czmtestkit

Release v1.1.0

Nanditha Mudunuru, Miguel Bessa, Albert Turon

Sep 18, 2022

CONTENTS

CHAPTER

ONE

COHESIVE ZONE MODEL TEST KIT

The python package `czmtestkit` works parallel to `Abaqus/CAE` mainly to test user element subroutines of cohesive zone models. Additionally, the package facilitates the implementaion of the `Mode Partitioning Method` for mixed-mode characterization of interfaces proposed by Moreira et al., (2020).

1.1 Prerequisites

Ensure that all the following requirements are satisfied.

1.1.1 Requirements:

1. Abaqus/CAE is available and can be opened with the following command from the command line. `abaqus cae`
2. Fortran compiler is linked to abaqus. If not, follow instructions by Victor Crespo-Cuevas (2021).
3. Python and pip have been installed and can be run from the command line. If unsure, follow instructions in https://packaging.python.org/en/latest/tutorials/installing-packages/

1.2 Install

Run the following command in a command window.

```
python -m pip install czmtestkit
```

In case an issue arises, try the following command.

```
pip install czmtestkit
```

Use the following command to upgrade the package.

```
python -m pip install --upgrade czmtestkit
```

1.3 Software description

1.3.1 v1.1.0

Update in type and functionality of 'nPoints' in doe_data for py_modules.run_sim function.

1.3.2 v1.0.0

Overview of current functionality of the package:

1) Generate models and run finite element analysis of standardized tests for mixed-mode fracture characterization of interfaces using Abaqus/CAE. The asymmetric double cantilever beam (ADCB) and asymmetric single leg bending (ASLB) are currently available. The models can be implemented with cohesive zone elements at the interface.
2) User element subroutines can be implemented to model the cohesive elements. Additionally, abaqus implementation of the cohesive zone using quadratic damage initiation and energy based linear damage evolution are available as a bench mark. Both BK criteria and power-law energy criteria are available.
3) Sequentially run multiple test from a design of experiments (doe). However, running on cluster or parallel computing is not possible yet.
4) Fetch history output from `.odb` files. Further post process the extracted data.
5) Read data from converged increments in `.msg` files.
6) Analytical models for the ADCB, ASLB and end notch flexure tests are also available and can be used to find fracture resistance curves from force-displacement curves or to predict force-displacement curves given the specimen dimensions and fracture properties.

Examples are avaiable in the *documentation* and the package is on *PyPI*. The code documentation for developers will be made available soon.

Contributions are welcome. To ensure a safe environment, all contributors are expected to adhere to the contributor guidelines and the contributor covenant code of conduct.

1.4 References

[1] R. Moreira, M. de Moura, F. Silva, F. Ramírez, and J. Rodrigues. Mixed-mode i + ii fracture characterisation of composite bonded joints. Journal of Adhesion Science and Technology, 34(13):1385–1398, 2020.

[2] Victor Crespo-Cuevas (2021) Linking ABAQUS 2019/2020 and Intel oneAPI Base Toolkit (FORTRAN Compiler). DOI:10.13140/RG.2.2.21568.05126

1.5 Cite

If you use this software, please cite it as below.

APA

```
Mudunuru, N., Bessa, M. A., & Turon Travesa, A. Python package to test the mixed-mode␣
↪response of user element subroutines of cohesive zone elements for implementation with␣
↪Abaqus/CAE (Version 1.0.0) [Computer software]. https://doi.org/10.4121/19410146
```

bibtex

```
@software{Mudunuru_Python_package_to,
author = {Mudunuru, Nanditha and Bessa, Miguel A. and Turon Travesa, Albert},
doi = {10.4121/19410146},
license = {GNU GENERAL PUBLIC LICENSE, Version 3, 29 June 2007},
title = {{Python package to test the mixed-mode response of user element subroutines of␣
↪cohesive zone elements for implementation with Abaqus/CAE}},
url = {https://github.com/NMudunuru/CzmTestKit.git},
version = {1.0.0}
}
```

1.6 License

Authors: Nanditha Mudunuru | Miguel Bessa | Albert Turon

CHAPTER

TWO

EXAMPLES

2.1 Mode Partitioning Method Implemented with ADCB specimen

This notebook is an example implemention of the mode partitioning method with BK criteria defined in section 3.1 on `Mode Partitioning Method`, presented in the master thesis title `Finite Element Model For Interfaces In Compatibilized Polymer Blends`.

Outline

1) Create design of experiments. ✓
2) Run simulations. ✓
3) Read odb data. ✓
4) Generate R curves. ✓
5) Extract Fracture toughness samples from R curves. ✓
6) Predict mode ratio based on the sampled fracture toughness and assumed mixed-mode energy criteria. ✓

2.1.1 Preprocessing

Import required modules.

[5]:
```
# imports
import json
import numpy as np
from IPython import display
from scipy.stats import qmc
from matplotlib import pyplot as plt
%matplotlib inline
```

Create a dictionary of the variables required to generate and run the Abaqus/CAE model that are common for all the tests.

[2]:
```
TestSet = 'ADCB_AbqImp' # Folder name for the set of experiments

# Constants for the experiment
fixed = {'JobID':'ADCB_AbqImp',
    'Length':100,
    'Width':25,
    'tTop':1.5,
```

(continues on next page)

(continued from previous page)

```
    'tBot':5.1,
    'tCz':0.2,
    'Crack':60,
    'DensityBulkTop':1.8e-9,
    'ETop':(109000.0, 8819.0, 8819.0, 0.34, 0.34, 0.38, 4315.0, 4315.0, 3200.0),
    'DensityBulkBot':1.8e-9,
    'EBot':(109000.0, 8819.0, 8819.0, 0.34, 0.34, 0.38, 4315.0, 4315.0, 3200.0),
    'DensityCz':1.8e-9,
    'StiffnessCz':1250,
    'GcNormal':0.42,
    'GcShear':4.2,
    'gFailureNormal':0.2,
    'gFailureShear':0.2,
    'MeshCrack':3,
    'MeshX':1,
    'MeshZ':0.6,
    'Displacement':20,
    'nCpu':1,
    'nGpu':0,
    'userSub':{'type':'None', 'path':'C:\\Users\\nandi\\Documents\\Abaqus Work Directory\
↪\MixedMode\\FDF.for', 'intProp':[]},
    'submit':True}
```

Further, define the variables that have to be changed for each experiment.

```
[ ]: n_pts = 3 # number of samples requied from the sobol sequence
     n_variables = 1 # number of variables
     l_bounds = [2.0] # lower limits for sampling
     u_bounds = [3.0] # upper limits for sampling

     sampler = qmc.Sobol(d=n_variables, seed=1896) # setting up the sequence generator
     sample = sampler.random(n_pts) # sampling the points
     pts = qmc.scale(sample, l_bounds, u_bounds).tolist() # scaling them using the sample␣
     ↪bounds

     # Defining a dictionary with list of values for the variable.
     doe = {'bkPower':pts}
     doe['nPoints'] = [x for x in range(len(pts))]
```

2.1.2 Running simulations

The `run_sim` function iterates through each point in the design of experiments (doe) and creates an instance of the input data by assigning the values for the variables from their corresponding lists in the `doe` dict along with all the fixed variables. For example, in this case, the `i`th instance will be assigned a value of `pts[i]` for the `bkPower` variable along with all the variables in the `fixed` dict. Further, for each instance, a folder `point_i` is created and set as working directory. Finally, the abaqus python script in `czmtestkit.abaqus_modules.ADCB2` is executed in the working directory with instance input data. The `czmtestkit.abaqus_modules.ADCB2` funciton creates a unit width CAE model with plain strain conditions and runs the finite element simulation of the model using Abaqus/CAE resulting in `.odb` file with the output database. The history of displacement and reaction force in the active degrees of freedom of the load edge are recorded using the Abaqus/CAE history output functionality and can be extracted from the `.odb` file. `czmtestkit.abaqus_modules.historyOutput` function extracts this data and generates a `.csv` with the data. Further, since `czmtestkit.abaqus_modules.ADCB2` creates a unit width CAE model, the `Results` function returns

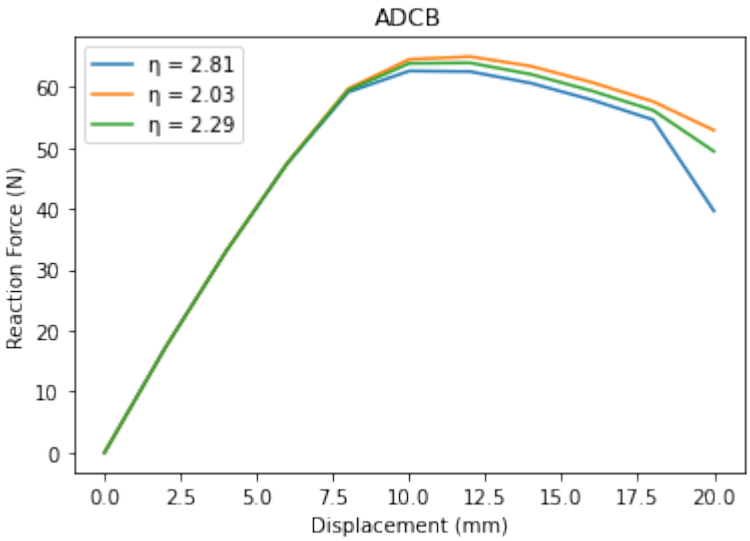
a dictionary with the magnitudes of the displacement and reaction force adjusted to required width by multiplying the actual width. `run_sims` iteratively runs these abaqus and python functions for each instance. The input parameter and output from all the instances are saved to the `Database.json` file in the test directory.

[5]:
```
from czmtestkit.py_modules import Results
from czmtestkit.py_modules import run_sim
run_sim(TestSet, doe, fixed, 'czmtestkit.abaqus_modules.ADCB2', 'czmtestkit.abaqus_
↪modules.historyOutput', Results)
```

Reading and plotting the output data.Read the database and plot the output data.

[6]:
```
# Reading output file
file = open(os.path.join(TestSet, 'Database.json'), 'r')
data_text = file.readlines()
Data = []
for entry in data_text:
    Data.append(json.loads(entry))

# Extracting output data and input variables
U = []
RF = []
eta = []
for entry in Data:
    U.append(np.array(entry['Displacement']))
    RF.append(np.array(entry['Reaction Force']))
    eta.append(entry['bkPower'])

file.close()

# Plotting the data
fig, ax = plt.subplots()
for i in range(len(eta)):
    ax.plot(U[i], RF[i], label='\u03B7 = {:.2f}'.format(eta[i]))
ax.set_title('ADCB')
ax.set_xlabel('Displacement (mm)')
ax.set_ylabel('Reaction Force (N)')
ax.legend()
```

[6]:
```
<matplotlib.legend.Legend at 0x2cd5f53c220>
```



2.1.3 Post Processing

The class `czmtestkit.py_modules.ADCB` is the analytical models from appendix B in the master thesis title `Finite Element Model For Interfaces In Compatibilized Polymer Blends` and the class method `czmtestkit.py_modules.ADCB.rCurve` converts the displacement and reaction force data to fracture resistance curves (r Curves) for a given instance. The `czmtestkit.py_modules.run_analysis` function iteratively runs the `czmtestkit.py_modules.ADCB.rCurve` method for all the instances (dictionaries) in `Database.json` of the `TestSet` directory and appends the generated r Curve data back to the database.

[7]:
```
from czmtestkit.py_modules import ADCB
from czmtestkit.py_modules import run_analysis

model = ADCB()
run_analysis(TestSet, model.rCurve)
```

Read the database and plot the output data.

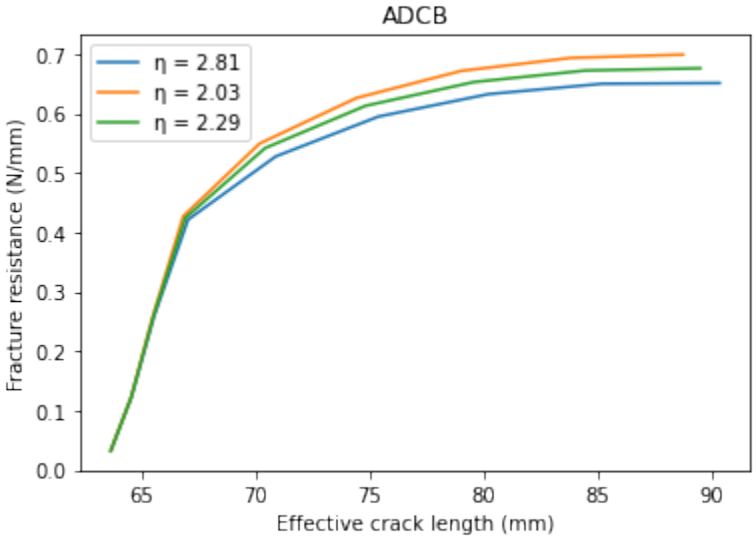
[8]:
```
# Reading output file
file = open(os.path.join(TestSet, 'Database.json'), 'r')
data_text = file.readlines()
Data = []
for entry in data_text:
    Data.append(json.loads(entry))

# Extracting output data
a_e = [] # effective crack length
gR = [] # fracture toughness / fracture resistance
eta = []
for entry in Data:
    a_e.append(np.array(entry['Crack Length']))
    gR.append(np.array(entry['Fracture Resistance']))
    eta.append(entry['bkPower'])
```

(continues on next page)

(continued from previous page)

```
file.close()

# Plotting
fig, ax = plt.subplots()
for i in range(len(eta)):
    ax.plot(a_e[i], gR[i], label='\u03B7 = {:.2f}'.format(eta[i]))
ax.set_title('ADCB')
ax.set_xlabel('Effective crack length (mm)')
ax.set_ylabel('Fracture resistance (N/mm)')
ax.legend()
```

[8]:
```
<matplotlib.legend.Legend at 0x2cd5f4c6c70>
```



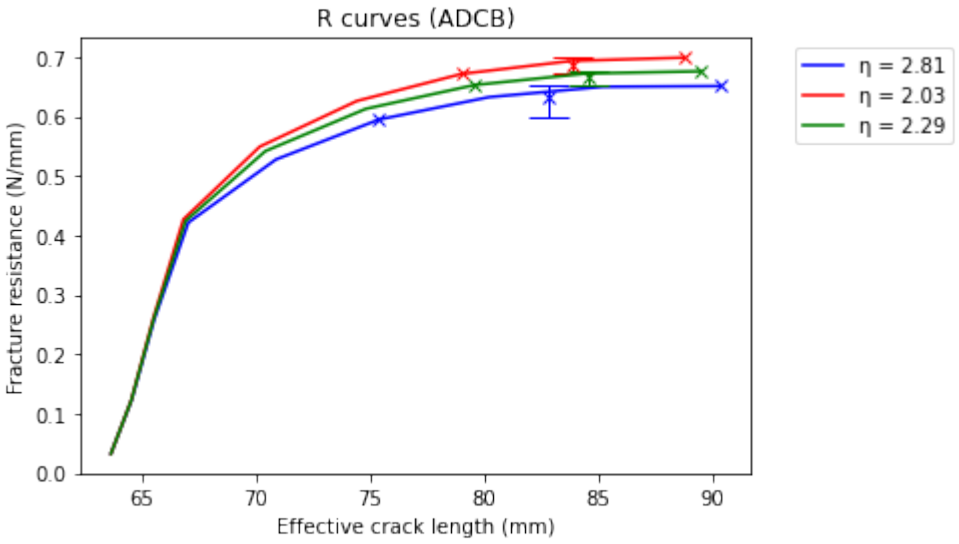
Sample the plateaus in the r Curves for fracture toughness estimation and plot the sample statistics (mean and confidence intervals).

