', back-up or decrease FEA inc
running = True
while running:
    if eq_flag == 0:
        # Evaluate the fracture criterion, sending to the function the flags
        # for all CTD(A) points at each crack front and the evaluation criterion
        # to be used
        frac_status_per_front = evaluateFractureCriterion(flags_per_front,eval)

        # First loop through all crack fronts and check for any that are
        # supercritical:(if there are, then the analysis will terminate as soon
        # as this is not the case or inform user to decrease load increment)
        for front in frac_status_per_front:
            if frac_status_per_front[front] == 2:
                previousIncrement(frac_status_per_front,front,eval,curr_inc,\
                                  curr_step,interesting_nodes,mapped_disp,\
                                  pts_per_front,main_side_elems,mate_side_elems,\
                                  odbStep)
                os.system('abaqus job='+jobName+' terminate')
                printTimeStamp(start_time)
                exit(0)

        # Next loop through all crack fronts and check for any that are critical:
        # (if there are, then analysis will terminate at current step/increment)
        critical = 0
        for front in frac_status_per_front:
            if frac_status_per_front[front] == 1:
                print "\n***Crack front %i is within tolerance of critical***"\
                      % front
                critical = 1
        if critical == 1:
            print "\nExtend critical crack(s) at increment %d of step %s"\
                  % (curr_inc,odbStep)
            os.system('abaqus job='+jobName+' terminate')
            printTimeStamp(start_time)
            exit(0)

    # Finally, if analysis has not terminated, all crack fronts are subcritical
    odb.close() # Close odb to update add'l analysis increments
    resumeJob()
    (curr_step, curr_inc) = getCurrentStatus()
    # If we started a new step, clear disps_per_inc dictionary and start over
    if curr_step > step:
        disps_per_inc.clear()
        disps_per_inc[0] = 0
        step = curr_step
    odb = openOdb(jobName)
    (disps_per_inc, odbStep) = getRelDispsAtInterestingNodes(disps_per_inc,\
                                                interesting_nodes,jobName,\
                                                curr_step,curr_inc,odb)
    # Raise flag if done with equilibration step
    if odbStep.upper()[0:3] != "EQU":
```

```
            eq_flag = 0

        relative_disp = disps_per_inc[curr_inc]
        if not mapped_disp:
            abs_disp = relative_disp
        else:
            abs_disp = getAbsDisps(mapped_disp,relative_disp)
        updateDisplacements(mesh,abs_disp)
        flags_per_front = computeAndWriteCTDResults(mesh,tol,d,CTDc,inpFile,\
                                                    fdbFile,jobName,odbStep,curr_inc,\
                                                    pts_per_front,main_side_elems,\
                                                    mate_side_elems)


# ************************* DEFINE CTD FUNCTIONS ******************************
# *****************************************************************************
def getInitialDisps(mapSoln,filePath):
    # Retrieve mapped displacements from mapped_disp.txt. Return empty dict if none.
    mapped_disp = {}       # Dict of mapped (initial) nodal displacements

    if mapSoln == "N":
        print "***Returning empty mapped_disp dict***"
        return mapped_disp

    elif mapSoln == "Y":
        changeDirectory(filePath)
        disp_file = 'mapped_disp.txt'

        # If the disp.txt file doesn't exist, exit
        if not os.path.exists(disp_file):
            print "***Could not locate mapped_disp.txt from previous analysis!***"
            print "(Make sure the file is located with the previous mesh file)"
            exit(0)

        file = open("mapped_disp.txt","r")
        buff = file.readline()
        while buff:
            data = buff.split()
            if data[0].upper()[0:4] == "DISP":
                buff = file.readline()
                data = buff.split()
                while buff:
                    nid = int(data[0])
                    tmp = []
                    for i in range(3):
                        tmp.append(float(data[i+1]))
                    mapped_disp[nid] = tmp
                    buff = file.readline()
                    data = buff.split()
            buff = file.readline()

    file.close()
    print "***Retrieved initial displacements for %d nodes" % len(mapped_disp)
    return mapped_disp

# *****************************************
```

```
def getCrackFaceNodes(mesh):

    main_side_nodes = []
    mate_side_nodes = []

    try:
        main_side_nodes = getGroupList(mesh,"main_side_nodes","node")
        mate_side_nodes = getGroupList(mesh,"mate_side_nodes","node")
    except:
        try:
            main_side_nodes = getGroupList(mesh,"all_main_side_nodes","node")
            mate_side_nodes = getGroupList(mesh,"all_mate_side_nodes","node")
        except:
            print "***No main/mate side nodes found for previous mesh!***"
            exit(0)

    return main_side_nodes, mate_side_nodes

# ****************************************
def getCrackFaceElems(mesh,main_side_nodes,mate_side_nodes):
    # Make sets of main/mate side crack face elements
    print "\nCollecting elements along the main and mate side crack face..."
    main_side_elems = set()
    mate_side_elems = set()

    for nid in main_side_nodes:
        elist = mesh.GetAdjacentElems(nid)
        for eid in elist: main_side_elems.add(eid)

    for nid in mate_side_nodes:
        elist = mesh.GetAdjacentElems(nid)
        for eid in elist: mate_side_elems.add(eid)

    return main_side_elems, mate_side_elems

# ****************************************
def getCrackFaceElemNodes(mesh,main_side_elems,mate_side_elems):
    # Gather only nodes of interest for updating displacements
    nodes = []
    for eid in main_side_elems:
        nids = mesh.GetElemInfo(eid)
        for n in nids:
            if n not in nodes:
                nodes.append(n)

    for eid in mate_side_elems:
        nids = mesh. GetElemInfo(eid)
        for n in nids:
            if n not in nodes:
                nodes.append(n)
    return nodes

# ****************************************
def getPoints(main_side_elems,mate_side_elems,main_side_nodes,fdbFile,mesh,d):
    print "\nDetermining coordinates of CTD(A) points %g behind \
           crack front..." % d
```

```
# First, create a dictionary of crack fronts as keys and
# lists of crack front node id's as values (from the fdb)
front_nodes = {}
nodes = []
f=0

fdb = open(fdbFile,'r')

buff = fdb.readline()
while buff:
    vals = buff.split()
    if vals[0].upper() == "NUM_FRONTS:":
        nfronts = int(vals[1])
        break
    buff = fdb.readline()

buff = fdb.readline()
while len(front_nodes) < nfronts:
    buff = fdb.readline()
    vals = buff.split()
    if vals[0].upper() == "FRONT_NODES:":
        n = int(vals[1])
        while len(nodes) < n:
            buff = fdb.readline()
            vals = buff.split()
            for i in range(len(vals)):
                nodes.append(int(vals[i]))
        front_nodes[f] = nodes
        nodes = []
        f+=1
fdb.close()

if len(front_nodes) != nfronts:
    print "Only found %i crack fronts in %s" % (i+1,fdbFile)
#print "\nHere is the front_nodes dict:\n"
#print front_nodes

# Second, create a nested dictionary where, for each crack front id (key),
# the value is a dictionary with front node id's as keys and corresponding
# Vec3D points located d distance "behind" each crack front node as values.
#
# pts_per_front = {crack_front_id: {front_node_id: CTD_point}}
#
pts_per_front = {}

# Loop over the crack fronts (handles multiple fronts)
for front in front_nodes:
    pts = {}
    # For this crack front, loop over the crack front nodes
    for nid in front_nodes[front]:
        flag = 0
        adj_nodes = mesh.GetAdjacentCornerNodes(nid)
        # Rely on topology to find the adjacent node directly
        # "behind" the crack front node (this approach may break
        # down in certain instances, e.g. if crack front mesh
```

```
            # template is not used.)  The immediately adjacent
            # nodes on the main and mate surfaces should be initially
            # coincident, so just query the main surface node set for
            # a match.  Note that for quadratic elements, only corner
            # nodes will return an adjacent corner node.
            for node in adj_nodes:
                if node in main_side_nodes:
                    flag = 1    # the front node is a corner node
                    break
            if flag == 0:        # Mid-side nodes along front don't have
                continue        # adjacent nodes on the crack face
            adj_ncoord = mesh.GetNodeCoords(node)
            front_ncoord = mesh.GetNodeCoords(nid)
            # Generate vector pointing from the crack front node
            # to the adjacent surface node
            vec = []
            for i in range(3):
                vec.append(adj_ncoord[i]-front_ncoord[i])
            mag = sqrt(pow(vec[0],2)+pow(vec[1],2)+pow(vec[2],2))
            # Now get point coordinates using the unit vector
            # components multiplied by the user specified distance, d.
            x = front_ncoord[0] + vec[0]/mag*d
            y = front_ncoord[1] + vec[1]/mag*d
            z = front_ncoord[2] + vec[2]/mag*d
            pts[nid] = Vec3D.Vec3D(x,y,z)
        pts_per_front[front] = pts
    print "\nReturning CTD(A) points at d=%g behind crack front nodes..." % d
    print pts_per_front
    print len(front_nodes)
    return pts_per_front


# ****************************************
def setTol(mesh,main_side_nodes):
    # Set PointInside tolerance to an adequate value local to the crack face:
    # 1/1000 of the smallest crack face edge length.  This will be used in
    # the functions IsPointOutsideMesh (to check CTD pts) and GetElemsAtPoint
    min_length = mesh.GetMaxDimension() # A starting dimension
    print "\nSetting tolerance for MeshTools functions..."
    for nid in main_side_nodes:
        elengths = mesh.GetAdjacentSurfEdgeLengths(nid)
        min_length = min(min_length,min(elengths))

    tol = min_length/100.0
    mesh.SetPointInsideTolerance(tol)
    return tol

# ****************************************
def getRelDispsAtInterestingNodes(disps_per_inc,nodes,jobName,curr_step,curr_inc,odb):
    # Get displacements at the nodes of interest for the current load step/inc.
    # These are "relative" displacements because they do not account for initial
    # mapped displacements.
    print "\nAccessing odb to get displacements for nodes of interest"
    print "through load step %i increment %i..." % (curr_step,curr_inc)

    # Fill dictionary of [x,y,z] displacements for all relevent nodes
```

```
    relative_disps = {}

    try:
        odbStep = odb.steps.keys()[curr_step-1]
    except:
        print "Looks like current step has not been written to odb file. Exiting."
        exit(0)

    # Loop through increments, from most recent stored increment to current
    for odbFrame in range(max(disps_per_inc)+1,curr_inc+1):
        try:
            fieldObject = odb.steps[odbStep].frames[odbFrame].fieldOutputs['U']
            print "Increment %i..." % odbFrame
            for nid in nodes:
                values = fieldObject.values[nid-1].data
                relative_disps[nid] = values
            disps_per_inc[odbFrame] = relative_disps
            relative_disps = {}
        except:
            print "No odb info yet for %s increment %i" % (odbStep,odbFrame)

    return disps_per_inc, odbStep
    print "Got 'em!"


# ****************************************
def getAbsDisps(mapped_disp,relative_disp):
    # Function to get absolute nodal displacements by summing the
    # mapped displacements with the current relative displacements.
    print "\nGetting absolute disps for nodes of interest at current step/inc..."
    abs_disp = {}

    for nid in relative_disp:
        disp = []
        for i in range(3):
            disp.append(relative_disp[nid][i]+mapped_disp[nid][i])
        abs_disp[nid] = disp
    return abs_disp
    print "Got 'em!"

# ****************************************
def updateDisplacements(mesh,abs_disp):
    # Write a temporary file of absolute displacements for nodes of interest
    # to be read into mesh results (update to current displacements)
    print "\nUpdating current displacements..."
    tmpfile = open('tmp_current_disps.txt',"w")
    for nid in abs_disp:
        line = '%d\t%.15f\t%.15f\t%.15f\n' %(nid,abs_disp[nid][0],\
                                             abs_disp[nid][1],abs_disp[nid][2])
        tmpfile.write(line)
    tmpfile.close()
    mesh.ReadFeawdCpDisps('tmp_current_disps.txt')
    os.system('del tmp_current_disps.txt')

# ****************************************
def computeAndWriteCTDResults(mesh,tol,d,CTDc,inpFile,fdbFile,jobName,odbStep,\
```

```
                    curr_inc,pts_per_front,main_side_elems,mate_side_elems):
# Compute CTD results and write to input file directory
changeDirectory(inpFile)

# Partition CTD values into subcritical, critical, and supercritical
flags_per_front = {}    # A dictionary where key is crack front id
                        # and value is list of flags for all CTD(A) points
flag = 0                # Flag written to results file

# Write the file header
resultsFile = 'CTDResults.txt'
if os.path.isfile(resultsFile):
    file = open(resultsFile,"a")
else:
    file = open(resultsFile,"w")
    file.write('CTD Results\n')
    file.write('Results provided along crack fronts (z_min to z_max)\n\n')
    file.write('\n************************************************************')
# Write sub-header
file.write('\n\nMesh file: %s\n' % inpFile)
file.write('Crack file: %s\n' % fdbFile)
file.write('Job name: %s\n' % jobName)
file.write('ABAQUS Step Name: %s\n' % odbStep)
file.write('Frame: %i\n' % curr_inc)
file.write('Distance behind crack front nodes: %g\n' % d)
file.write('Critical opening displacement: %g\n\n' % CTDc)

# Write CTD results for CTD points at each crack front
for front in pts_per_front:
    print "front"
    flags = []  # A list of flags for each CTD(A) point

    file.write('\nCrack Front ID: %i\n' % front)
    file.write('\nCorresponding\tCTD_I\t\tCTD_II\t\tCTD_III\tCTD_mag\tFlag')
    file.write('\tCTD_ptZCoord\nFrontNodeId\n')
    file.write('------------------------------------------------------------\n')

    temp_dict = {}
    print pts_per_front[front]
    # First sort the front node id's by their z-coords.
    for nid in pts_per_front[front]:                    # Unsorted
        ncoords = mesh.GetNodeCoords(nid)
        zcoord = ncoords[2]
        temp_dict[zcoord] = nid
    sorted_front_nids = sortDictValsByKey(temp_dict)  # Sorted

    # Move along crack front, computing CTD at points "behind"
    # each front node ID
    for nid in sorted_front_nids:
        CTD_pt = pts_per_front[front][nid]
        status,dist = mesh.IsPointOutsideMesh(CTD_pt)
        if status != -2:
            file.write('***CTD_pt (%g, %g, %g) is outside mesh and \
                      has been skipped!' % (CTD_pt[0],CTD_pt[1],CTD_pt[2]))
            file.write(' (Behind front node %i)***' % nid)
            continue
```

```
# Associate CTD_pt with elems on main/mate faces
# (If less than two elems found, decrease meshTools tol and try again)
# Reset MeshTools tol each time in case it was increased previously
mesh.SetPointInsideTolerance(tol)
elist = mesh.GetElemsAtPoint(CTD_pt)
initial_tol = tol

# Get displacement on the main side crack face for this CTD point
# (displacement accounts for initial mapped displacements)
main_eid = -1
while main_eid == -1:
    for eid in elist:
        # Loop through elems until find main_side element
        if eid in main_side_elems:
            #print "main side elem for nid %g: %g" % (nid,eid)
            main_eid = eid
            break
    # If found main_side element, jump out of while loop
    if main_eid != -1:
        continue
    # Otherwise reset the tolerance and try again
    else:
        #print "newtol"
        new_tol = initial_tol*10.0
        #print new_tol
        mesh.SetPointInsideTolerance(new_tol)
        elist = mesh.GetElemsAtPoint(CTD_pt)
        initial_tol = new_tol
disp1 = mesh.GetPtDisp(CTD_pt,main_eid)
ct=1
# Get displacement on the mate side crack face for this CTD point
# (displacement accounts for initial mapped displacements)
mate_eid = -1
while mate_eid == -1:
    for eid in elist:
        #print "elem:"
        #print eid
        # Loop through elems until find mate_side element
        if eid in mate_side_elems:
            #print "mate side elem for nid %g: %g" % (nid,eid)
            mate_eid = eid
            break
    # If found mate_side element, jump out of while loop
    if mate_eid != -1:
        continue
    # Otherwise reset the tolerance and try again
    else:
        #print "newtol"
        new_tol = initial_tol*10.0
        #print new_tol
        mesh.SetPointInsideTolerance(new_tol)
        elist = mesh.GetElemsAtPoint(CTD_pt)
        initial_tol = new_tol
disp2 = mesh.GetPtDisp(CTD_pt,mate_eid)
```

```
            # Sometimes same element associated for main and mate surfaces
            # (notice especially with poorly shaped elements around crack front).
            # Check for this condition and exit if that's the case.
            if main_eid == mate_eid:
                print "Same main/mate elem found for CTD pt behind node %g" % nid
                print "Check the quality of the mesh. Exiting!"
                exit(0)