[11]:
```
# defining the plateau region
a_l = 75 # Crack length at plateau start
a_u = 110 # Crack length at plateau end

# picking samples and calculating the sample statistics
## Initializing empty vectors
g_sample = []
alim_sample = []
glim_sample = []
Gt = []
CI = []
## Iterating through the tests
for i in range(len(eta)):
    ind = np.where(np.logical_and(a_e[i]>=a_l, a_e[i]<=a_u)) # Fetching samples with␣
↪crack length within the limits choosen
    g_sample.append(gR[i][ind[0].tolist()]) # Sampling fracture tougness corresponding␣
↪to the sampled crack lengths
    alim_sample.append(np.array([a_e[i][ind[0][0]], a_e[i][ind[0][-1]]])) # Crack length␣
↪sample limits
```

(continues on next page)

(continued from previous page)

```
    glim_sample.append(np.array([gR[i][ind[0][0]], gR[i][ind[0][-1]]])) # Fracture␣
↪toughness sample limits
    mean = g_sample[i].mean() # Fracture toughness sample mean
    p025 = np.percentile(g_sample[i], 2.5) # Fracture toughness lower quartile
    p975 = np.percentile(g_sample[i], 97.5) # Fracture toughness upper quartile
    Gt.append(mean) # Appending to list of means of all the tests
    CI.append([[mean - p025], [p975 - mean]]) # Appending to list of confidence␣
↪intervals of all the tests

# Plotting the r-Curve with fracture toughness samples and statistics
c = ['b', 'r', 'g', 'mediumvioletred', 'purple','olive','grey','maroon','teal','m']
# Iterating through Tests 1, 2, and 3
for i in range(len(eta)):
    plt.plot(a_e[i], gR[i], label='\u03B7 = {:.2f}'.format(eta[i]), color=c[i])
    plt.plot(alim_sample[i], glim_sample[i], 'x', color=c[i])
    plt.errorbar(alim_sample[i].mean(), Gt[i], yerr=CI[i], fmt = 'x', color=c[i],␣
↪ecolor=c[i], elinewidth = 1, capsize=10) #label='Mean: \u03B7 = {:.3f}'.format(eta[i])
plt.legend(bbox_to_anchor=(1.05, 1.0), loc='upper left')
plt.xlabel('Effective crack length (mm)')
plt.ylabel('Fracture resistance (N/mm)')
plt.title('R curves (ADCB)')
plt.show()
```



Since, the BK criterion was used to in the Abaqus/CAE model, the same is used to estimate a mode ratio for each estimated fracture toughness.

[13]:
```
def BK_criteria(gT, GIc, GIIc, eta):
    return ((gT - GIc)/(GIIc - GIc))**(1/eta)
```

The fracture toughness for one instance of the doe is a list of values sampled from its r Curve. Predicting the mode ratio for this list results in a list of predicted mode ratios for the instance. The means and confidence intervals of the lists of predicted mode ratios and the fracture toughness for each instance of the test are plotted here.

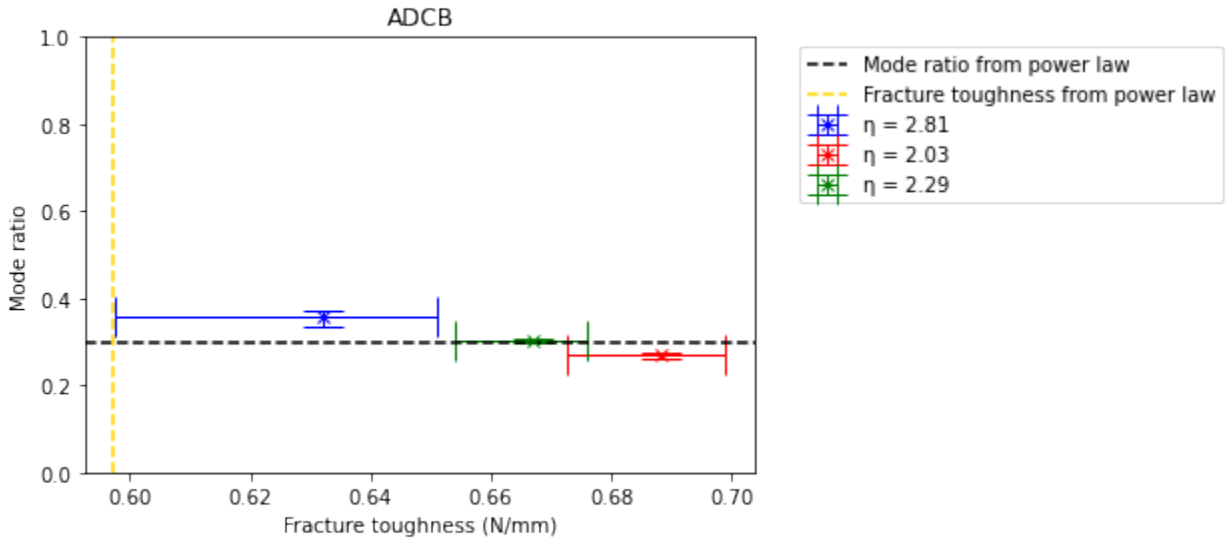
[14]:
```
# Initializing empty vectors
```

(continues on next page)

(continued from previous page)

```
B = []
b_samp = []
CI_B = []
# Predicted ADCB mode ratios and sample statistics
for i in range(len(eta)):
    b = BK_criteria(g_sample[i], fixed['GcNormal'], fixed['GcShear'], eta[i])
    b_samp.append(b)
    mean = b.mean()
    p025 = np.percentile(b, 2.5)
    p975 = np.percentile(b, 97.5)
    B.append(mean)
    CI_B.append([[mean - p025], [p975 - mean]])
    plt.errorbar(Gt[i], B[i], xerr=CI[i], yerr=CI_B[i], fmt = 'x', label='\u03B7 = {:.2f}
↪'.format(eta[i]), color=c[i], ecolor=c[i], elinewidth = 1, capsize=10)

## Plot area steup
plt.axhline(y=0.297807890402863, color='black', linestyle='--', label='Mode ratio from␣
↪power law')
plt.axvline(x=0.597068684259331, color='gold', linestyle='--', label='Fracture toughness␣
↪from power law')
plt.legend(bbox_to_anchor=(1.05, 1.0), loc='upper left')
plt.ylim((0, 1))
plt.xlabel('Fracture toughness (N/mm)')
plt.ylabel('Mode ratio')
plt.title('ADCB')
```

[14]:
```
Text(0.5, 1.0, 'ADCB')
```



2.2 Comparing The Small Displacement and Large Displacement Formulations

This notebook is an example comparing the mixed-mode response of two user element subroutines modeling the cohesive zone elements. The following mixed-mode ADCB tests are the tests defined in section 3.2.3 on `Mixed-Mode Tests With Low Modulus Bulk`, presented in the master thesis title `Finite Element Model For Interfaces In Compatibilized Polymer Blends`.

Outline

1) Create design of experiments. ✓
2) Run simulations. ✓
3) Read odb data. ✓
4) Generate R curves. ✓
5) Extract Fracture toughness samples from R curves. ✓
6) Predict mode ratio based on the sampled fracture toughness and assumed mixed-mode energy criteria. ✓

2.2.1 Preprocessing

Import required modules.

[1]:
```
# imports
import json
import numpy as np
from IPython import display
from scipy.stats import qmc
from matplotlib import pyplot as plt
from matplotlib.lines import Line2D
%matplotlib inline
```

Create a dictionary of the variables required to generate and run the Abaqus/CAE model that are common for all the tests.

[2]:
```
TestSet = 'ADCB_UEL' # Folder name for the set of experiments

# Constants for the experiments
fixed = {'JobID':'ADCB_UEL',
    'Length':50,
    'Width':25,
    'tBot':10,
    'tCz':0.2,
    'Crack':1,
    'DensityBulkTop':1.8e-9,
    'ETop':(5.0, 5.0, 5.0, 0.3, 0.3, 0.3, 5/(2*(1+0.3)), 5/(2*(1+0.3)), 5/(2*(1+0.3))),
↪#(109000.0, 8819.0, 8819.0, 0.34, 0.34, 0.38, 4315.0, 4315.0, 3200.0),
    'DensityBulkBot':1.8e-9,
    'EBot': (5.0, 5.0, 5.0, 0.3, 0.3, 0.3, 5/(2*(1+0.3)), 5/(2*(1+0.3)), 5/(2*(1+0.3))),
↪#(109000.0, 8819.0, 8819.0, 0.34, 0.34, 0.38, 4315.0, 4315.0, 3200.0),
    'DensityCz':1.8e-9,
    'StiffnessCz':1,
```

(continues on next page)

(continued from previous page)

```
    'GcNormal': 0.1,
    'GcShear':1,
    'gFailureNormal':2,
    'gFailureShear':2,
    'MeshCrack':1,
    'MeshX':1,
    'MeshZ':1,
    'Displacement':20,
    'bkPower':1,
    'nCpu':1,
    'nGpu':0,
    'submit':True}
```

Further, define the variables that have to be changed for each experiment.

```
[3]: top = [[3], [5], [8]] # Values of interest for the thickness of the upper adherand or␣
     ↪bulk
     sub = [[{'type':'UEL', 'path':'SDF.for', 'intProp':[]}],
         [{'type':'UEL', 'path':'LDF.for', 'intProp':[]}]] # User subroutines of interest

     # Creating lists with all combinations of the two variables
     topList = []
     subList = []
     for i in top:
         for j in sub:
             topList.append(i)
             subList.append(j)
     n_pts = len(topList)

     # Defining a dictionary with lists of values for the variables.
     doe = {'tTop':topList,
         'userSub':subList}
     doe['nPoints'] = [x for x in range(n_pts)]
```

2.2.2 Running simulations

The `run_sim` function iterates through each point in the design of experiments (doe) and creates an instance of the input data by assigning the values for the variables from their corresponding lists in the `doe` dict along with all the fixed variables. For example, in this case, the `i`th instance will be assigned a value of `topList[i]` for the `tTop` variable and `subList[i]` for the `userSub` variable along with all the variables in the `fixed` dict. Further, for each instance, a folder `point_i` is created and set as working directory. Finally, the abaqus python script in `czmtestkit.abaqus_modules.ADCB2` is executed in the working directory with instance input data using the following command.

```
[ ]: from czmtestkit.py_modules import run_sim
     run_sim(TestSet, doe, fixed, 'czmtestkit.abaqus_modules.ADCB2')
```

`czmtestkit.abaqus_modules.ADCB2` creates a unit width CAE model with plain strain conditions and runs the finite element simulation of the model using Abaqus/CAE resulting in `.odb` file with the output database. The history of displacement and reaction force in the active degrees of freedom of the load edge are recorded using the Abaqus/CAE history output functionality and can be extracted from the `.odb` file. `czmtestkit.abaqus_modules.historyOutput` function extracts this data and generates a `.csv` with the data. Further, since `czmtestkit.abaqus_modules.ADCB2` creates a unit width CAE model, the `Results` function returns a dict with the magnitudes of the displacement and

reaction force adjusted to required width by multiplying the actual width. `run_sims` can also iteratively run these abaqus post processing and python post processing functions for each instance. The input parameter and output from all the instances are saved to the `Database.json` file in the test directory.

Note: The Abaqus/CAE files for this testset have not been included in the git repository.