            # Compute CTD's and raise flag if critical value is reached
            # NOTE: The specific fracture criterion evaluation is handled in the
            # function evaluateFractureCriterion
            CTD = disp1 - disp2    # vector of disps
            CTD_I = abs(CTD[1])   # Mode I --> y-disp
            CTD_II = abs(CTD[0])  # Mode II--> x-disp
            CTD_III = abs(CTD[2]) # Mode III-> z-disp
            CTD_mag = sqrt(pow(CTD_I,2)+pow(CTD_II,2)+pow(CTD_III,2))
            tol = CTDc*0.02    # Tolerance is 2% of specified critical value
            if abs(CTD_mag-CTDc) < tol:   # CTD = CTDc +- tol
                flag = 1
                flags.append(flag)
            elif CTD_mag > CTDc:          # CTD >> CTDc
                flag = 2
                flags.append(flag)
            else:
                flag = 0                    # CTD << CTDc
                flags.append(flag)
            file.write('%i\t\t%g\t%g\t%g\t%g\t%i\t%g\n' % \
                       (nid,CTD_I,CTD_II,CTD_III,CTD_mag,flag,CTD_pt[2]))

        flags_per_front[front] = flags
    print "\nDone writing to the results file!"
    file.close()
    return flags_per_front

# ****************************************
def evaluateFractureCriterion(flags_per_front,eval):

    # eval --> =0.0 corresponds to mid-thickness only option
    #      --> !=0 corresponds to % of CTD points that must meet criterion

    frac_status_per_front = {}  # A dictionary where key is crack front id
                                # and value is:
                                # 0 --> subcritical, continue
                                # 1 --> critical, extend
                                # 2 --> supercritical, back-up

    # For the mid-thickness only criterion evaluation:
    if eval == 0.0:
        # Loop through each crack front
        for front in flags_per_front:
            index = []
            # If even number of points through-thickness,
            # get indices of the middle two points
            if len(flags_per_front[front]) % 2 == 0:
                index.append((len(flags_per_front[front])-2)/2)
                index.append(len(flags_per_front[front])/2)
```

```
            # Otherwise, if odd number of points through-thickness,
            # get index of the middle CTD(A) point
            else:
                index.append((len(flags_per_front[front])-1)/2)
            for i in index:
                # The critical case:
                if flags_per_front[front][i] == 1:
                    frac_status_per_front[front] = 1
                    break
                # The supercritical case:
                elif flags_per_front[front][i] == 2:
                    frac_status_per_front[front] = 2
                    break
                # The subcritical case:
                elif flags_per_front[front][i] == 0:
                    frac_status_per_front[front] = 0
                    break

    # For the evaluation case where eval% of points must meet criterion
    elif eval != 0.0:
        # Loop through each crack front
        for front in flags_per_front:
            # Count number of sub-, super-, and critical flags along front
            sub = 0     # Clear the count
            super = 0   # Clear the count
            crit = 0    # Clear the count
            total = len(flags_per_front[front])
            for flag in flags_per_front[front]:
                if flag == 1:
                    crit+=1
                elif flag == 2:
                    super+=1
                elif flag == 0:
                    sub+=1
            # The critical case:
            if (float(crit)/float(total)) >= eval:
                frac_status_per_front[front] = 1
                continue
            # The supercritical case:
            elif (float(super)/float(total)) >= 1-eval:
                frac_status_per_front[front] = 2
                continue
            # The subcritical case:
            else:
                frac_status_per_front[front] = 0


    return frac_status_per_front

# ********************************** MAIN ***************************************
# *******************************************************************************
# NOTE: This script script cannot be executed from the CAE if *MAP SOLUTION is
# specified since the keyword is not yet supported by the CAE (and consequently
# the old job cannot be specified)
if __name__ == "__main__":
```

```
print __doc__

inpFile = sys.argv[-4]          # deformed .inp file (current crack step)
undefInpFile = sys.argv[-3]     # undeformed .inp file in Undeformed sub-dir
fdbFile = sys.argv[-2]          # .fdb file generated by FRANC3D\NG
odbFile = sys.argv[-1]          # .odb file of previous analysis that ABAQUS
                                # should use to map solution

changeDirectory(inpFile)

if not odbFile.upper() == "NONE":
    # Old job path
    oldJob = odbFile.split(".")[0]
    mapSoln = "Y"

    # Check for all the necessary restart files of the old job
    res = oldJob+'.res'
    mdl = oldJob+'.mdl'
    stt = oldJob+'.stt'
    prt = oldJob+'.prt'
    if not os.path.exists(res):
        print "\tCannot find restart file for the oldJob."
        print "\tCheck that it exists. Exiting."
        exit(0)
    if not os.path.exists(mdl):
        print "\tCannot find .mdl file for the oldJob."
        print "\tCheck that it exists. Exiting."
        exit(0)
    if not os.path.exists(stt):
        print "\tCannot find stt file for the oldJob."
        print "\tCheck that it exists. Exiting."
        exit(0)
    if not os.path.exists(prt):
        print "\tCannot find restart file for the oldJob."
        print "\tCheck that it exists. Exiting."
        exit(0)

else:
    oldJob = "None"
    mapSoln = "N"

# Check that status file doesn't already exist.  If it does, delete it so that
# we don't tail the existing file
staFile = inpFile.split(".")[0]+".sta"
if os.path.isfile(staFile):
    response = raw_input("\nThe file %s already exists.  Overwrite file? (y/n): "\
               % staFile)
    if response.upper()[0] == "N":
        print "Exiting program."
        exit(0)
    elif response.upper()[0] == "Y":
        os.system('del '+staFile)
    else:
        print "Unknown response. Exiting."
        exit(0)
```

```
# Request CTD parameters:
print "---------------- USER INPUTS ------------------"
# --> Distance, d, behind crack front
d = askForFloat("\nMonitor opening at this distance behind crack front: ")
# --> CTD or CTOA
response = raw_input("\nWould you like to specify critical 'CTD' or 'CTOA'?: ")
while response.upper() not in ("CTD","CTOA"):
    response = raw_input("\n\tSpecify 'CTD' or 'CTOA': ")
if response.upper() in "CTD":
    CTDc = askForFloat("\n\tCritical CTD: ")
elif response.upper() in "CTOA":
    CTOAc = askForFloat("\n\tCritical CTOA (degrees): ")
    # Convert CTOAc value to CTDc for Mode I
    rad = CTOAc*math.pi/180
    CTDc = rad*d
# --> Criterion
print "\nHow is fracture criterion evaluated?"
response = askYesNo("\n\tEvaluate at mid-thickness only?: ")
if response in "Y":
    eval = 0.0
if response in "N":
    print "\n\tCriterion will be evaluated at a percentage of points"
    response = askForInt("\n\tSpecify percentage of points that must meet criterion: ")
    eval = float(response)/100

print "\n---------------- STARTING FEA ------------------"
# Start the analysis
runAnalysis(d,CTDc,eval,oldJob,mapSoln,undefInpFile,inpFile,fdbFile)
```

B. Deformation Mapping Script

Two scripts are provided for mapping deformation from the previous mesh to the current, undeformed mesh generated by FRANC3D\NG. The first script, called *runMapScript.py*, simply prompts the user for a series of input, including paths to various required files. The user will also be asked whether or not the analysis is a local-global analysis (i.e. if the model has global and sub-regions). If it is, then displacements on the global region will simply be transferred from old to new meshes, as the global mesh remains unchanged and does not require invoking the inverse isoparametric mapping routine. The user will also be asked whether or not the previous analysis is a restart analysis so that the correct restart files are accessed and displacements are taken from the correct load increment. The script can be run from any directory. The user should execute the script from the MS-DOS command window using a command like: "...>abaqus python runMapScript.py".

The script *runMapScript.py* automatically calls the script *masterMapDisplacements.py*, which retrieves displacements from the previous FE analysis, maps them onto the current undeformed

mesh, and generates a deformed mesh file (*.inp) for the current crack increment. The deformed *.inp file will contain the *Map Solution ABAQUS [1] keyword and an equilibration step, during which all nodes with applied boundary conditions are fixed. The script also generates the sub-directory *Undeformed* and moves the original undeformed mesh into that directory. Output and print statements during mapping are written to the abaqus.rpy file in the current working directory.

After mapping, the script *CTDjobControlCall.py* is called and the next FE analysis automatically begins and continues until $CTD \approx CTD_{crit}$.

Successful execution of these scripts require the compiled libraries Vec3D, ColTensor, FullTensor, and MeshTools to be located in a directory accessible by ABAQUS® (e.g. in the ABAQUS>Python>Obj directory). Elements currently supported for mapping include linear and quadratic brick elements (C3D8 and C3D20), quadratic wedge elements (C3D15), and quadratic tetradhedral elements (C3D10).

Electronic versions of the deformation mapping scripts are available for download at *www.cfg.cornell.edu*.

```
,,,
runMapScript.py

This script:
    1) generates a .model file from the current undeformed
       .inp file,
    2) generates mapped displacements from old mesh to new
       undeformed mesh,
    3) appends these displacements to the .model file,
    4) moves the undeformed .inp file to a new directory, and
    5) rewrites the .inp file in the deformed configuration,
    6) automatically calls CTDjobControlCall.py to start the
       nextFEA if "yes" at prompt.

Requires an initial undeformed .inp file for the current
step, a .model file for the previous step (with a header
for mapped displacements), as well as the following compiled
scripts:
Vec3D, ColTensor, FullTensor, MeshTools

AD Spear, MG Veilleux, and JD Hochhalter
(written October 2009 with subsequent modifications)

,,,
# *************************
import os, sys
```

```
from sys import exit
from abaqusConstants import *
import odbAccess

# *************************
def askForPath(prompt,extension):
    # Request path from user
    response = raw_input(prompt)
    while not os.path.exists(response):
        print "Specified path could not be located."
        print "Check that path exists and try again."
        response = raw_input(prompt)
    while not response.split('.')[-1] == extension:
        print "\n**File path should have extension '%s'**" % extension
        response = raw_input(prompt)
    return response

# *************************
def askYesNo(question):
    # Request yes/no response from user
    # (Returns "Y" or "N")
    response = raw_input(question)
    while response.upper()[0] not in ("Y","N"):
        response = raw_input(question)
        if response.upper()[0] not in ("Y","N"):
            print "Wrong answer!  Try again"
    return response.upper()[0]

# ****************************************
def getFdb(path):
    fdb = str()
    # First try splitting on "."
    tmp = path.split(".")
    tmp[-1] = '.fdb'
    for i in range(len(tmp)):
        fdb+=tmp[i]
    # If doesn't exist, try splitting on "_"
    if not os.path.exists(fdb):
        fdb = str()
        tmp = path.split("_")
        tmp[-1] = '.fdb'
        for i in range(len(tmp)):
            fdb+=tmp[i]
        if not os.path.exists(fdb):
            print "The file %s cannot be located." % fdb
            exit(0)
    print "\nGetting %s" % fdb
    return fdb

# *************************
def getOdb(path,localglobal,res):
    odb = str()
    tmp = path.split(".")
    if localglobal == "Y" and res == "N":
        tmp[-1] = '_full.odb'
    elif localglobal == "Y" and res == "Y":
```

```
        tmp[-1] = '_full_res.odb'
    elif localglobal == "N" and res == "Y":
        tmp[-1] = '_res.odb'
    else:
        tmp[-1] = '.odb'
    for i in range(len(tmp)):
        odb+=tmp[i]
    while not os.path.exists(odb):
        print "The file %s cannot be located." % odb
        odb = raw_input("\nEnter path to corresponding odb file:\n")
    print "\nGetting %s" % odb
    return odb

# *************************
def askForInt(number):
    # Request int from user
    response = int(raw_input(number))
    while response < 0 :
        print "Requested value must be a positive integer.  Try again."
        response = int(raw_input(number))
    return response

# ********************************** MAIN ***************************************
# *******************************************************************************

if __name__ == "__main__":

    print __doc__
    currInpFile = ""
    currInpFile_full = ""

    # Ask user if analysis is local-global analysis
    prompt0 = "Is analysis local-global \
              (i.e. *_full files generated by F3DNG)?:\n"
    localglobal = askYesNo(prompt0)

    # Request .inp file of job from which displacements will be mapped
    # (if local-global, just the local input file generated by F3DNG)
    if localglobal == "Y":
        prompt1 = "\nPath of old, undeformed input file \
                  (local region)\n(e.g. C:/.../oldMesh.inp):\n"
        prevInp = askForPath(prompt1,'inp')
    else:
        prompt1 = "\nPath of old, undeformed input file \
                  \n(e.g. C:/.../oldMesh.inp):\n"
        prevInp = askForPath(prompt1,'inp')

    # Get top level directory to search for .fdb and .odb files
    tmp = prevInp.split("/Undeformed/")
    dir = tmp[0]+"/"+tmp[-1]
    print dir

    # Search for corresponding .fdb file
    fdbPrev = getFdb(dir)
    print fdbPrev
```

```
# If the analysis is local-global, need both
# the *.inp and *_full.inp files
if localglobal == "N":
   # Request current .inp file to which displacements will be mapped
    prompt2 = "\nPath of current .inp file to which displacements \
               will be mapped\n(e.g. C:/.../newMesh.inp):\n"
    currInpFile = askForPath(prompt2,'inp')
    # Check for existence of corresponding .fdb file
    fdbCurr = getFdb(currInpFile)

elif localglobal == "Y":
    # Request *.inp generated by F3DNG
    prompt2a = "\nPath of most recent (local) *.inp file \
                generated from F3DNG:\n"
    currInpFile = askForPath(prompt2a,'inp')
    # Check for existence of corresponding .fdb file
    fdbCurr = getFdb(currInpFile)
    # Also check for the *_full.inp of the current, joined file
    tmp = currInpFile.split(".")
    tmp[-1] = "_full.inp"
    for i in range(len(tmp)):
        currInpFile_full+=tmp[i]
    if not os.path.exists(currInpFile_full):
        print "Cannot locate the file %s" % currInpFile_full
        exit(0)
    else:
        print "Retrieving %s" % currInpFile_full

# Search for .odb file of old mesh and access the file
prompt0 = "\nMap disps. from a *_full_res.odb file?:\n"
res = askYesNo(prompt0)
odbFile = getOdb(dir,localglobal,res)
try:
    odb = odbAccess.openOdb(path=odbFile,readOnly=TRUE)
except:
    print "ERROR: Unable to open the specified odb %s" % odbFile
    exit(0)

# Ask for analysis step to map displacements from
prompt3 = "\nName of step displacements will be mapped from:\n"
stepName = raw_input(prompt3)
if stepName not in odb.steps.keys():
    print "\nERROR: Step '%s' does not exist in %s\n"\
          "\tCheck for the case in the step name." % (stepName, odbFile)
    exit(0)

# Ask for frame number (load increment) to map displacements from
frame = askForInt("\nFrame number (increment) to map \
                displacements from:\n")
if len(odb.steps[stepName].frames) < frame:
    print "\nERROR: Frame %i does not exist in %s\n"\
          "\tCheck for the case in the .sta file." % (frame, odbFile)
    exit(0)

odb.close()
```

```
    # Input parameters to masterMapDisplacements.py script***
    os.system('abaqus cae noGUI=masterMapDisplacements.py -- '\
              +res+' '+localglobal+' '+prevInp+' '+fdbPrev+' '+odbFile+' '\
              +currInpFile+' '+currInpFile_full+' '+fdbCurr+' '\
              +stepName+' '+str(frame))

    # Pass arguments to CTDjobControlCall script
    prompt4 = "\nStart FEA with CTD job control?:\n"
    start = askYesNo(prompt4)

    if start == "Y":
        # Current deformed input file
        if localglobal == "Y":
            defInpFile = currInpFile_full
        else:
            defInpFile = currInpFile

        # Undeformed input file located in Undeformed sub-dir
        undefInpFile = str()
        tmp = currInpFile.split("/")
        fileName = tmp[-1]
        tmp[-1] = "Undeformed"
        for i in range(len(tmp)):
            undefInpFile+=tmp[i]+"/"
        undefInpFile+=fileName

        # .fdb file
        fdb = getFdb(currInpFile)

        print "Calling CTDjobControlCall.py"
        os.system('abaqus python CTDjobControlCall.py -- '\
                  +defInpFile+' '+undefInpFile+' '+fdb+' '+odbFile)

    else:
        print "Displacement mapping complete!"