[ ]:
```
from czmtestkit.py_modules import Results
from czmtestkit.py_modules import run_sim
run_sim(TestSet, doe, fixed, abaqus_postProc='czmtestkit.abaqus_modules.historyOutput',
↪postProc=Results)
```

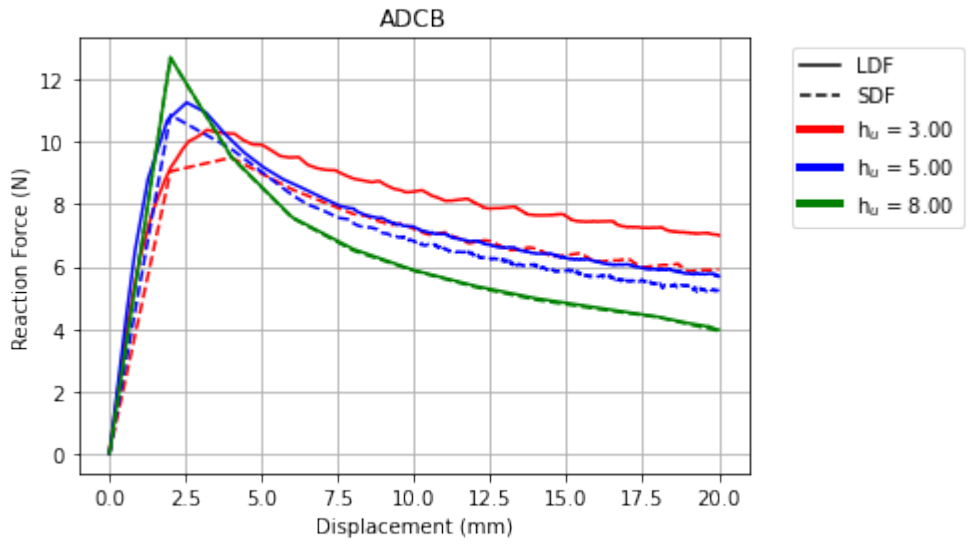
Read the database and plot the output data.

[4]:
```
# Reading output file
import os
file = open(os.path.join(TestSet, 'Database.json'), 'r')
data_text = file.readlines()
Data = []
for entry in data_text:
    Data.append(json.loads(entry))

# Extracting output data and input variables
U = []
RF = []
czImp = []
hu = []
for entry in Data:
    U.append(np.array(entry['Displacement']))
    RF.append(np.array(entry['Reaction Force']))
    hu.append(entry['tTop'])
    SubROut = os.path.split(entry['userSub']['path'])[-1].split('.')[0]
    czImp.append(SubROut)

file.close()

# Defining arrays useful for generating plots
ls = []
c = []
cl = []
m = []
for i in range(n_pts):
    if czImp[i]=='LDF':
        ls.append('solid')
        cl.append('r')
    else:
        ls.append('dashed')
        cl.append('b')

    if hu[i]==3:
        c.append('r')
        m.append('s')
    elif hu[i]==5:
        c.append('b')
        m.append('d')
```

(continues on next page)

(continued from previous page)

```
    else:
        c.append('g')
        m.append('*')

# Plotting
fig, ax = plt.subplots()
for i in range(n_pts):
    ax.plot(U[i], RF[i], c=c[i], linestyle=ls[i])
ax.set_title('ADCB')
ax.set_xlabel('Displacement (mm)')
ax.set_ylabel('Reaction Force (N)')
plt.grid()
## Custom legend
custom_lines = [Line2D([0], [0], color='black', ls='solid'),
                Line2D([0], [0], color='black', ls='dashed'),
                Line2D([0], [0], color='r', lw=4),
                Line2D([0], [0], color='b', lw=4),
                Line2D([0], [0], color='g', lw=4)]
ax.legend(custom_lines, ['LDF', 'SDF', 'h$_u$ = {:.2f}'.format(3), 'h$_u$ = {:.2f}'.
↪format(5), 'h$_u$ = {:.2f}'.format(8)], bbox_to_anchor=(1.05, 1.0), loc='upper left')
```



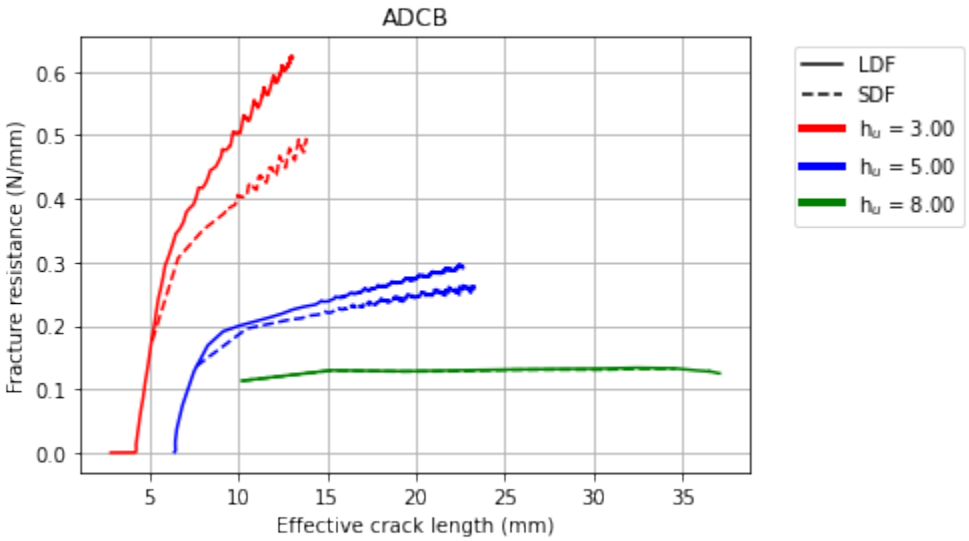
2.2.3 Post Processing

The class `czmtestkit.py_modules.ADCB` is the analytical models from appendix B in the master thesis title `Finite Element Model For Interfaces In Compatibilized Polymer Blends` and the class method `czmtestkit.py_modules.ADCB.rCurve` converts the displacement and reaction force data to fracture resistance curves (r Curves) for a given instance. The `czmtestkit.py_modules.run_analysis` function iteratively runs the `czmtestkit.py_modules.ADCB.rCurve` method for all the instances (dictionaries) in `Database.json` of the `TestSet` directory and appends the generated r Curve data back to the database.

[ ]:
```
from czmtestkit.py_modules import ADCB # Analytical model for the test
from czmtestkit.py_modules import run_analysis # Runs the function all the instance in␣
↪the doe.

model = ADCB()
run_analysis(TestSet, model.rCurve)
```

Read the database and plot the output data.

[5]:
```
# Reading output file
file = open(os.path.join(TestSet, 'Database.json'), 'r')
data_text = file.readlines()
Data = []
for entry in data_text:
    Data.append(json.loads(entry))

# Extracting output data
a_e = [] # effective crack length
gR = [] # fracture toughness / fracture resistance
for entry in Data:
    a_e.append(np.array(entry['Crack Length']))
    gR.append(np.array(entry['Fracture Resistance']))

file.close()

# Plotting
fig, ax = plt.subplots()
for i in range(n_pts):
    ax.plot(a_e[i], gR[i], c=c[i], linestyle=ls[i])
plt.grid()
ax.set_title('ADCB')
ax.set_xlabel('Effective crack length (mm)')
ax.set_ylabel('Fracture resistance (N/mm)')
## Custom legend
custom_lines = [Line2D([0], [0], color='black', ls='solid'),
                Line2D([0], [0], color='black', ls='dashed'),
                Line2D([0], [0], color='r', lw=4),
                Line2D([0], [0], color='b', lw=4),
                Line2D([0], [0], color='g', lw=4)]
ax.legend(custom_lines, ['LDF', 'SDF', 'h$_u$ = {:.2f}'.format(3), 'h$_u$ = {:.2f}'.
↪format(5), 'h$_u$ = {:.2f}'.format(8)], bbox_to_anchor=(1.05, 1.0), loc='upper left')
```



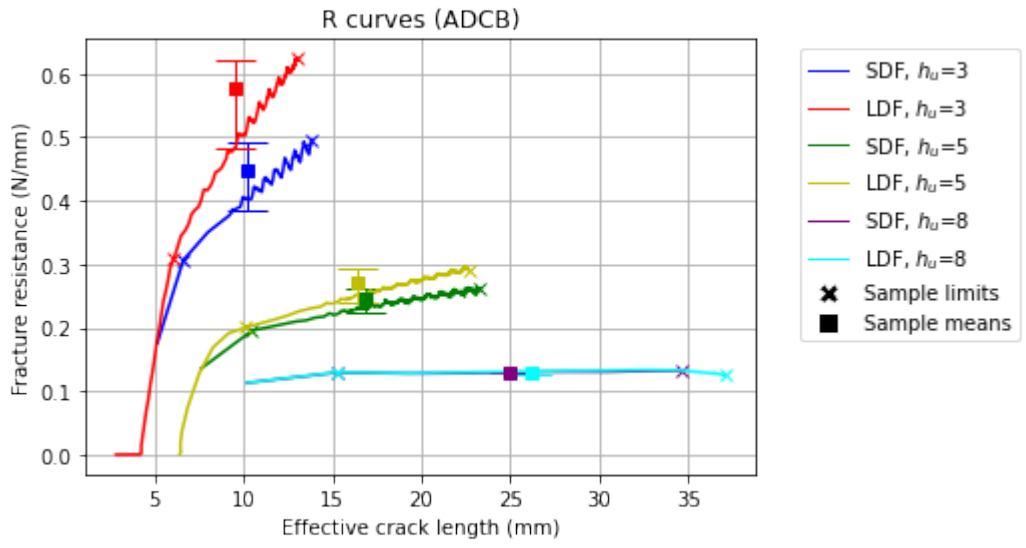
Sample the plateaus in the r Curves for fracture toughness estimation and plot the sample statistics (mean and confidence intervals).

[10]:
```
# defining the plateau region
a_l = [6,6,10,10,15,15] # Crack length at plateau start
a_u = [40,40,40,40,40,40] # Crack length at plateau end

# picking samples and calculating the sample statistics
## Initializing empty vectors
g_sample = []
alim_sample = []
glim_sample = []
Gt = []
CI = []

## Iterating through the tests
for i in range(n_pts):
    ind = np.where(np.logical_and(a_e[i]>=a_l[i], a_e[i]<=a_u[i])) # Fetching samples␣
↪with crack length within the limits choosen
    g_sample.append(gR[i][ind[0].tolist()]) # Sampling fracture tougness corresponding␣
↪to the sampled crack lengths
    alim_sample.append(np.array([a_e[i][ind[0][0]], a_e[i][ind[0][-1]]])) # Crack length␣
↪sample limits
    glim_sample.append(np.array([gR[i][ind[0][0]], gR[i][ind[0][-1]]])) # Fracture␣
↪toughness sample limits
    mean = g_sample[i].mean() # Fracture toughness sample mean
    p025 = np.percentile(g_sample[i], 2.5) # Fracture toughness lower quartile
    p975 = np.percentile(g_sample[i], 97.5) # Fracture toughness upper quartile
    Gt.append(mean) # Appending to list of means of all the tests
    CI.append([[mean - p025], [p975 - mean]]) # Appending to list of confidence␣
↪intervals of all the tests

# Plotting the r-Curve with fracture toughness samples and statistics
C = ['b', 'r', 'g', 'y', 'purple', 'cyan'] # List of colors for plotting
```

(continues on next page)

(continued from previous page)

```
Lab = [] # List of labels for custom legend
custom_lines = [] # List of keys for custom legend
fig, ax = plt.subplots() # Empty figure

## Iterating through Tests and plotting data
for i in range(n_pts):
    custom_lines.append(Line2D([0], [0], color=C[i], lw=1))
    Lab.append(czImp[i]+', $h_u$={}'.format(hu[i]))
    plt.plot(a_e[i], gR[i], color=C[i])
    plt.plot(alim_sample[i], glim_sample[i], 'x', color=C[i])
    plt.errorbar(alim_sample[i].mean(), Gt[i], yerr=CI[i], fmt = 's', color=C[i],␣
↪ecolor=C[i], elinewidth = 1, capsize=10) #label='Mean: \u03B7 = {:.3f}'.format(eta[i])

## Custom legend
custom_lines.append(Line2D([0], [0], marker='X', color='w', markerfacecolor='black',␣
↪markersize=10))
Lab.append('Sample limits')
custom_lines.append(Line2D([0], [0], marker='s', color='w', markerfacecolor='black',␣
↪markersize=10))
Lab.append('Sample means')
ax.legend(custom_lines, Lab, bbox_to_anchor=(1.05, 1.0), loc='upper left')

## Plot area steup
plt.xlabel('Effective crack length (mm)')
plt.ylabel('Fracture resistance (N/mm)')
plt.title('R curves (ADCB)')
plt.grid()
plt.show()
```

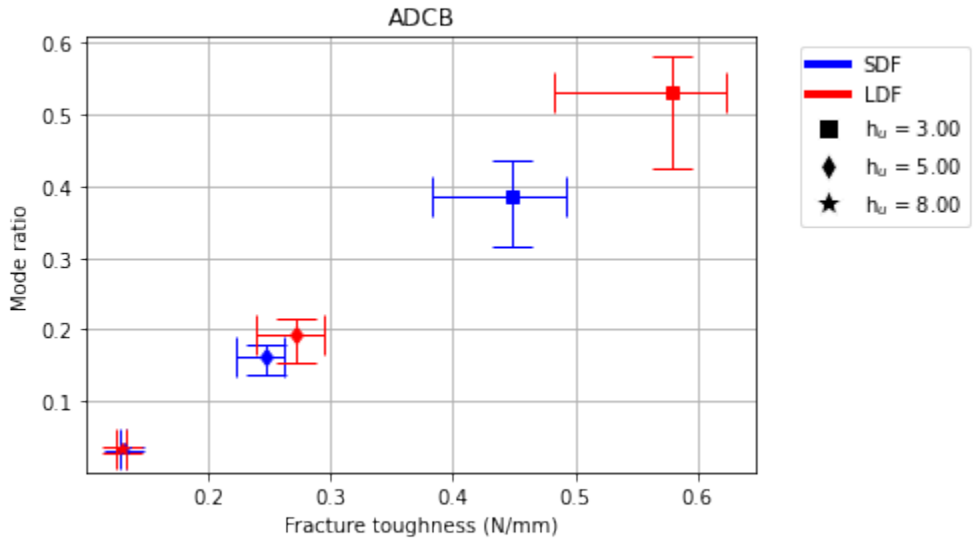


Since, the BK criterion was used to in the Abaqus/CAE model, the same is used to estimate a mode ratio for each estimated fracture toughness.