    # Finally, move the abaqus.rpy file to the directory of the current mesh
    print "Moving the abaqus.rpy file..."
    currPath = str()
    tmp = currInpFile.split("/")
    for i in range(len(tmp)-1):
        currPath+=tmp[i]
        currPath+="/"
    currPath+="abaqus.rpy"
    os.rename("abaqus.rpy",currPath)
    os.remove("abaqus_acis.log")

'''
masterMapDisplacements.py

Executed by runMapDisplacements.py

'''
# *************************
import Vec3D
import ColTensor
```

```
import FullTensor
import MeshTools
import os, glob, sys, time
from abaqus import *
from abaqusConstants import *
import part
import assembly
import mesh
from sys import argv, exit
import odbAccess

# *************************
def changeDirectory(pathToFile):
    path = str()
    tmp = pathToFile.split("/")
    for i in range(len(tmp)-1):
        path+=tmp[i]
        path+="/"
    print path
    os.chdir(path)

# *************************
def createSubDirectory(filePath,subDirName):
    newdir = ''
    path = filePath.split("/")
    for i in range(len(path)-1):
        newdir+=path[i]+"/"
    newdir+=subDirName+"/"
    if not os.path.isdir(newdir):
        os.mkdir(newdir)
    return newdir

# *************************
def createMeshToolsObject(filePath,extension):
    base = str()
    tmp = filePath.split(".")
    tmp[-1] = ''
    for i in range(len(tmp)):
        base+=tmp[i]
    meshObj = MeshTools.MeshTools(base,extension)
    return meshObj

# *************************
def getGroupList(meshObj,listName,type):
    list = meshObj.GetGroupInfo(listName,type)
    if len(list) == 0:
        print "***No members found in group: %s ***" % listName
    return list

# *************************
def printTimeStamp(start_time):
    # Print time stamp
    end_time = time.time()
    mins = int((end_time-start_time)/60)
    secs = round((end_time-start_time)%60,3)
    print "\n************************************************"
```

```
    print "\nTotal time for mapping: %i:%g" % (mins,secs)
    print "\n***************************************************"

# ****************** GET INITIAL DISPLACEMENTS FROM OLD MESH ********************
# *******************************************************************************
def getInitialDisplacements(filePath):
    # Get initial displacements, if any, from the ## Displacement header of the
    # .model file of the previous mesh
    initial_disps = {}
    disp_file = str()
    buff = filePath.split('/')
    for i in range(len(buff)-1):
        disp_file+=buff[i]+'/'
    disp_file+='mapped_disp.txt'
    print disp_file
    # If the disp.txt file doesn't exist, continue, but inform analyst
    if not os.path.exists(disp_file):
        print "***Could not locate mapped_disp.txt from previous analysis!***"
        print "(Make sure the file is located with the previous mesh file)"
        return initial_disps

    file = open(disp_file,'r')
    buff = file.readline()
    while buff:
        data = buff.split()
        if data[0].upper()[0:4] == "DISP":
            buff = file.readline()
            data = buff.split()
            while buff:
                nid = int(data[0])
                tmp = []
                for i in range(3):
                    tmp.append(float(data[i+1]))
                initial_disps[nid] = tmp
                buff = file.readline()
                data = buff.split()
        buff = file.readline()

    file.close()
    print "***Retrieved initial displacements for %d nodes***" % len(initial_disps)
    return initial_disps

# *************** GET RELATIVE DISPLACEMENTS FROM PREVIOUS ANALYSIS **************
# *******************************************************************************
def getRelativeDisplacements(odbFile,stepName,frame,oldNodeList,localglobal):
    # Extract displacements at the critical step and increment from the previous
    # analysis
    relative_disps = {}     # Dictionary containing odb disps. for all nodes
    l_relative_disps = {}   # Dictionary containing odb disps. for local nodes
    g_relative_disps = {}   # Dictionary containing odb disps. for global nodes
    fullNodeList = []       # List of all nodes from odb-->indices correspond to
                            # indices in .odb node list
    g_odbIndices = []       # List of node indices from odb global nodes for
                            # referencing nodes by index rather than node ID
                            # (if local-global analysis)
```

```
try:
    odb = odbAccess.openOdb(path=odbFile,readOnly=TRUE)
except:
    print "ERROR: Unable to open %s" % odbFile
    exit(0)

# Get step number corresponding to stepName
for i in range(len(odb.steps.keys())):
    if stepName in odb.steps.keys()[i]:
        stepNum = i+1

# Create list of all nodes in model
# NOTE: nodeLabels do not necessarily correspond to fieldObject index
# especially for local/global join
for instance in odb.rootAssembly.instances.keys():
    for i in range(len(odb.rootAssembly.instances[instance].nodes)):
        fullNodeList.append(odb.rootAssembly.instances[instance].nodes[i].label)

fieldObject = odb.steps[stepName].frames[frame].fieldOutputs['U']

# If the model is local-global, then separate odb displacements into
# local and global dictionaries, to be handled separately
if localglobal == "Y":
    for nid in fullNodeList:
        index = fullNodeList.index(nid)
        disps = fieldObject.values[index].data
        if nid in oldNodeList:
            l_relative_disps[nid] = disps
        else:
            g_relative_disps[nid] = disps
            g_odbIndices.append(index)

# Otherwise include all odb disps into a single dictionary
elif localglobal == "N":
    for nid in fullNodeList:
        index = fullNodeList.index(nid)
        disps = fieldObject.values[index].data
        relative_disps[nid] = disps

else:
    print "Unknown response to local/global command prompt"
    exit(0)

odb.close()

if len(relative_disps) > 0:
    print "***Retrieved odb disps for %d nodes" % len(relative_disps)
elif len(l_relative_disps) > 0 and len(g_relative_disps)>0:
    print "***Retrieved odb disps for %d local nodes and %d global nodes" % \
          (len(l_relative_disps), len(g_relative_disps))
else:
    print "***No relative displacements retrieved from odb. \
           Exiting map script."
    exit(0)
return stepNum,fullNodeList,relative_disps,l_relative_disps,\
       g_relative_disps,g_odbIndices
```

```
# ********************** COMPUTE ABSOLUTE DISPLACEMENTS ************************
# ******************************************************************************
def computeAbsoluteDisplacements(initial_disps,relative_disps,nodeList,prevMesh):
    # Compute the absolute nodal displacements for the old mesh at the critical
    # load step/increment of interest
    abs_disps = {}

    # Quick check that length of lists are the same
    if len(relative_disps) != len(nodeList):
        print "Size of lists disagree in computeAbsoluteDisplacements function!"
        exit(0)

    if len(initial_disps) != len(relative_disps):
        if len(initial_disps) == 0:
            print "***No initial displacements found in previous mesh***"
            print "***Check that previous mesh did not have initial \
                  displacements***"
            print "***(Mapping will only include relative displacements from the odb)"
            # Set initial displacements to zero
            for node in nodeList:
                initial_disps[node] = [0,0,0]
        else:
            print "***Initial and relative displacements found for different \
                   number of nodes!"
            print "***Check model and odb files from previous mesh for the discrepancy!"
            exit(0)

    for node in nodeList:
        d = []
        for i in range(0,3):
            try:
                val = float(initial_disps[node][i])+float(relative_disps[node][i])
                d.append(val)
            except:
                print "***Error computing absolute displacements for node %d!" % node
                print "***Check that initial and relative disp vals exist for the node"
                exit(0)
        abs_disps[node] = d

    return abs_disps

# ****************** MAP DISPLACEMENTS TO NEW UNDEFORMED MESH ********************
# ******************************************************************************
def mapDisplacementsToUndeformedMesh(mesh1,abs_disps,mesh2):

    # ************** mapDisplacementsToUndeformedMesh SUB-FUNCTIONS **************

    '''
    # *************************
    def parseModelFileForGroup(file,keyword):
        list = []
        fd = open(file,"r")

        buff = fd.readline()
        while buff:
```

```
        data = buff.split()
        if len(data) == 5:      # Implying group sub-header line
            if data[4].upper() == keyword:
                print "Found %s in %s" % (keyword,file)
                buff = fd.readline()
                data = buff.split()
                while buff and data[0] != "#":
                    for i in range(len(data)):
                        list.append(int(data[i]))
                    buff = fd.readline()
                    data = buff.split()
                break
        buff = fd.readline()

    fd.close()

    return list
'''

# *************************
def makeElemSet(mesh,nodeList):
    # Create a set of elements that contain the nodes in nodeList
    setName = set()
    for nid in nodeList:
        elist = mesh.GetAdjacentElems(nid)
        for eid in elist:
            setName.add(eid)

    if len(setName) == 0:
        print "***ERROR: None found for the case %s" % set

    return setName

# *************************
def setPointInsideTol(mesh,mainNodeList,mateNodeList):
    # Set point inside tolerance to 1/1000 of the smallest crack
    # face edge length
    min_length = mesh.GetMaxDimension()    # Initialize a minimum length

    for nid in mainNodeList:
        elengths = mesh.GetAdjacentSurfEdgeLengths(nid) # function returns
        min_length = min(min_length,min(elengths))      # error if nid is
                                                        # not surface node
    for nid in mateNodeList:
        elengths = mesh.GetAdjacentSurfEdgeLengths(nid)
        min_length = min(min_length,min(elengths))

    print "min_length"
    print min_length
    initial_tol = min_length/1000.0
    mesh.SetPointInsideTolerance(initial_tol)
    print "***Min edge length is: %f" % min_length
    print "***Tolerance is: %f" % initial_tol
    return initial_tol

# *************************
```

```
def mapCrackFaceNodeDisps():

    print "***WARNING: main- and mate-side choices are assumed to be\n \
       consistent in both crack models. "
    print "***Check fdb files to check validity of this assumption!"

    # The mesh objects for the previous and current meshes should contain
    # the necessary main/mate side node groups (excluding crack front nodes)
    # for the cases with and without a crack front template.