[8]:
```
def BK_criteria(gT, GIc, GIIc, eta):
    return ((gT - GIc)/(GIIc - GIc))**(1/eta)
```

The fracture toughness for one instance of the doe is a list of values sampled from its r Curve. Predicting the mode ratio for this list results in a list of predicted mode ratios for the instance. The means and confidence intervals of the lists of predicted mode ratios and the fracture toughness for each instance of the test are plotted here.

[11]:
```
# Initializing empty vectors
B = []
CI_B = []
fig, ax = plt.subplots() # Empty figure


# Predicting and plotting mode ratios corresponding to the fracture toughness samples
for i in range(n_pts):
    b = BK_criteria(g_sample[i], fixed['GcNormal'], fixed['GcShear'], fixed['bkPower'])
↪#Predicting mode ratios
    mean = b.mean() # Mean of predicted mode ratios
    p025 = np.percentile(b, 2.5) # lower quartile mode ratios
    p975 = np.percentile(b, 97.5) # upper quartile mode ratios
    B.append(mean) # mean mpde ratio
    CI_B.append([[mean - p025], [p975 - mean]]) # 95% confidence intervals of mode␣
↪predicted mode ratios
    plt.errorbar(Gt[i], B[i], xerr=CI[i], yerr=CI_B[i], fmt = m[i], color=cl[i],␣
↪ecolor=cl[i], elinewidth = 1, capsize=10)

## Custom legend
custom_lines = [Line2D([0], [0], color='b', lw=4),
                Line2D([0], [0], color='r', lw=4),
                Line2D([0], [0], marker=m[0], color='w', markerfacecolor='black',␣
↪markersize=10),
                Line2D([0], [0], marker=m[2], color='w', markerfacecolor='black',␣
↪markersize=10),
                Line2D([0], [0], marker=m[4], color='w', markerfacecolor='black',␣
↪markersize=15)]
ax.legend(custom_lines, ['SDF', 'LDF', 'h$_u$ = {:.2f}'.format(hu[0]), 'h$_u$ = {:.2f}'.
↪format(hu[2]), 'h$_u$ = {:.2f}'.format(hu[4])], bbox_to_anchor=(1.05, 1.0), loc='upper␣
↪left')

## Plot area steup
plt.xlabel('Fracture toughness (N/mm)')
plt.ylabel('Mode ratio')
plt.title('ADCB')
plt.grid()
plt.show()
```



2.3 Visualizing Internal Variables Printed From The User Element Subroutine

This notebook presents the steps to use the `czmtestkit` package to read and visualize internal variables from the user element subroutine printed to the `.msg` file while running the simulation. Inorder to use this functionality of the package, print the variable `Var` of interest to the `.msg` file using the following command in the fortran code of the subroutine:

```
WRITE(7,*) 'VarKey = ', Var
```

Run `ADCB_UEL.ipynb` to generate sample files. Read the `.msg` file and find the line numbers in the `.msg` file corresponding to the stable increments using `czmtestkit.py_modules.find_convergedIncrements`.

[6]:
```
from czmtestkit.py_modules import find_convergedIncrements
File= 'ADCB_UEL\point_05\ADCB_UEL.msg'
converged_time,  last_unconv_it, converged_it = find_convergedIncrements(File)
print('Last line numbers from converged increments:', '\n',converged_it, '\n')
print('Last line numbers from the last increment before converged increments:', '\n',␣
↪last_unconv_it, '\n')
print('Step times of converged increments:', '\n',converged_time, '\n')
```

```
Last line numbers from converged increments:
 [13275, 19849, 25333, 32927, 43743, 75658, 95899, 103400, 122627, 191886, 213140,␣
↪224839, 865040, 1506327, 1537206, 1545720, 1547865, 1555295, 1557440]

Last line numbers from the last increment before converged increments:
 [12147, 18734, 24207, 31801, 40518, 74557, 94799, 102300, 121527, 188687, 212040,␣
↪223739, 862883, 1505226, 1534007, 1544620, 1544620, 1552094, 1552094]

Step times of converged increments:
 ['0.100', '0.200', '0.300', '0.400', '0.500', '0.600', '0.700', '0.800', '0.900', '0.925
↪', '0.950', '0.975', '0.981', '0.984', '0.987', '0.991', '0.994', '0.999', '1.00']
```

Initialize an `increment` class instance for every stable increment and assign the limiting line numbers corresponding to the increment. Line number corresponding to the last line from the increment before a converged increment is the start of the increment and the line number corresponding to the last of converged increment is the end of the increment.

Then, for each stable increment, find the line numbers where the variable `Var` is printed using the key word `VarKey` and the `find_lineNumbers` method.

```
increment.findlineNumbers('VarKey', p, (m,n))
```

where p is the number of lines for each entry of the variable `Var` and `m,n` are the number of rows and columns. For example, the variable `Var` is printed to the `.msg` file as

```
VarKey = 1.000000 2.000000 3.000000
 4.000000 5.000000 6.000000
```

Since the entry extends to two lines the p is 2. Further, if the variable `Var` has two rows and three columns with m = 2 and n = 3.

`find_lineNumbers` method creates an instance of the `data` class and stores the corresponding line numbers as instance attributes each time the `Var` is printed within the limits of the stable increment. A list of all the `data` class instances created is stored as an attribute of the `increment` class instance.

Further, `find_data` method fetches the data between the line numbers fetched by `find_lineNumbers` method and converts it to matrix of the shape `m,n` for all the `data` class instances in the list. For example above entry is converted to

$$\begin{bmatrix} 1.000000 & 3.000000 & 5.000000 \\ 2.000000 & 4.000000 & 6.000000 \end{bmatrix}$$

The matix is assigned to the `value` attribute of the corresponding `data` class instance. Finally, `find_data` creates `data` attribute for the `increment` class instance which is a dictionary with the `VarKey` as keys and the lists of corresponding `data` class instances as values.

[2]:
```
from czmtestkit.py_modules import increment
entries = []
for j in range(len(converged_it)):
    ent = increment()
    ent.fileName = File
    ent.step_time = converged_time[j]
    ent.start = last_unconv_it[j]
    ent.stop = converged_it[j]
    entries.append(ent)
    ent.find_lineNumbers('IP',1,(2,1))
    ent.find_lineNumbers('GIP',1,(3,1))
    ent.find_lineNumbers('TAU',1,(3,1))
    ent.find_lineNumbers('DTAU',3,(3,3))
    ent.find_lineNumbers('DG',24,(3,24))
    ent.find_lineNumbers('DR',24,(3,24))
    ent.find_data()
```

Finally, edit the `setup_frame` and `frame_plot` methods of the `frame_iterator` class to present variables of interest as colormaps and create a movie of frames from all the stable increments. Use the `setup_frame` method to setup the plot area, titles, and axes, and `frame_plot` method to plot the increments.

Note: Do not change the input parameters of the setup_frame and frame_plot methods.

[3]:
```
import numpy as np
from IPython import display
from matplotlib import pyplot as plt
from matplotlib import animation as animation
%matplotlib inline

class frame_iterator:
    def __init__(self, array, fileName):
        self.mainArray = array
        self.itr = len(array)
        self.name = fileName

    def animate(self): # Animator
        self.setup_frame()
        Writer = animation.FFMpegWriter
        writer = Writer(fps=1, metadata=dict(artist='Nanditha Mudunuru'), bitrate=1800)

        anim = animation.FuncAnimation(self.fig, self.frame_plot, frames=self.itr,␣
↪blit=False, repeat=True)
        anim.save(self.name+'.mp4', writer=writer)

    def setup_frame(self): # Main frame
        fig = plt.figure(figsize=(12,15))
        self.fig = fig

        # dR color map
        ax1 = fig.add_subplot(121)
        self.ax1 = ax1
        ax1.set_title('dR', fontsize=20)
        self.im1 = ax1.imshow(np.zeros([24,3]), cmap='seismic')
        self.fig.colorbar(self.im1, ax=self.ax1, orientation='vertical')

        # Major ticks and labels
        ax1.set_yticks(np.arange(0, 24, 1))
        ax1.set_xticks(np.arange(0, 3, 1))
        ax1.set_yticklabels(np.arange(1, 25, 1))
        ax1.set_xticklabels(np.arange(1, 4, 1))

        # Minor ticks and grid
        ax1.set_yticks(np.arange(-.5, 24, 1), minor=True)
        ax1.set_xticks(np.arange(-.5, 3, 1), minor=True)
        ax1.grid(which='minor', color='black', linewidth=2)

        # tau color map
        ax2 = fig.add_subplot(222)
        self.ax2 = ax2
        ax2.set_title('\u03C4', fontsize=20)
        self.im2 = ax2.imshow(np.zeros([3,1]), cmap='seismic')
        self.fig.colorbar(self.im2, ax=self.ax2, orientation='vertical')

        # Major ticks and labels
        ax2.set_xticks(np.arange(0, 1, 1))
        ax2.set_yticks(np.arange(0, 3, 1))
```

(continues on next page)

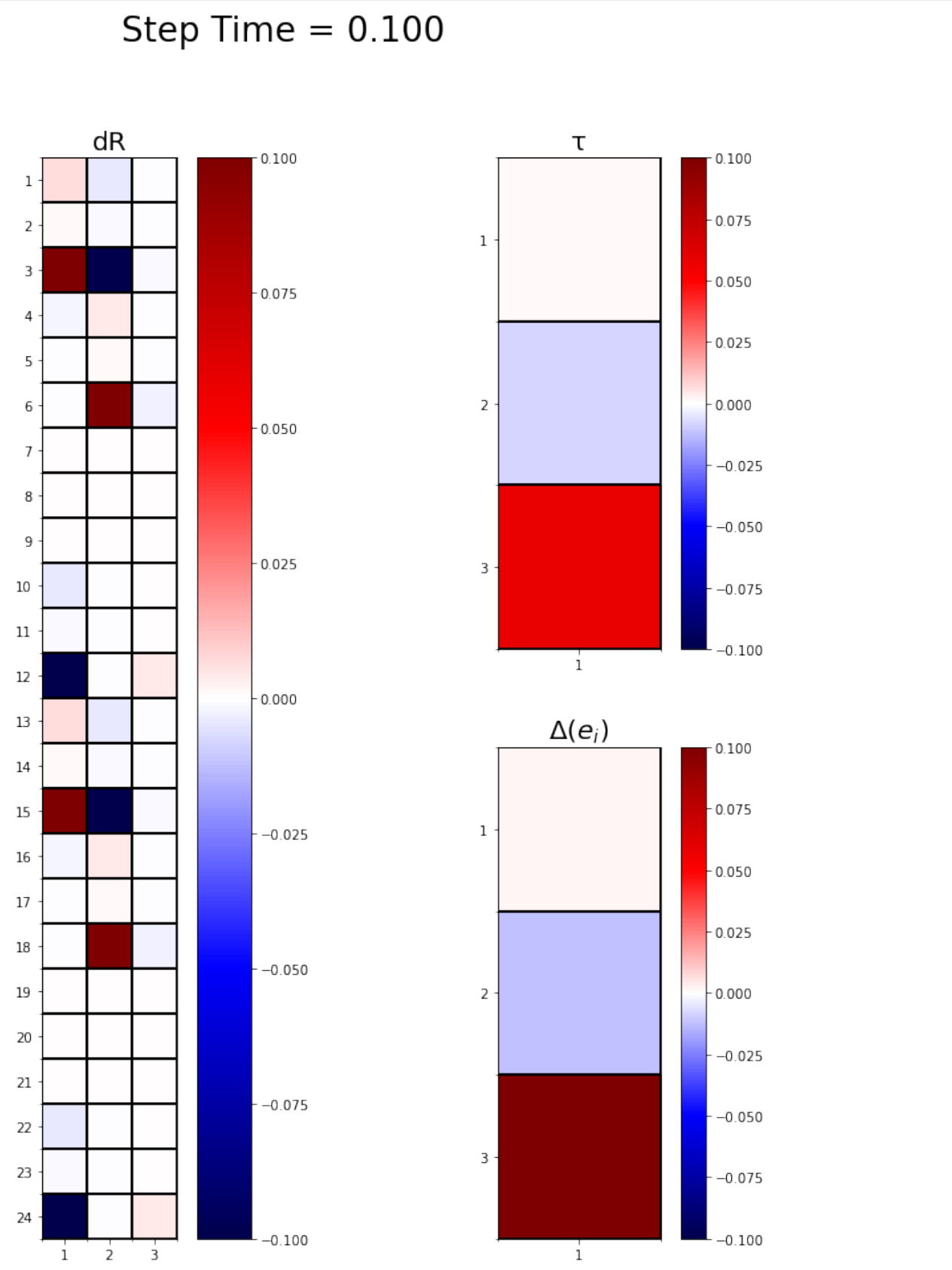
(continued from previous page)

```
        ax2.set_xticklabels(np.arange(1, 2, 1))
        ax2.set_yticklabels(np.arange(1, 4, 1))

        # Minor ticks and grid
        ax2.set_xticks(np.arange(-.5, 1, 1), minor=True)
        ax2.set_yticks(np.arange(-.5, 3, 1), minor=True)
        ax2.grid(which='minor', color='black', linewidth=2)

        # GIP color map
        ax3 = fig.add_subplot(224)
        self.ax3 = ax3
        ax3.set_title(r'$\Delta(e_i)$', fontsize=20)
        self.im3 = ax3.imshow(np.zeros([3,1]), cmap='seismic')
        self.fig.colorbar(self.im3, ax=self.ax3, orientation='vertical')

        # Major ticks and labels
        ax3.set_xticks(np.arange(0, 1, 1))
        ax3.set_yticks(np.arange(0, 3, 1))
        ax3.set_xticklabels(np.arange(1, 2, 1))
        ax3.set_yticklabels(np.arange(1, 4, 1))

        # Minor ticks and grid
        ax3.set_xticks(np.arange(-.5, 1, 1), minor=True)
        ax3.set_yticks(np.arange(-.5, 3, 1), minor=True)
        ax3.grid(which='minor', color='black', linewidth=2)

        return

    def frame_plot(self, i): # Frame update at each stable increment
        IP_idx = 0
        dr = self.mainArray[i].data['DR'][IP_idx].value.T
        tau = self.mainArray[i].data['TAU'][IP_idx].value
        gip = self.mainArray[i].data['GIP'][IP_idx].value
        time = self.mainArray[i].step_time

        self.fig.suptitle('Step Time = {}'.format(time), fontsize=26)
        self.im1.set_data(dr)
        self.im2.set_data(tau)
        self.im3.set_data(gip)
        return

test = frame_iterator(entries, 'h_u=8')
test.animate()
plt.close()
display.Video("h_u=8.mp4")
```

[3]:
```
<IPython.core.display.Video object>
```

Or plot individual frames from required increments

[5]:
```
test.setup_frame()
test.frame_plot(0)
plt.show()
```


Step Time = 0.100
dR
τ
$\Delta(e_i)$
0.100
0.075
0.050
0.025
0.000
−0.025
−0.050
−0.075
−0.100

CHAPTER

THREE

GUIDELINES FOR CONTRIBUTORS

Code of conduct

Contributers are to kindly stick to the following guidelines and the contributor covenant code of conduct to avoid conflicts and misunderstandings during the collaboration process.

3.1 Types of Contributions

A contribution can be one of the following cases:

1. you have a question;
2. you think you may have found a bug (including unexpected behaviour);
3. you want to make some changes to the code base (e.g. to fix a bug, to add a new feature, to update documentation).

The sections below outline the steps in each case.

3.1.1 Questions

1. use the search functionality here to see if someone already filed the same issue;
2. if your issue search did not yield any relevant results, make a new issue;
3. apply the "Question" label; apply other labels when relevant.

3.1.2 Find Bugs

If you think you may have found a bug:

1. use the search functionality here to see if someone already filed the same issue;
2. if your issue search did not yield any relevant results, make a new issue, making sure to provide enough information to the rest of the community to understand the cause and context of the problem. Depending on the issue, you may want to include:
    - the SHA hashcode of the commit that is causing your problem;
    - some identifying information (name and version number) for dependencies you're using;
    - information about the operating system;
    - detailed steps to reproduce the bug.

3. apply relevant labels to the newly created issue.

3.1.3 Changes to Source Code: fix bugs and add features

1. (important) announce your plan to the rest of the community before you start working. This announcement should be in the form of a (new) issue;
2. (important) wait until some consensus is reached about your idea is a good idea;
3. if needed, fork the repository to your own Github profile and create your feature branch out of the latest master commit. While working on your feature branch, make sure to stay up to date with the master branch by pulling in changes;
4. make sure the existing tests still work;
5. add your tests (if applicable);
6. update or expand the documentation;
7. push your feature branch to (your fork of) this repository on GitHub;
8. create the pull request, e.g. following the instructions here.

Following sections are specific instruction to guide you in executing steps 3 to 8 of the list above.

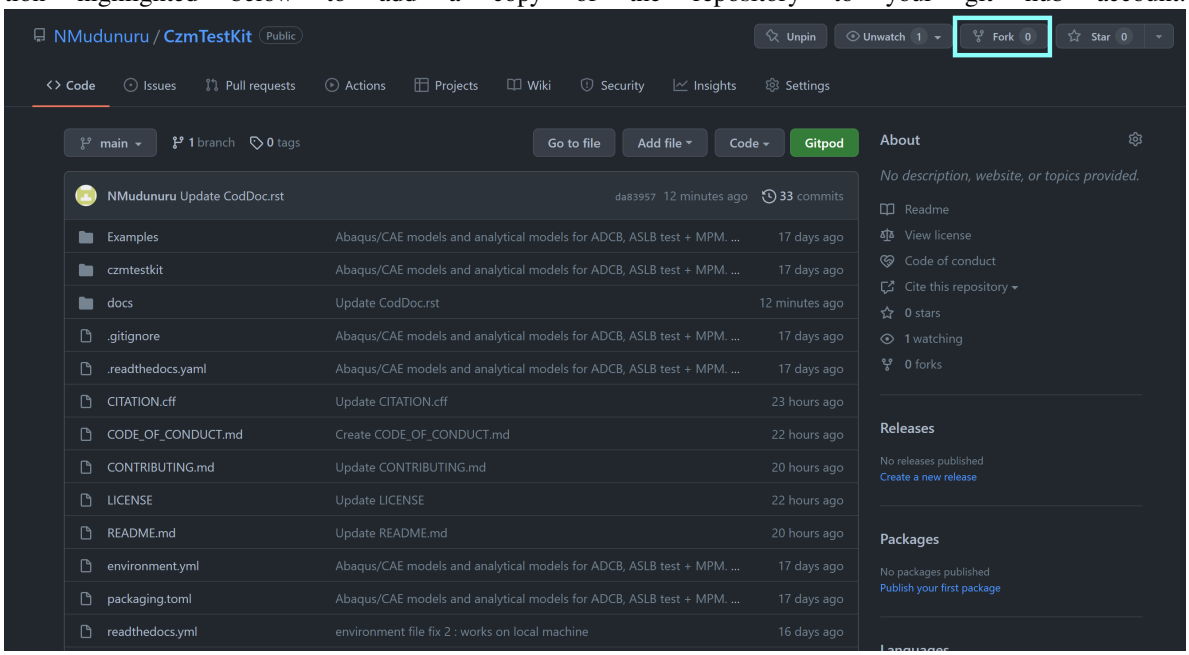
3.2 Steps for Contributing

3.2.1 Steps for developing the package

Ensure that the prerequists from the documentation have been satisfied.

Windows 10

1. Fork the git repository https://github.com/NMudunuru/CzmTestKit.git of the package. The link to the git repository will take you to the following page. Use the Fork option highlighted below to add a copy of the repository to your git hub account.



2. Setup the `CzmTestKit` repository on your local machine by cloning your fork repository.

```
$ git clone <url of your fork>
```

3. Create conda environment `CzmTestKit` with dependencies required for the package. This can be done using the `environment.yml` file in the `CzmTestKit` directory.

```
$ cd <path to the package environment file for example C:\Users\User\Desktop\
↪CzmTestKit>
$ conda env create -f environment.yml
```

4. Activate the environment.

```
$ conda activate CzmTestKit
```

5. Install the source code of the package. Do not use `$ pip install CzmTestKit` here, as this will install the distributed PyPI version of the package. The goal here is to install the package from the local package for testing. Therefore, use the following command from the local `CzmTestKit` directory.

```
$ pip install -e .
```

6. Edit the source code and reinstall the package. Repeat steps 4, 5 to reinstall the package. See the source code documentation for guidelines on current code functionality and structure. If needed, contact the primary authors to guide you through the code.

7. Test the changes and repeat the previous step if further changes are necessary.

8. When you are ready, update the documentation and push the changes to your remote. Then, send a pull request to the main package repository in https://github.com/NMudunuru/CzmTestKit.git, e.g. following the instructions here. Use the following checklist to ensure that the changes are clear and well documented.

```
- [] Update the following meta data in the docstrings of the module source code:
    - [] Date of edit or update.
    - [] Author info and email.
    - [] Module version.
    - [] Inline comments. (Include author and version info for updates)
- [] Create a tutorial for using the update in the form of an example jupyter␣
↪notebook in the `Examples` subdirectoty.
- [] Add the tutorial notebook to the documentation.
- [] Update the description and the version in the `README.md`, and the `setup.py`␣
↪file.
- [] Update documentation version in `docs\source\conf.py`
```

Add this list to your pull request message and change the `[]`s of completed items in the list to `[x]`.

If you feel like you have a valuable contribution to make, but you don't know how to complete some of the items in this checklist such as writing or running tests or updating the documentation: don't let this discourage you from making the pull request; we can help you! Just go ahead and submit the pull request, but keep in mind that you might be asked to make additional commits to your pull request.

3.2.2 Steps for testing the documentation

The package documentation is hosted by `readthedocs`, where the changes merged to `main` branch of the git repository are automatically reflected in the published documentation. However, it is necessary to locally test the documentation before pushing to the `main`. Use the following steps to locally build and test the documentation.

Windows 10

1. Setup the local repository using instructions from the previous section.
2. Create conda environment `docs` with dependencies required for building the documentation. This can be done using the `environment.yml` file in the `docs` subdirectory of the `CzmTestKit` directory.

```
$ cd <path to the docs environment file for example C:\Users\User\Desktop\
↪CzmTestKit\docs>
$ conda env create -f environment.yml
```

3. Activate the environment.

```
$ conda activate docs
```

4. From the `docs` subdirectory in `CzmTestKit`, execute the documenation source.

```
$ cd <path to the docs environment file for example C:\Users\User\Desktop\
↪CzmTestKit\docs>
$ make html
```

Executing the documentation source will result in a `build` directory in the `docs` subdirectory with the html files in `docs\build\html` subdirectory, where `docs\build\html\index.html` will be the home page for the documentation. If your command line interface does not recongnize the `make` command you can use the following method to generate the documentation build

```
$ sphinx-build source build
```

5. Repeat the previous step till the updates in the documentation are satisfactory.

3.3 Code Documentation

3.3.1 Module for use With Python Environment

czmtestkit.py_modules Package

Functions

Inertia(b, h)	**Second moment of area along the out of plane through the mid point of diagonal for a rectangular crossection.**
Results(dict)	**Calculates the effective displacement and reaction force from history output.**
abqFun(InputData, function, wd)	**Run abaqus-python modules as subprocesses.**
find_convergedIncrements(fileName)	**Find line numbers corresponding to converged increments.**
run_analysis(JobID, analysis_func[, setup_func])	**Sequentially run python functions using dictionaries from the `Database.json`.**
run_sim(name, doe_data, fixed_data[, ...])	**Sequentially run Abaqus/CAE simulations and/or post processing functions for a design of experiments.**

Inertia

czmtestkit.py_modules.**Inertia**(*b*, *h*)

Second moment of area along the out of plane through the mid point of diagonal for a rectangular crossection.

Parameters **b** (*float*): crossection width (length along the first axis of the global coordinate system)

h (*float*): crossection height (length along the second axis of the global coordinate system)

Metadata

Environment

Abaqus/CAE

Version

v1.0.0

Date

2022-01-18

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

Results

czmtestkit.py_modules.**Results**(*dict*)

Calculates the effective displacement and reaction force from history output.

Parameters **dict** (*dict*): Input data for the instance in the design of experiments required to execute *Results()* functions.

Keys Values

'JobID' (*str*) File name of the *.csv* file with history output of reaction force and displacement extracted from the *.odb* file.

'Width' (*float*) Since the CAE models are of unit width, the results are adjusted using the actual width as a multiplier.

Returns

OutputData (*dict*): History output data.

Keys Values

'Reaction Force' (*list*) Magnitude of the reaction force effective over the total specimen width.

'Displacement' (*list*) Opening displacement.

'NodeSet' (*list*) Name of the node from which the history output was extracted.

Example

Read reaction force and history output extracted from `.odb` file and printed to to `.csv` file like the one below. (See *czmtestkit.abaqus_modules.historyOutput()* to generate this `.csv` file.)

Table 3.1: ExampleJob.csv

Node ASSEMBLY.2	Node ASSEMBLY.2	Node ASSEMBLY.2	Node ASSEMBLY.2	Node ASSEMBLY.2	Node ASSEMBLY.2
RF	RF	RF	U	U	U
1	2	3	1	2	3
-0.0	0.0	-0.0	0.0	0.0	0.0
0.0018873203080014075	0.0	0.6899987459182704	0.0	0.0	2.0
-0.000110366716398858	0.0	1.3232073783874503	0.0	0.0	4.0
-3.97297917515971e-05	0.0	1.895939111709599	0.0	0.0	6.0
9.446500371268488e-06	0.0	2.3647692203521707	0.0	0.0	8.0
0.0002365063992328894	0.0	2.503397226333623	0.0	0.0	10.0
5.893718480365351e-05	0.0	2.4995803833007803	0.0	0.0	12.0
-0.000678690907079726	0.0	2.423223257064823	0.0	0.0	14.0
2.220768510596831e-05	0.0	2.313085794448853	0.0	0.0	16.0
-6.12034546065843e-06	0.0	2.183591127395633	0.0	0.0	18.0
-0.00282513070851564	0.0	1.5845751762390103	0.0	0.0	20.0

If the actual specimen width is $25mm$, Then run the `Results()` fetches effective displacements(u) and reaction force(rf) at *Node ASSEMBLY.2* such that:

$$U = \sqrt{{u_1}^2 + {u_2}^2 + {u_3}^2}$$

$$RF = 25 * \sqrt{{rf_1}^2 + {rf_2}^2 + {rf_3}^2}$$

```
InDict = {'JobID': 'ExampleJob', 'Width': 25}
Output = Results(InDict)
print(Output)
```

Output

```
{"Displacement": [0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0, 18.0, 20.0],
↪"NodeSet": ["Node ASSEMBLY.2"], "Reaction Force": [0.0, 17.2500331765394, 33.
↪08018457475526, 47.39847780314658, 59.11923050927596, 62.58493093763735, 62.
↪48950959989038, 60.580583802698186, 57.82714486388642, 54.58977818510519, 39.
↪61444236730786]}
```

Metadata

Environment

Python

Version

v1.0.0

Date

2021-06-18

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

abqFun

czmtestkit.py_modules.**abqFun**(*InputData*, *function*, *wd*)

Run abaqus-python modules as subprocesses. See `czmtestkit.abaqus_modules` for available functions and instructions to create your own abaqus-python function that is compatible with the `czmtestkit`.