    # First collect the main/mate side node sets
    try:
        main_side_nodes1 = getGroupList(mesh1,"main_side_nodes","node")
        mate_side_nodes1 = getGroupList(mesh1,"mate_side_nodes","node")
    except:
        try:
            main_side_nodes1 = getGroupList(mesh1,"all_main_side_nodes","node")
            mate_side_nodes1 = getGroupList(mesh1,"all_mate_side_nodes","node")
        except:
            print "***No main/mate side nodes found for previous mesh!***"
            exit(0)

    try:
        main_side_nodes2 = getGroupList(mesh2,"main_side_nodes","node")
        mate_side_nodes2 = getGroupList(mesh2,"mate_side_nodes","node")
    except:
        try:
            main_side_nodes2 = getGroupList(mesh2,"all_main_side_nodes","node")
            mate_side_nodes2 = getGroupList(mesh2,"all_mate_side_nodes","node")
        except:
            print "***No main side nodes found for current mesh!***"
            exit(0)
    #print len(main_side_nodes1)
    #print len(mate_side_nodes1)
    #print len(main_side_nodes2)
    #print len(mate_side_nodes2)

    # Next make sets of main/mate crack face elements for the previous mesh
    print "***Collecting main/mate side elems from previous mesh..."
    main_side_elems1 = makeElemSet(mesh1,main_side_nodes1)
    mate_side_elems1 = makeElemSet(mesh1,mate_side_nodes1)

    # Set an initial PointInside tolerance based on the previous mesh
    # This is the tolerance for checking existence of a point from
    # the new mesh in the old mesh.
    initial_tol = setPointInsideTol(mesh1,main_side_nodes1,mate_side_nodes1)

    # Map displacements onto the crack face nodes of the current mesh
    crkface_node_disps = {}

    # First map displacements to main_side nodes of current mesh
    for nid in main_side_nodes2:
       # Reset tolerance in case it was previously increased.
        tol = initial_tol
        mesh1.SetPointInsideTolerance(tol)
        elist = []
```

```
info = mesh2.GetNodeInfo(nid)
pt = info[0]
status,dist = mesh1.IsPointOutsideMesh(pt)

# Make sure pt. is within the bounding box of the mesh
if status == -3:
    print "***Node id: %i " % nid
    print "***Nodal coords: "
    print pt
    print "***Status (old eid): %i" % status
    raise ValueError, "Query point must be inside mesh"

# If the point in the new mesh exists in a crackface element of
# the old mesh, then interpolate displacements only from the containing
# element (i.e. do not interpolate over surrounding elements in high
# gradient region!)

# Associate node with elements in previous mesh. If empty range tree
# search (which happens numerically on rare instances), increase tol
# and try again.
while len(elist) == 0:
    try:
        elist = mesh1.GetElemsAtPoint(pt)
    except:
        print "Tol increased for nid %g due to empty range tree search"\
              % nid
        new_tol = tol*10.0
        mesh1.SetPointInsideTolerance(new_tol)
        tol = new_tol

main_eid = -1
mate_flag = 0

# Loop through associated elems and look for containing main_side elem
# in previous mesh (won't be the case for nodes in new surface area).
for eid in elist:
    if eid in main_side_elems1:
        main_eid = eid
        break
    if eid in mate_side_elems1:
        mate_flag = 1

# If a mate_side element was associated, but not a main_side element,
# then enter while loop to increase query tolerance.
while main_eid == -1 and mate_flag == 1:
    new_tol = tol*10.0
    print "increasing tolerance to %f for nid %g" % (new_tol,nid)
    mesh1.SetPointInsideTolerance(new_tol)
    elist = mesh1.GetElemsAtPoint(pt)
    for eid in elist:
        if eid in main_side_elems1:
            main_eid = eid
            break
    tol = new_tol
```

```
    disp = mesh1.GetPtDisp(pt,main_eid)
    crkface_node_disps[nid] = disp
print "looped through all main_side_nodes2"

# Next map displacements to mate_side nodes of current mesh
# (just like above)
for nid in mate_side_nodes2:
    # Reset tolerance in case it was previously increased.
    tol = initial_tol
    mesh1.SetPointInsideTolerance(tol)
    elist = []

    info = mesh2.GetNodeInfo(nid)
    pt = info[0]
    status,dist = mesh1.IsPointOutsideMesh(pt)

    # Make sure node clearly exists inside previous mesh
    if status == -3:
        print "***Node id: %i " % nid
        print "***Nodal coords: "
        print pt
        print "***Status (old eid): %i" % status
        raise ValueError, "Query point must be inside mesh"

    # Associate node with elements in previous mesh.  If empty range tree
    # search (which happens numerically on rare instances), increase tol
    # and try again.
    while len(elist) == 0:
        try:
            elist = mesh1.GetElemsAtPoint(pt)
        except:
            print "Tol increased for nid %g due to empty range tree search"\
                  % nid
            new_tol = tol*10.0
            mesh1.SetPointInsideTolerance(new_tol)
            tol = new_tol

    mate_eid = -1
    main_flag = 0

    # Loop through associated elems and look for containing mate_side elem
    # in previous mesh (won't be the case for nodes in new surface area).
    for eid in elist:
        if eid in mate_side_elems1:
            mate_eid = eid
            break
        if eid in main_side_elems1:
            main_flag = 1

    # If a main_side element was associated, but not a mate_side element,
    # then enter while loop to increase query tolerance.
    while mate_eid == -1 and main_flag == 1:
        new_tol = tol*10.0
        print "increasing tolerance to %f for nid %g" % (new_tol,nid)
        mesh1.SetPointInsideTolerance(new_tol)
        elist = mesh1.GetElemsAtPoint(pt)
```

```
            for eid in elist:
                if eid in mate_side_elems1:
                    mate_eid = eid
                    break
            tol = new_tol

        disp = mesh1.GetPtDisp(pt,mate_eid) # No averaging over adjacent elems
        crkface_node_disps[nid] = disp
    print "looped through all mate_side_nodes2"
    return crkface_node_disps, initial_tol


# *************** mapDisplacementsToUndeformedMesh MAIN ******************
mapped_disps = {}

# Note: Displacements currently associated with mesh1 are initial disps,
# which were mapped during previous run of this script.
for nid in abs_disps:
    coord = Vec3D.Vec3D(abs_disps[nid][0],abs_disps[nid][1],abs_disps[nid][2])
    mesh1.UpdateNodalDisps(nid,coord)

# Map node displacements at the crack face nodes first
# (this routine interpolates displacements for points only within the
# containing crackface element and doesn't average over surrounding elements)
(crkface_node_disps, initial_tol) = mapCrackFaceNodeDisps()

# Now map node displacements for all nodes
mesh2nodes = mesh2.GetNodeList()
mesh2nodes.sort()

# Note that pt_query tolerance may be increased if a node is not deemed
# inside the previous mesh; if tolerance is increased to the min. edge
# length, an error is raised and the script is terminated.
for nid in mesh2nodes:
    tol = initial_tol
    if nid in crkface_node_disps:
        mapped_disps[nid] = crkface_node_disps[nid]
    else:
        info = mesh2.GetNodeInfo(nid)
        pt = info[0]
        status,dist = mesh1.IsPointOutsideMesh(pt)

        # Status:   -2 --> query_pt is clearly inside previous mesh
        #           -1 --> query_pt is close to surface, but not clearly inside
        #           -3 --> query_pt is not inside previous mesh
        while status != -2:
            if status == -1:
                new_tol = tol*10.0
                # (Uncomment these lines to put a limit on search tol)
                #if new_tol > initial_tol*1000.0:
                    #print "MeshTools tolerance exceeded smallest edge length!"
                    #print "Node id %i of currMesh is within %f of prevMesh \
                           surface" % (nid,dist)
                    #exit(0)
                mesh1.SetPointInsideTolerance(new_tol)
                status,dist = mesh1.IsPointOutsideMesh(pt)
```

```
                    tol = new_tol
                    print "Tolerance has been reset to %f due to nid %i of currMesh"\
                           % (tol,nid)
                else:
                    txt = "Query point must be inside mesh"
                    raise ValueError, txt
            # If the tolerance was increased for nid, reset to original
            mesh1.SetPointInsideTolerance(initial_tol)
            # Get displacement by interpolating displacements from prevMesh
            # at point of interest
            mapped_disps[nid] = mesh1.GetPtDisp(pt)

    return mapped_disps


# ****************** APPEND MAPPED DISP RESULTS TO .MODEL FILE ********************
# *********************************************************************************
def writeMappedDispsToFile(mapped_disps,currInpFile,stepName,frame):
    # Write the mapped displacement info to a .txt file