Parameters **InputData** (*str*): `.json` file name with input dictionary.

function (*str*): abaqus_modules function based on abaqus-python script to be executed.

wd (*str*): work directory for the abaqus_modules functIon.

Example

Assume that `czmtestkit.abqPy_Func1` is a function based on abaqus-python scripting language to run Abaqus/CAE sim

Also assume that `abqPy_Func1` takes a dictionary as input parameter with *param_1*, *param_2* and *param_3* as keys within the dictionary. Run the abaqus-python script using *`abqFun()`* as shown below

```
# Create dictionary
In_dict_Func1 = {
        'param_1': value_param1
        'param_2': value_param2
        'param_3': value_param3
}

# Write data to a .json file
import json
filePath = 'dataIn.json'
with open(filePath, 'a') as file:
        json.dump(In_dict_Func1, file)
```

(continues on next page)

(continued from previous page)

```
        file.write("\n")

# Execute the abaqus-python script with czmtestkit in the current working directory
import os
czmtestkit.py_modules.abqFun(filePath, 'czmtestkit.abaqus_modules.abqPy_Func1',os.
↪getcwd())
```

Metadata

Environment

Python

Version

v1.0.0

Date

2021-12-11

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

find_convergedIncrements

`czmtestkit.py_modules.`**`find_convergedIncrements`**(*fileName*)

Find line numbers corresponding to converged increments.

Abaqus prints the statement `EQUILIBRIUM NOT ACHIEVED WITHIN TOLERANCE` at the end of each increment where convergence was not achieved and the step time details including the `FRACTION OF STEP COMPLETED` at the end of each converged increment. *`find_convergedIncrements()`* retrives the line numbers for the end of each increment using these strings as keywords. Futher, it finds the fraction of step time of converged increments from the corresponding lines and appends it to `converged_time`. Along with appending the line number corresponding to the end of each converged increment to `converged_lastLine`, *`find_convergedIncrements()`* also finds and appends the largest line numbers marking the end of an increment (converged or unconverged) that is closest to and smaller than the entries in `converged_lastLine`. This marks the end of an increment that occured just prior to the converged increment and consequently also marks the start of the converged increment.

Parameters **fileName** (*str*): path to the *.msg* file including the file name and extension.

Returns

converged_time (*List*): Increment time of converged increments. The times in the list are floats.

converged_firstLine (*List*): First line of the converged increments. The line numbers in the list are integers.

converged_lastLine (*List*): Last line of the converged increments. The line numbers in the list are integers.

Example

If `filename.msg` file has the following content,

```
678          FORCE     EQUILIBRIUM NOT ACHIEVED WITHIN TOLERANCE.
```

```
1118         FORCE     EQUILIBRIUM NOT ACHIEVED WITHIN TOLERANCE.
```

```
2978 TIME INCREMENT COMPLETED  0.500    ,  FRACTION OF STEP COMPLETED  0.500
```

```
3158 TIME INCREMENT COMPLETED  0.750    ,  FRACTION OF STEP COMPLETED  0.750
```

```
3852         FORCE     EQUILIBRIUM NOT ACHIEVED WITHIN TOLERANCE.
```

```
4200 TIME INCREMENT COMPLETED  1.000    ,  FRACTION OF STEP COMPLETED  1.000
```

Executing the *find_convergedIncrements()* function finds the line numbers and increment times.

```
time, start, stop = find_convergedIncrements("filename.msg")
print('Time = ', time)
print('Start = ', start)
print('Stop = ', stop)
```

Output

```
Time = [0.5, 0.75, 1]
Start = [1118, 2978, 3852]
Stop = [2978, 3158, 4200]
```

Metadata

Environment

Python

Version

v1.0.0

Date

2022-03-06

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

---

run_analysis

czmtestkit.py_modules.**run_analysis**(*JobID*, *analysis_func*, *setup_func=None*)

Sequentially run python functions using dictionaries from the `Database.json`. (See example from *run_sim()* for details on generating the `Database.json`). Output dictionary items are appended to the `Database.json` file.

Parameters **JobID** (*str*): ID for colleciton of tests in the design of experiments.

analysis_func (*function object*): Post processing function using output from *run_sim()*.

setup_func (*function object*): Setup for the post processing function.

Example

To analyze data from the `Database` generated with *run_sim()* for example:

```
$ <current working directory>
└── ExampleDOE
    ├── point_00
    ├── point_01
    ├── point_02
    ├── point_03
    └── Database.json
```

Database.json

```
{'param_1': value_param1, ..., 'output2': value0_out2}
{'param_1': value_param1, ..., 'output2': value1_out2}
{'param_1': value_param1, ..., 'output2': value2_out2}
{'param_1': value_param1, ..., 'output2': value3_out2}
```

using an analysis function that uses a dictionary from the Database as input. For example:

```
def analysisFunc(dict):
    ...
    return {'analysisOut1': <value>}
```

(continues on next page)

(continued from previous page)

```
dictIn = {'param_1': value_param1, ..., 'output2': value0_out2}
dictOut = analysisFunc(dictIn)
```

run_analysis() can be used to iterate through the entries in the Database and append the resulting dictionary items to the database. Running the following code

```
run_analysis('ExampleDOE', analysisFunc)
```

will result in updated `Database.json`.

Database.json

```
{'param_1': value_param1, ..., 'output2': value0_out2,'analysisOut1': value0_out3}
{'param_1': value_param1, ..., 'output2': value1_out2,'analysisOut1': value1_out3}
{'param_1': value_param1, ..., 'output2': value2_out2,'analysisOut1': value2_out3}
{'param_1': value_param1, ..., 'output2': value3_out2,'analysisOut1': value3_out3}
```

Metadata

Environment

Python

Version

v1.0.0

Date

2021-12-11

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

---

run_sim

czmtestkit.py_modules.**run_sim**(*name*, *doe_data*, *fixed_data*, *abaqus_simFunc=None*, *abaqus_postProc=None*, *postProc=None*)

Sequentially run Abaqus/CAE simulations and/or post processing functions for a design of experiments.

Parameters **name** (*str*): ID for colleciton of tests in the design of experiments.

doe_data (*dict*): dictionary with variables for tests in the design of experiments.

'nPoints' List of indicies corresponding to the tests to be run. (The defintion of this attribute was changed, see version history for details.)

'Variable_Key_1' List of values for the variable named *Variable_Key_1*.

'Variable_Key_2' List of values for the variable named *Variable_Key_2*.

fixed_data (*dict*): dictionary with constants for tests in the design of experiments.

'Constant_Key_1' Value of the variable named *Constant_Key_1*.

'Constant_Key_2' Value of the variable named *Constant_Key_2*.

abaqus_simFunc (*str*): Name of abaqus-python function from `czmtestkit.abaqus_modules`. See `Example` for the differnce between `abaqus_simFunc` and `abaqus_postProc` parameters and `czmtestkit.abaqus_modules` for available functions and instructions to create your own abaqus-python function that is compatible with the `czmtestkit`.

abaqus_postProc (*str*): Name of abaqus-python function from `czmtestkit.abaqus_modules`. See `Example` for the differnce between `abaqus_simFunc` and `abaqus_postProc` parameters and `czmtestkit.abaqus_modules` for available functions and instructions to create your own abaqus-python function that is compatible with the `czmtestkit`.

postProc (*function object*): Executable python post processing function.

Example

Assume that `abqPy_Func1` is a function in the `czmtestkit` based on abaqus-python scripting language to run Abaqus/CAE simulation. Also assume that `abqPy_Func1` takes a dictionary as input parameter with *param_1*, *param_2* and *param_3* as keys within the dictionary.

```
In_dict_Func1 = {
    'param_1': value_param1,
    'param_2': value_param2,
    'param_3': value_param3,
}
```

If *param_1* requires a string input then *value_param1* has to be a string and so forth. Further, if the design of experiments (doe) is such that the *param_2* is to assume different values:

```
doe_param2 = [value0_param2, value1_param2, value2_param2, value3_param2]
```

and the aim is to run the simulations for the first and the third points (*value0_param2, value2_param2*) in this doe, then define the input dictionaries as follows:

```
dict_fix = {
    'param_1': value_param1,
    'param_3': value_param3,
```

(continues on next page)

(continued from previous page)

```
}

dict_var = {
    'param_2' : doe_param2,
    'nPoints' : [0,2]
}
```

Use the *run_sim()* function to sequentially run the tests:

```
run_sim('ExampleDOE', dict_var, dict_fix, abaqus_simFunc="czmtestkit.abaqus_modules.
↪abqPy_Func1")
```

This creates the main directory *ExampleDOE* in the current work directory and sub directories for each test. Further, with sub directories as working directories, each test in the *nPoints* is executed resulting in abaqus files.

```
$ <current working directory>
└── ExampleDOE
    ├── point_00
    │   ├── <abaqus file 1 test 1>
    │   ├── <abaqus file 2 test 1>
    │   ├── <abaqus file 3 test 1>
    │   ├── <abaqus file 4 test 1>
    │   ├── <abaqus file 5 test 1>
    │   └── point_00.json
    └── point_02
        ├── <abaqus file 1 test 2>
        ├── <abaqus file 2 test 2>
        ├── <abaqus file 3 test 2>
        ├── <abaqus file 4 test 2>
        ├── <abaqus file 5 test 2>
        └── point_02.json
```

Here, the first test is exectued with the following dictionary saved to `point_00.json` and passed to `czmtestkit.abaqus_modules.abqPy_Func1()` as input.

```
{
    'param_1': value_param1,
    'param_2': value0_param2,
    'param_3': value_param3,
}
```

and the input dictionary for the second test with the third point in the doe saved to `point_02.json` as

```
{
    'param_1': value_param1,
    'param_2': value2_param2,
    'param_3': value_param3,
}
```

Remaining tests in the doe can be executed by adding corresponding indices to the 'nPoints' list in the *doe_data*. Further, if a postprocessing function `abqPy_Func2` has to be executed with parameters *param_1*, *param_2* and a new parameter *param_4*. *run_sim()* can be used again.

```
dict_fix2 = {
    'param_1': value_param1,
    'param_4': value_param4,
}

dict_var2 = {
    'param_2' : doe_param2,
    'nPoints' : [0,1,2,3] # assuming `abqPy_Func1` has already been executed for all␣
↪the points. The directories should have already been created.
}

run_sim('ExampleDOE', dict_var2, dict_fix2, abaqus_postProc="czmtestkit.abaqus_
↪modules.abqPy_Func2")
```

Similarly, python postprocessor (`py_func`) can be executed sequentially for the doe using `run_sim(..., postProc=py_func)`. Executing a `postProc` function results in the creation of a `Database.json` in the test directory with input dictionaries from all the tests.

```
$ <current working directory>
└── ExampleDOE
    ├── point_00
    ├── point_01
    ├── point_02
    ├── point_03
    └── Database.json
```

Database.json

```
{'param_1': value_param1, 'param_2': value0_param2, 'param_3': value_param3, 'param_
↪4': value_param4 }
{'param_1': value_param1, 'param_2': value1_param2, 'param_3': value_param3, 'param_
↪4': value_param4 }
{'param_1': value_param1, 'param_2': value2_param2, 'param_3': value_param3, 'param_
↪4': value_param4 }
{'param_1': value_param1, 'param_2': value3_param2, 'param_3': value_param3, 'param_
↪4': value_param4 }
```

If the `py_func` results in an output dictionary `{'output1': <value>, 'output2': <value>}`, the contents of the output dictionary are appended to the input dictionaries before writting to the `Database.json`.

Database.json

```
{'param_1': value_param1, 'param_2': value0_param2, 'param_3': value_param3, 'param_
↪4': value_param4, 'output1': value0_out1, 'output2': value0_out2}
{'param_1': value_param1, 'param_2': value1_param2, 'param_3': value_param3, 'param_
↪4': value_param4, 'output1': value1_out1, 'output2': value1_out2}
{'param_1': value_param1, 'param_2': value2_param2, 'param_3': value_param3, 'param_
↪4': value_param4, 'output1': value2_out1, 'output2': value2_out2}
{'param_1': value_param1, 'param_2': value3_param2, 'param_3': value_param3, 'param_
↪4': value_param4, 'output1': value3_out1, 'output2': value3_out2}
```

Note: All three functions can be executed at once.

```
FixDict = {
    'param_1': value_param1,
    'param_3': value_param3,
    'param_4': value_param4,
}

VarDict = {
    'param_2' : doe_param2,
    'nPoints' : [0,1,2,3]
}

run_sim('ExampleDOE', VarDict, FixDict, abaqus_simFunc="czmtestkit.abaqus_modules.
↪abqPy_Func1", abaqus_postProc="czmtestkit.abaqus_modules.abqPy_Func2",␣
↪postProc=py_func)
```

Metadata

Environment

Python

Version

v1.1.0	Updated type and functionality of doe_data['npoints']. v1.0.0: (*Int*) Number of points in the design on experiments.
v1.0.0	base version

Date

2021-12-11

Authors

Nanditha Mudunuru

Contribution: v1.0.0, v1.1.0

Email: nanditha.mudunuru@gmail.com

Classes

ADCB()	**Analyse DCB and ADCB specimens using Timoshenko beam theory and Castigliano theorem as described in appendix B of the master thesis `[1]`_.**
ASLB()	**Analyse SLB and ASLB specimens using Timoshenko beam theory and Castigliano theorem as described in appendix B of the master thesis `[1]`_.**
ENF()	**Analyse ENF specimens using Timoshenko beam theory and Castigliano theorem as described in appendix B of the master thesis `[1]`_.**
Model()	**Parent class for analytical models of standardized tests described in appendix B of the master thesis `[1]`_.**
data()	**Read and store a matrix written to Abaqus/CAE `.msg` files.**
increment()	**Read and store matrices written to Abaqus/CAE `.msg` file.**

ADCB

class czmtestkit.py_modules.**ADCB**

Bases: *czmtestkit.py_modules.analyticalMixedMode.Model*

Analyse DCB and ADCB specimens using Timoshenko beam theory and Castigliano theorem as described in appendix B of the master thesis *[1]*.