    # Create mapped_disp.txt in same directory as current input file
    changeDirectory(currInpFile)
    file = open("mapped_disp.txt","w")

    file.write("Displacements mapped from: %s Increment: %i\n" % (stepName,frame))
    for nid in mapped_disps:
        file.write("%i %.16e %.16e %.16e\n" % (nid,mapped_disps[nid][0],\
                                               mapped_disps[nid][1],\
                                               mapped_disps[nid][2]))
    file.close()

# ********************** COMPUTE DEFORMED COORDINATES *************************
# *************** For Global Region (if applicable) **************
def computeGlobalCoords(odbFile,stepName,frame,g_relative_disps,g_odbIndices):
    # Create dictionary of final coordinates for global nodes in previous mesh
    # g_new_coords = {nid: (x,y,z)}

    print "Computing absolute disps for global region"
    g_new_coords = {}

    if len(g_relative_disps) != len(g_odbIndices):
        print "Length of global node index list and rel_disp dict inconsistent!"
        eixt(0)

    try:
        odb = odbAccess.openOdb(path=odbFile,readOnly=TRUE)
    except:
        print "ERROR: Unable to open %s" % odbFile
        exit(0)

    # Initiate a counter for g_new_coords dict
    cnt = 1

    # Simply add relative disps. to node coords to get deformed coords
    # for global region of previous mesh (no inverse isoparametric
    # mapping involved)
```

```
    for instance in odb.rootAssembly.instances.keys():
        instance = odb.rootAssembly.instances[instance]
        for index in g_odbIndices:
            tmp = []
            node = instance.nodes[index]
            nid = node.label
            for i in range(0,3):
                val = float(g_relative_disps[nid][i])+float(node.coordinates[i])
                tmp.append(val)

            g_new_coords[cnt] = tmp
            cnt+=1
    odb.close()

    return g_new_coords

# *************** For Local or Entire Region (whichever applicable) **************
def computeDeformedCoords(currInpFile,mapped_disps):

    # Change directories to location of current .inp file if not there already
    changeDirectory(currInpFile)

    undef_coords = {}
    def_coords = {}
    scale = 1

    # Parse the original .inp for the current mesh to get the undeformed coords.
    file = open(currInpFile,"r")
    line = file.readline()
    while line:
        if line[0:5].upper() == "*NODE":
            line = file.readline()

            while line[0:1] != "*":
                data = line.split(",")
                nid = int(data[0])
                undef_coords[nid] = [float(data[1]),float(data[2]),float(data[3])]
                line = file.readline()
            break
        line = file.readline()

    file.close()

    # Do a quick check that min/max node id's are consistent
    if max(undef_coords) != max(mapped_disps) or \
        min(undef_coords) != min(mapped_disps):
        print "***Node numbering is inconsistent between .inp file and \
               mapped_disps dict!\n"
        exit(0)

    # Compute deformed nodal coordinates
    for nid in undef_coords:
        x_new = undef_coords[nid][0]+mapped_disps[nid][0]*scale
        y_new = undef_coords[nid][1]+mapped_disps[nid][1]*scale
        z_new = undef_coords[nid][2]+mapped_disps[nid][2]*scale
        def_coords[nid] = [x_new,y_new,z_new]
```

```
    return def_coords

# ************************* WRITE DEFORMED MESH (.inp) *************************
# ******************************************************************************
def writeDeformedInpFile(def_coords,g_def_coords,currInpFile,stepNum,frame,res):

    # *************** Scan input file for BC nodesets SUB-FUNCTION ***************
    def getBCnodesets(inpFile):
        # Scan for BC nodesets
        BCnodes = []
        ifile = open(inpFile,"r")
        line = ifile.readline()

        while line[0:6].upper() != "*BOUND":
            line = ifile.readline()

        while line:
            line = ifile.readline()
            while line[0:1] != "*":
                nset = line.split(",")[0]
                if nset not in BCnodes:
                    BCnodes.append(nset)
                line = ifile.readline()

            line = ifile.readline()
            while line and line[0:6].upper() != "*BOUND":
                line = ifile.readline()

        ifile.close()
        return BCnodes

    # ******************** writeDeformedInputFile SUB-FUNCTION ********************
    def writeFile(copyFile,currInpFile,def_coords,g_def_coords,stepNum,frame,\
                  BCnodes,res):
        ifile = open(copyFile,"r")      # Read the original file for copying
        ofile = open(currInpFile,"w")   # Overwrite the original as deformed

        print "***Writing deformed .inp file...\n"
        line = ifile.readline()
        while line[0:5].upper() != "*NODE":
            ofile.write(line)
            line = ifile.readline()

        print "len of def_coords:"
        print len(def_coords)
        print "max. val in def_coords:"
        print max(def_coords)
        print "len of g_def_coords:"
        print len(g_def_coords)

        # Write new, deformed nodal information
        ofile.write("*Node\n")
        for nid in def_coords:
            ofile.write(str(nid)+", "+str(def_coords[nid][0])+", "+\
                                        str(def_coords[nid][1])+", "+\
```

```
                                    str(def_coords[nid][2])+"\n")

# If local-global analysis, insert the deformed node coords for
# global region
line = ifile.readline()
if len(g_def_coords) > 0:
    g_currNodeList = []         # A list of global nids to check against
    offset = max(def_coords)    # The max. nid of the local region
    nid = line.split(",")[0]
    while nid != offset:
        line = ifile.readline()
        nid = int(line.split(",")[0])

    # After reaching offset, start writing global info
    for i in range(1,len(g_def_coords)+1):
        line = ifile.readline()
        g_nid = line.split(",")[0]
        g_currNodeList.append(g_nid)
        ofile.write(str(g_nid)+", "+str(g_def_coords[i][0])+", "+\
                    str(g_def_coords[i][1])+", "+\
                    str(g_def_coords[i][2])+"\n")

    # Check that the number of global nodes in previous and
    # current models is the same
    if len(g_def_coords) != len(g_currNodeList):
        print "The number of global nodes do not correspond \
               between old and new mesh models"
        print "Check script!"
        exit(0)

# Now copy the rest of the file data
line = ifile.readline()
while line[0:8].upper() != "*ELEMENT":
    line = ifile.readline()

ofile.write(line)
line = ifile.readline()
while line[0:5].upper() != "*STEP":
    ofile.write(line)
    line = ifile.readline()

# Write map step before first step
if res == "Y":  # Step number includes the Equilibrate step \
                # from first analysis
    newline = "*MAP SOLUTION,STEP=%d,INC=%d,UNBALANCED STRESS=RAMP\n**\n" %\
    (stepNum+1,frame)
else:
    newline = "*MAP SOLUTION,STEP=%d,INC=%d,UNBALANCED STRESS=RAMP\n**\n" %\
    (stepNum,frame)
ofile.write(newline)
newline = "** Name: Constrained Type: Displacement/Rotation\n**\n"
ofile.write(newline)
newline = "**\n*Step, name=Equilibrate\n*Static\n0.25, 1., 1e-05, 0.25\n"
ofile.write(newline)
newline = "**\n*Restart, write, frequency=1, overlay\n"
ofile.write(newline)
```

```
        newline = "*File Format, ASCII\n"
        ofile.write(newline)
        newline = "*Node File, Frequency=9999\n"
        ofile.write(newline)
        newline = "U,\n"
        ofile.write(newline)
        newline = "**\n** BOUNDARY CONDITIONS\n**\n"
        ofile.write(newline)
        for i in range(len(BCnodes)):
            newline = "**Name: Fixed_%d Type: Displacement/Rotation\n" % i
            ofile.write(newline)
            newline = "*Boundary\n%s, 1, 1\n%s, 2, 2\n%s, 3, 3\n" % \
            (BCnodes[i],BCnodes[i],BCnodes[i])
            ofile.write(newline)

        newline = "**\n*End Step\n** ----------------------------------------\n"
        ofile.write(newline)

        # Write the rest of the file, adding "op=NEW" to boundary
        # condition line, which deactivates BC's from the mapping step.
        while line:
            if line[0:6].upper() == "*BOUND":
                line = "*Boundary, op=NEW\n"
                ofile.write(line)
                line = ifile.readline()
                continue
            ofile.write(line)
            line = ifile.readline()

        ifile.close()
        ofile.close()

    # ********************** writeDeformedInputFile MAIN *************************
    # Move undeformed input file to new directory
    dir = createSubDirectory(currInpFile,"Undeformed")
    fname = currInpFile.split("/")[-1].split(".")
    copyFile = dir+fname[-2]+"_Undeformed."+fname[-1]
    if not os.path.exists(copyFile):
        os.rename(currInpFile,copyFile)