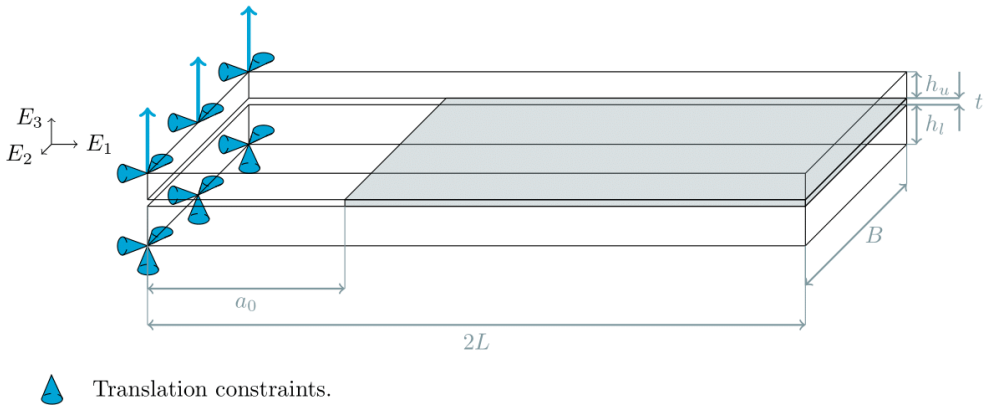


Fig. 3.1: **Asymmetric Double Cantilever Beam schematic** *[1]*.
*Here, the translation degrees of freedom parallel to the axis of the `blue* cones are fixed. Additionally, the shaded region represents the cohesive zone interface while the unshaded region represents the bulk adherands or plies.`

> **Warning:** The input parameters should be consistent in their units of measurement. Following are some commonly used groups of units in engineering:

Table 3.2: Consistent set of units [2].

MASS	LENGTH	TIME	FORCE	STRESS	ENERGY
kg	m	s	N	Pa	J
kg	mm	ms	kN	GPa	kN-mm
g	mm	ms	N	MPa	N-mm

Tip: Finite element simulation of this test can be obtained using the *czmtestkit.abaqus_modules.ADCB()* functions.

References:

1) Mudunuru, N. (2022, March 30). Finite Element Model For Interfaces In Compatibilized Polymer Blends. TU Delft Education Repositories. Retrieved on April 21, 2022, from http://resolver.tudelft.nl/uuid:88140513-120d-4a34-b893-b84908fe2373

2) LS-Dyna. (n.d.). Consistent units. Retrieved April 21, 2022, from https://www.dynasupport.com/howtos/general/consistent-units

Metadata

Environment

Abaqus/CAE

Version

v1.0.0

Date

2022-01-18

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

Methods Summary

compliance(a[, data])	**Find the compliance for effective crack length.**
crackLength(u[, data])	**Find the effective crack length for a given opening displacement at the load end.**
resistance(P, a[, data])	**Find the effective fracture resistance (G) given the instantaneous reaction force and effective crack length.**
setup([data])	**Setup equation coefficients.**

Methods Documentation

`compliance`(*a*, *data=None*)

Find the compliance for effective crack length.

Parameters **a** (*float* or *numpy array*): effective crack length.

data (*dict*) Optional : Specimen dimensions and properties. See *Model.setup()* for required key-value pairs.

Only necessary if *ADCB.setup()* has not been previously executed.

Returns **C** (*float* or *numpy array*): effective compliance from the specimen.

`crackLength`(*u*, *data=None*)

Find the effective crack length for a given opening displacement at the load end.

Parameters **u** (*float*): opening displacement.

data (*dict*) Optional : Specimen dimensions and properties. See *Model.setup()* for required key-value pairs.

Only necessary if *ADCB.setup()* has not been previously executed.

Returns **a** (*float*): effective crack length.

`resistance`(*P*, *a*, *data=None*)

Find the effective fracture resistance (G) given the instantaneous reaction force and effective crack length.

Parameters **P** (*float* or *list*): reaction force.

a (*float* or *list*): effective crack length.

data (*dict*) Optional : Specimen dimensions and properties. See *Model.setup()* for required key-value pairs.

Only necessary if *ADCB.setup()* has not been previously executed.

Returns **G** (*float* or *list*): instantaneous fracture resistance or toughness.

`setup`(*data=None*)

Setup equation coefficients.

Parameters **data** (*dict*) Optional : Specimen dimensions and properties. See *Model.setup()* for required key-value pairs.

Only necessary if *Model.setup()* has not been previously executed.

ASLB

class czmtestkit.py_modules.**ASLB**

Bases: *czmtestkit.py_modules.analyticalMixedMode.Model*

Analyse SLB and ASLB specimens using Timoshenko beam theory and Castigliano theorem as described in appendix B of the master thesis *[1]*.

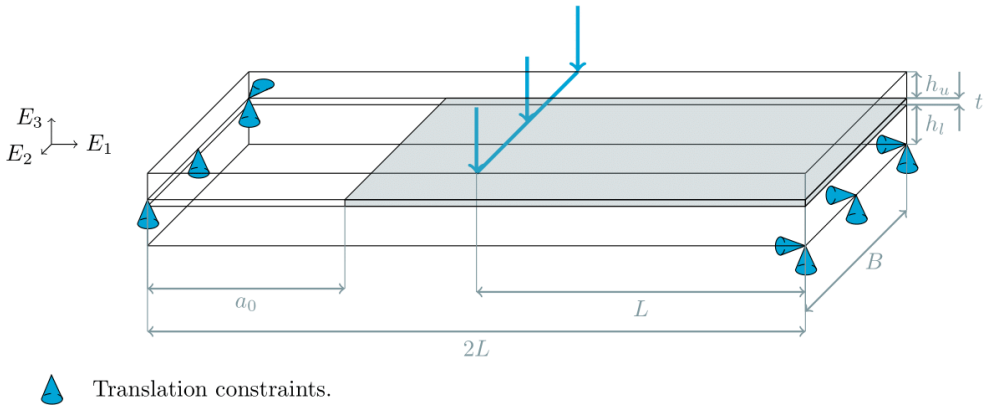


Fig. 3.2: **Asymmetric Single Leg Bending schematic** *[1]*.
*Here, the translation degrees of freedom parallel to the axis of the `blue* cones are fixed. Additionally, the shaded region represents the cohesive zone interface while the unshaded region represents the bulk adherands or plies.`

Warning: The input parameters should be consistent in their units of measurement. Following are some commonly used groups of units in engineering:

Table 3.3: Consistent set of units [2].

MASS	LENGTH	TIME	FORCE	STRESS	ENERGY
kg	m	s	N	Pa	J
kg	mm	ms	kN	GPa	kN-mm
g	mm	ms	N	MPa	N-mm

Tip: Finite element simulation of this test can be obtained using the *czmtestkit.abaqus_modules.ASLB()* or *czmtestkit.abaqus_modules.ASLB2()* functions.

References:

1) Mudunuru, N. (2022, March 30). Finite Element Model For Interfaces In Compatibilized Polymer Blends. TU Delft Education Repositories. Retrieved on April 21, 2022, from http://resolver.tudelft.nl/uuid:88140513-120d-4a34-b893-b84908fe2373

2) LS-Dyna. (n.d.). Consistent units. Retrieved April 21, 2022, from https://www.dynasupport.com/howtos/general/consistent-units

Metadata

Environment

Abaqus/CAE

Version

v1.0.0

Date

2022-01-18

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

Methods Summary

compliance(a[, data])	**Find the compliance for effective crack length.**
crackLength(u[, data])	**Find the effective crack length for a given opening displacement at the load end.**
resistance(P, a[, data])	**Find the effective fracture resistance (G) given the instantaneous reaction force and effective crack length.**
setup([data])	**Setup equation coefficients.**

Methods Documentation

compliance(*a*, *data=None*)

Find the compliance for effective crack length.

Parameters **a** (*float* or *numpy array*): effective crack length.

data (*dict*) Optional : Specimen dimensions and properties. See *Model.setup()* for required key-value pairs.

Only necessary if *ASLB.setup()* has not been previously executed.

Returns **C** (*float* or *numpy array*): effective compliance from the specimen.

crackLength(*u*, *data=None*)

Find the effective crack length for a given opening displacement at the load end.

Parameters **u** (*float*): opening displacement.

data (*dict*) Optional : Specimen dimensions and properties. See *Model.setup()* for required key-value pairs.

Only necessary if *ADCB.setup()* has not been previously executed.

Returns **a** (*float*): effective crack length.

resistance(*P*, *a*, *data=None*)

Find the effective fracture resistance (G) given the instantaneous reaction force and effective crack length.

Parameters **P** (*float* or *list*): reaction force.

a (*float* or *list*): effective crack length.

data (*dict*) Optional : Specimen dimensions and properties. See *Model.setup()* for required key-value pairs.

Only necessary if *ASLB.setup()* has not been previously executed.

Returns **G** (*float* or *list*): instantaneous fracture resistance or toughness.

setup(*data=None*)

Setup equation coefficients.

Parameters **data** (*dict*) Optional : Specimen dimensions and properties. See *Model.setup()* for required key-value pairs.

Only necessary if *Model.setup()* has not been previously executed.

ENF

class czmtestkit.py_modules.**ENF**

Bases: *czmtestkit.py_modules.analyticalMixedMode.Model*

Analyse ENF specimens using Timoshenko beam theory and Castigliano theorem as described in appendix B of the master thesis *[1]*.

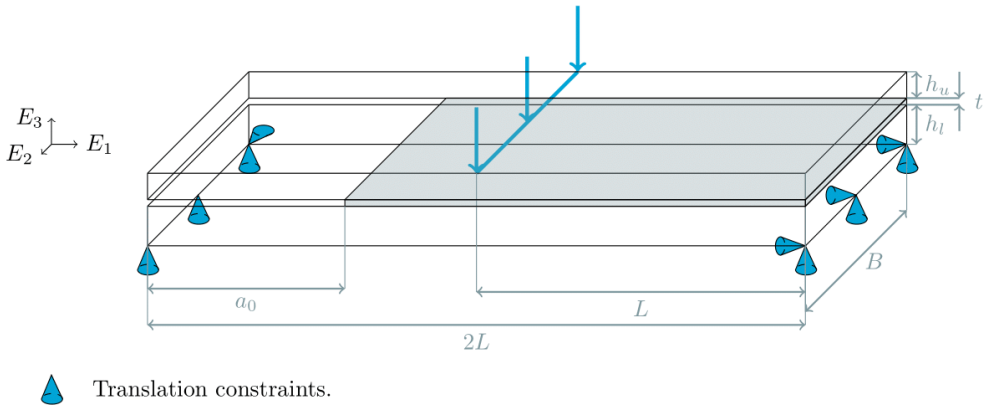


Fig. 3.3: **End Notch Flexure schematic** *[1]*.
*Here, the translation degrees of freedom parallel to the axis of the `blue* cones are fixed. Additionally, the shaded region represents the cohesive zone interface while the unshaded region represents the bulk adherands or plies.`

Warning: The input parameters should be consistent in their units of measurement. Following are some commonly used groups of units in engineering:

Table 3.4: Consistent set of units [2].

MASS	LENGTH	TIME	FORCE	STRESS	ENERGY
kg	m	s	N	Pa	J
kg	mm	ms	kN	GPa	kN-mm
g	mm	ms	N	MPa	N-mm

References:

1) Mudunuru, N. (2022, March 30). Finite Element Model For Interfaces In Compatibilized Polymer Blends. TU Delft Education Repositories. Retrieved on April 21, 2022, from http://resolver.tudelft.nl/uuid:88140513-120d-4a34-b893-b84908fe2373

2) LS-Dyna. (n.d.). Consistent units. Retrieved April 21, 2022, from https://www.dynasupport.com/howtos/general/consistent-units

---

Metadata

Environment

Abaqus/CAE

Version

v1.0.0

Date

2022-01-18

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

Methods Summary

compliance(a[, data])	**Find the compliance for effective crack length.**
crackLength(u[, data])	**Find the effective crack length for a given opening displacement at the load end.**
resistance(P, a[, data])	**Find the effective fracture resistance (G) given the instantaneous reaction force and effective crack length.**
setup([data])	**Setup equation coefficients.**

Methods Documentation

`compliance`(*a*, *data=None*)

Find the compliance for effective crack length.

Parameters **a** (*float* or *numpy array*): effective crack length.

data (*dict*) Optional : Specimen dimensions and properties. See *Model.setup()* for required key-value pairs.

Only necessary if *ENF.setup()* has not been previously executed.

Returns **C** (*float* or *numpy array*): effective compliance from the specimen.

`crackLength`(*u*, *data=None*)

Find the effective crack length for a given opening displacement at the load end.

Parameters **u** (*float*): opening displacement.

data (*dict*) Optional : Specimen dimensions and properties. See *Model.setup()* for required key-value pairs.