    # Prescan the original input to get nodes(or nodesets) with
    # prescribed BC's before writing the deformed input file.
    BCnodes = getBCnodesets(copyFile)

    # Write the deformed input file
    writeFile(copyFile,currInpFile,def_coords,g_def_coords,stepNum,frame,BCnodes,res)

# ********************************** MAIN ***************************************
# *******************************************************************************

if __name__ == "__main__":

    print __doc__

    path = os.getcwd
    res = sys.argv[-10]               # "Y" if mapping from a previous restart file
```

```
localglobal = sys.argv[-9]
prevInp = sys.argv[-8]            # Previous .inp file (local only, if applicable)
fdbPrev = sys.argv[-7]
odbFile = sys.argv[-6]
currInpFile = sys.argv[-5]         # local region only if local-global analysis
currInpFile_full = sys.argv[-4]    # *_full.inp; Empty if not local-global analysis
fdbCurr = sys.argv[-3]
stepName = sys.argv[-2]
frame = int(sys.argv[-1])

start_time = time.time()

# Check for file with initial displacements; get them.
initial_disps = getInitialDisplacements(fdbPrev)

# Create MeshTools object of the current and previous .inp files
prevMesh = createMeshToolsObject(prevInp,"INP")
currMesh = createMeshToolsObject(currInpFile,"INP")

# Get list of nodes from previous mesh
oldNodeList = prevMesh.GetNodeList()

# Function returns stepNum corresponding to stepName,
# list of all nodes in previous odb,
# dictionary of odb disps for all nodes **if not local-global**
# dictionaries of odb disps for local and global regions **if local-global**
# (empty dicts if inapplicable)
(stepNum,prevFullNodeList,relative_disps,l_relative_disps,\
g_relative_disps,g_odbIndices) = getRelativeDisplacements(odbFile,\
                                 stepName,frame,oldNodeList,localglobal)

if localglobal == "N":
    # All arguments --> entire mesh
    abs_disps = computeAbsoluteDisplacements(initial_disps,\
                relative_disps,prevFullNodeList,prevMesh)
elif localglobal == "Y":
    # All arguments --> local region only
    abs_disps = computeAbsoluteDisplacements(initial_disps,l_relative_disps,\
                                             oldNodeList,prevMesh)

# Map displacements (only on local region, if applicable)
mapped_disps = mapDisplacementsToUndeformedMesh(prevMesh,abs_disps,currMesh)
writeMappedDispsToFile(mapped_disps,currInpFile,stepName,frame)

# Compute deformed coordinates
def_coords = computeDeformedCoords(currInpFile,mapped_disps)
if localglobal == "Y":
    # Simply add global node disps to node coords of previous odb
    g_def_coords = computeGlobalCoords(odbFile,stepName,frame,\
                                       g_relative_disps,g_odbIndices)
elif localglobal == "N":
    g_def_coords = []

# Write the deformed input file
if localglobal == "Y":
    # Write deformed *_full.inp file; move undeformed input
```

```
        # files to *_Undeformed folder
        writeDeformedInpFile(def_coords,g_def_coords,currInpFile_full,\
                             stepNum,frame,res)
        new_path = str()
        tmp = currInpFile.split("/")
        fileName = tmp[-1]
        tmp[-1] = "Undeformed"
        for i in range(len(tmp)):
            new_path+=tmp[i]+"/"
        new_path+=fileName
        os.rename(currInpFile,new_path)
    elif localglobal == "N":
        # Write deformed *.inp file
        writeDeformedInpFile(def_coords,g_def_coords,currInpFile,\
                             stepNum,frame,res)

    printTimeStamp(start_time)

    #exit(0)
```

C. Details from Integrally-stiffened Panel Test Program

Researchers at Alcoa Technical Center fabricated several ISPs as part of a test program in 1998. The purpose of the test program was to compare fatigue crack growth and residual strength among ISPs machined from either of two, lower wing skin, aluminum alloys—AA 2024-T351 or C433-T39. Test information has been utilized by others for analysis purposes [15, 57, 59]. Alcoa Technical Center has provided the current authors with test details, which are provided here for completeness.

Fatigue crack growth and residual strength tests were conducted at Alcoa Technical Center for four ISPs, two machined from AA 2024-T351 and two machined from C433-T39. Dimensions of all panels are shown in Fig. 16. In each panel, an initial two-bay saw cut of length $2a_{saw} \approx 2.54$ cm was made at mid-height to completely sever the middle stiffener. The initial cut was then pre-cracked to length $2a_0 \approx 5.08$ cm by applying uniaxial cyclic load in the y-direction, where $P_{min} = 31$ kN and $P_{max} = 311$ kN. Subsequently, a modified transport wing standard (TWIST) loading spectrum [60], shown in Table 6, was applied in the y-direction to propagate the fatigue crack. The applied spectrum had a mean flight stress of $S_{mf} = 68.9$ MPa, truncated to level V, and included a ground-air-ground (GAG) cycle with a reduced ground level stress of $S_{ground} = -34.5$ MPa. Taxi loads were neglected. Depending on the panel, the fatigue crack was propagated until both crack fronts were 2.54 cm short of reaching the intact stiffeners ($2a_i \approx 24.1$ cm) or until both crack fronts just

entered the intact stiffeners ($2a_i \approx 30.0$ cm). Subsequently, each panel was loaded monotonically in uniaxial tension until failure occurred by unstable crack growth. A test matrix of the four panels and corresponding residual strengths is shown in Table 7. The ISP simulated in this work follows panels 5 and 5A.

Table 6 Standard TWIST spectrum scaled to mean flight stress $S_{mf} = 68.9$ MPa with variable amplitude loads S_a (left). Modified spectrum (right) applied uniaxially to the Alcoa ISP prior to residual strength testing. Courtesy Alcoa Technical Center.

Standard TWIST profile scaled to S_{mf} = 68.9 Mpa

Block Level	Load Level ($S_{mf} \pm S_a$), MPa	Number of cycles per 4000 flights
I	68.9 ± 110.3	1
II	68.9 ± 103.4	2
III	68.9 ± 89.6	5
IV	68.9 ± 79.3	18
V	68.9 ± 68.6	52
VI	68.9 ± 57.9	152
VII	68.9 ± 47.2	800
VIII	68.9 ± 36.5	4170
IX	68.9 ± 25.9	34800
X	68.9 ± 15.3	358665
GAG	-34.5 (last peak per flight)	4000

Modified TWIST profile applied to ISPs

Block Level	Load Level ($S_{mf} \pm S_a$), MPa	Number of cycles per 4000 flights
I	--	truncate loads to level V
II	--	
III	--	
IV	--	
V	68.9 ± 68.6	78
VI	68.9 ± 57.9	152
VII	68.9 ± 47.2	800
VIII	68.9 ± 36.5	4170
IX	68.9 ± 25.9	34800
X	68.9 ± 15.3	358665
GAG	-34.5 (last peak per flight)	4000

Table 7 Description of all ISPs and corresponding residual strengths from Alcoa test program. Courtesy Alcoa Technical Center.

Panel ID	Material	Crack length, $2a_i$, just prior to residual strength test (cm)	Residual strength (kN)
3	2024-T351	24.1 cm	1225
4	2024-T352	30.0 cm	1015
5A	C433-T39	24.1 cm	1660
5	C433-T39	24.1 cm	1385
6	C433-T40	30.0 cm	1295

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01-01-2011	Technical Memorandum	

4. TITLE AND SUBTITLE

Surrogate Modeling of High-Fidelity Fracture Simulations for Real-Time Residual Strength Predictions

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Spear, Ashley D.; Priest, Amanda R.; Veilleux, Michael G.; Ingraffea, Anthony R.; and Hochhalter, Jacob D.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

284848.02.05.07.02

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER

L-19954

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)

NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

NASA/TM-2011-216879

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Category 26
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES

14. ABSTRACT

A surrogate model methodology is described for predicting, during flight, the residual strength of aircraft structures that sustain discrete-source damage. Starting with design of experiment, an artifcial neural network is developed that takes as input discretesource damage parameters and outputs a prediction of the structural residual strength. Target residual strength values used to train the artificial neural network are derived from 3D finite element-based fracture simulations. Two ductile fracture simulations are presented to show that crack growth and residual strength are determined more accurately in discrete-source damage cases by using an elastic-plastic fracture framework rather than a linear-elastic fracture mechanics-based method. Improving accuracy of the residual strength training data does, in turn, improve accuracy of the surrogate model. When combined, the surrogate model methodology and high fidelity fracture simulation framework provide useful tools for adaptive flight technology.

15. SUBJECT TERMS

Residual Strength; Metals; Metallic Materials; Genetic Algorithms; Fracture; Adaptive Flight Technology

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	87	(443) 757-5802