Only necessary if *ADCB.setup()* has not been previously executed.

Returns **a** (*float*): effective crack length.

`resistance`(*P*, *a*, *data=None*)

Find the effective fracture resistance (G) given the instantaneous reaction force and effective crack length.

Parameters **P** (*float* or *list*): reaction force.

a (*float* or *list*): effective crack length.

data (*dict*) Optional : Specimen dimensions and properties. See *Model.setup()* for required key-value pairs.

Only necessary if *ENF.setup()* has not been previously executed.

Returns **G** (*float* or *list*): instantaneous fracture resistance or toughness.

`setup`(*data=None*)

Setup equation coefficients.

Parameters **data** (*dict*) Optional : Specimen dimensions and properties. See *Model.setup()* for required key-value pairs.

Only necessary if *Model.setup()* has not been previously executed.

Model

class czmtestkit.py_modules.**Model**

Bases: object

Parent class for analytical models of standardized tests described in appendix B of the master thesis *[1]*.

References:

1) Mudunuru, N. (2022, March 30). Finite Element Model For Interfaces In Compatibilized Polymer Blends. TU Delft Education Repositories. Retrieved on April 21, 2022, from http://resolver.tudelft.nl/uuid:88140513-120d-4a34-b893-b84908fe2373

Metadata

Environment

Abaqus/CAE

Version

v1.0.0

Date

2022-01-18

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

Methods Summary

rCurve([u, P, data])	**Find the effective fracture resistance (G) and instantaneous crack length, given the reaction force corresponding to open displacements.**
reactionForce([u, data])	**Find the reaction force corresponding to open displacements, given the critical fracture toughness (** G_C **).**
setup([data])	**Setup class attributes corresponding to specimen geometry and properties.**

Methods Documentation

`rCurve`(*u=None*, *P=None*, *data=None*)

Find the effective fracture resistance (G) and instantaneous crack length, given the reaction force corresponding to open displacements.

Parameters **P** (*float* or *list*): reaction force.

u (*float* or *list*): opening displacement.

data (*dict*) Optional : Specimen dimensions and properties. See *Model.setup()* for required key-value pairs in addition to the following:

'Displacement' list of opening displacements.

'Reaction Force' list of reaction forces.

Note: If the *setup()* method of *Model* or one of its child classes has already been executed, pass the displacement and reaction force inputs to *rCurve()* method using the `u` and `P` arguments. If not, the input dictionary from *Model.setup()* must be passed along with the additional key-value pairs using the `data` argument. Either `data` or the `P, u` arguments are required, not both.

Warning: Can only be used when a child class with methods *setup()* and *resistance()* are defined. For example, see *czmtestkit.py_modules.ADCB*, *czmtestkit.py_modules.ASLB* and *czmtestkit.py_modules.ENF*

`reactionForce`(*u=None*, *data=None*)

Find the reaction force corresponding to open displacements, given the critical fracture toughness (G_C **).**

Parameters **u** (*float* or *list*): opening displacement.

data (*dict*) Optional : Specimen dimensions and properties. See *Model.setup()* for required key-value pairs in addition to the following:

'Displacement' list of opening displacements.

'gT' Fracture toughness.

Note: If the *setup()* method of *Model* or one of its child classes has already been executed, pass the displacement to *reactionForce()* method using the `u` argument. If not, the input dictionary from *Model.setup()* must be passed along with the additional key-value pairs using the `data` argument. Either the `data` or the `u` arguments are required, not both.

Warning: Can only be used when a child class with methods *setup()*, *crackLength()* and *compliance()* are defined. For example, see *czmtestkit.py_modules.ADCB*, *czmtestkit.py_modules.ASLB* and *czmtestkit.py_modules.ENF*

`setup`(*data=None*)

Setup class attributes corresponding to specimen geometry and properties.

Parameters **data** (*dict*) Optional : Specimen dimensions and properties.

Only necessary if `setup()` method of a child class has not been previously executed.

'JobID' Name of the job.

'Length' Length of the specimen $2L$.

'Width' Width of the specimen B.

'tTop' Thickness of the top adherand/ply h_u.

'tBot' Thickness of the bottom adherand/ply h_l.

'tCz' Thickness of the cohesive zone t.

'Crack' Crack length a_0.

'E' or 'ETop' Tuple of engineering constants for the elastic behaviour of the top adherand/ply.

```
(E1,E2,E3,12,13,23,G12,G13,G23)
```

Warning: The input parameters should be consistent in their units of measurement. Following are some commonly used groups of units in engineering:

Table 3.5: Consistent set of units [1].

MASS	LENGTH	TIME	FORCE	STRESS	ENERGY
kg	m	s	N	Pa	J
kg	mm	ms	kN	GPa	kN-mm
g	mm	ms	N	MPa	N-mm

References:

1) LS-Dyna. (n.d.). Consistent units. Retrieved April 21, 2022, from https://www.dynasupport.com/howtos/general/consistent-units

data

class czmtestkit.py_modules.**data**

Bases: `object`

Read and store a matrix written to Abaqus/CAE `.msg` files.

Attributes **data.type** (*str*): Keyword in the first line to be excluded when reading the data between specified line numbers. Ideally, this should indicate the vairable name of the matrix/array printed to the *.msg* file.

data.start (*int*): Line number corresponding to the first line contating the matrix data in the *.msg* file.

data.stop (*int*): Line number corresponding to the last line contating the matrix data in the *.msg* file. *self.start* = *self.stop* if the matrix/array is printed on a single line.

data.shape (*tuple*): Shape of the matrix [optional].

shape[0] (*int*): number of rows.

shape[1] (*int*): number of columns.

data.value (*list*): Elements of the matrix read from the *.msg* file.

Example

If a variable *Var*

$$Var = \begin{bmatrix} 1.000000 & 3.000000 & 5.000000 \\ 2.000000 & 4.000000 & 6.000000 \end{bmatrix}$$

is printed to the *.msg* file with the keyword *VarKey* as follows:

```
107 VarKey = 1.000000 2.000000 3.000000
108 4.000000 5.000000 6.000000
```

To fetch this data from `fileName.msg` file, create an instance of data class.

```
dataInst = data()
```

The keyword to be excluded when reading the matrix is the *data.type*.

```
dataInst.type = 'VarKey'
```

Since the matrix extends between line 107 and 108:

```
dataInst.start = 107
dataInst.stop = 108
```

Further, the *Var* matrix has two rows and three columns, therefore:

```
dataInst.shape = (2, 3)
```

With these attributes assigned to an instance of the *data* class, executing the *data.findValue()* method fetches and assigns the elements of *Var* matrix to *data.value* as a list `[ 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 ]`.

```
dataInst.findValue("fileName.msg")
print(dataInst.value)
```

Output:

```
[1.000000, 2.000000, 3.000000, 4.000000, 5.000000, 6.000000]
```

Metadata

Environment

Python

Version

v1.0.0

Date

2022-03-06

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

Methods Summary

findValue(fileName)	**Method to find the matrix/array between specified line numbers in the `filename.msg` file.** (See *data* class for an example.)

Methods Documentation

findValue(*fileName*)

Method to find the matrix/array between specified line numbers in the `filename.msg` file. (See *data* class for an example.)

Parameters **fileName** (*str*): path to the *.msg* file including the file name and extension.

Note: Output from the function is unstructured and has to be reshaped based on the requirements or the *data.shape* attribute.

Warning: *data* attributes *data.type*, *data.start*, and *data.stop* must be defined to execute the *data.findValue()* method.

Metadata

Environment

Python

Version

v1.0.0

Date

2022-03-06

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

increment

class czmtestkit.py_modules.**increment**

Bases: object

Read and store matrices written to Abaqus/CAE .msg file.

Attributes **increment.step_time** (*float*): Step time fraction of the increment [optional].

increment.start (*int*): Line number corresponding to the first line of the increment.

increment.stop (*int*): Line number corresponding to the last line of the increment.

increment.fileName (*str*): path to the *.msg* file including the file name and extension.

increment.outInst (*list*): *data* objects with matrix/arrays from the *.msg* file corresponding to the increment found using the *increment.find_lineNumbers()* and *increment.find_data()* methods.

increment.data (*dict*): Data from the increment organized into a dict. Keys of the dict indicate the keyword used to find the data and the Values of the dict carry lists of corresponding *data* objects.

Example

If the following variables are written to filename.msg file:

```
VarKey1 = 1.000000 2.000000
```

```
VarKey2 = 1.000000 2.000000 3.000000
4.000000
```

```
VarKey1 = 1.100000 2.100000
```

```
27 VarKey2 = 1.100000 2.100000 3.100000
28 4.100000
```

```
50 VarKey1 = 1.200000 2.200000
```

```
53 VarKey2 = 1.200000 2.200000 3.200000
54 4.200000
```

```
57 VarKey1 = 1.300000 2.300000
```

```
60 VarKey2 = 1.300000 2.300000 3.300000
61 4.300000
```

where *VarKey1* is the keyword for a variable *Var1* which is a column matrix and *VarKey2* represents a square matrix *Var2*.

$$Var1 = \begin{bmatrix} \# \\ \# \end{bmatrix}$$

$$Var2 = \begin{bmatrix} \# & \# \\ \# & \# \end{bmatrix}$$

To fetch these variables printed between lines 22 and 58, initialize an *increment* class instance.

```
incInst = increment()
incInst.fileName = "filename.msg"
incInst.start = 22
incInst.stop = 58
```

Further, find *Var1* using `VarKey1` keyword and create a list of *data* class instances of the three occurances. Here, the matrix elements for *Var1* are printed on a single line and the matrix is a column matrix with two rows, so use execute the *increment.find_lineNumbers()* as follows:

```
incInst.find_lineNumbers("VarKey1", 1, (2,1))
```

This appends *data* class instance of the three occurances of `VarKey1` to *increment.outInst* for example, details of the 2nd occurance can be extracted as follows:

```
print(incInst.outInst[1].type, incInst.outInst[1].start, incInst.outInst[1].stop,␣
↪incInst.outInst[1].shape)
```

Output

```
VarKey1 50 50 (2,1)
```

Append the occurances of the square matrix *Var2* of dimensions 2x2 printed to two consecutive lines to the *increment.outInst* list.

```
incInst.find_lineNumbers("VarKey2", 2, (2,2))
```

This appends *data* class instance of the two occurances of `VarKey2` to *increment.outInst* for example, details of the 1st occurance of *Var2* which is 4th in the list can be extracted as follows:

```
print(incInst.outInst[3].type, incInst.outInst[3].start, incInst.outInst[3].stop,␣
↪incInst.outInst[3].shape)
```

Output

```
VarKey2 27 28 (2,2)
```

Finally fetch the values corresponding to the five selected matrices using *increment.find_data()*.

```
incInst.find_data()

# First occurence of VarKey2
print(incInst.outInst[3].value)
print(incInst.data['VarKey2'][0].value)

# Third occurence of VarKey1
print(incInst.outInst[2].value)
print(incInst.data['VarKey1'][2].value)
```

Output

```
[[1.100000, 3.100000],
[2.100000, 4.100000]]
[[1.100000, 3.100000],
[2.100000, 4.100000]]
[[1.300000],
[2.300000]]
[[1.300000],
[2.300000]]
```

Metadata

Environment

Python

Version

v1.0.0

Date

2022-03-06

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

Methods Summary

find_data()	**Fetch values of the matrices/arrays corresponding to** *data* **class instances in** *increment.outInst*.
find_lineNumbers(key, length, shape)	**Find a keyword and create** *data* **class instance for each occurance of the keyword between** *increment.start* **and** *increment.stop* **in the** `.msg` **file. The** *data* **instances are appended to** `increment.outInst` **attribute.** (See *increment* class for an example.)

Methods Documentation

find_data()

Fetch values of the matrices/arrays corresponding to *data* **class instances in** *increment.outInst*. (See *increment* class for an example.)

Note: This funciton only fetches and reshapes the values of *data* instances. *data* instances must have already been defined using *increment.find_lineNumbers()* method before executing this method.

Metadata

Environment

Python

Version

v1.0.0

Date

2022-03-06

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

`find_lineNumbers`(*key*, *length*, *shape*)

Find a keyword and create *data* **class instance for each occurance of the keyword between** *increment.start* **and** *increment.stop* **in the** `.msg` **file. The** *data* **instances are appended to** `increment.outInst` **attribute.** (See *increment* class for an example.)

Parameters **key** (*str*): Keyword to find the matrix/array of interest. Ideally, this should indicate the vairable name of the matrix/array printed to the *.msg* file.

length (*int*): Number of lines the matrix extends to, starting from the line containing the Keyword.

shape (*tuple*): Shape of the matrix.

shape[0] (*int*): number of rows.

shape[1] (*int*): number of columns.

Tip: This method can be executed multiple times with different keywords. Since the *data* class instances store the keywords and the line numbers, the values can be systematically retrived. See *increment.find_data()*.

Note: This funciton only creates *data* instances and finds line numbers corresponding to required matrix. *data.findValue()* has to be executed to fetch the actual values. Use *increment.find_data()* to do this automatically for all the instances in `increment.outInst`.

Warning: *increment* attributes *increment.type*, *increment.start*, and *increment.stop* must be defined to execute the *increment.find_lineNumbers()* method.

Metadata

Environment

Python

Version

v1.0.0

Date

2022-03-06

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

3.3.2 Abaqus Python Modules for use With Abaqus/CAE

czmtestkit.abaqus_modules Package

Functions

ADCB(dict)	**Create and submit Asymmetric Double Cantilever Beam (ADCB) test with plain strain boundary conditions using Abaqus/CAE.**
ADCB2(dict)	**Create and submit Asymmetric Double Cantilever Beam (ADCB) test with plain strain boundary conditions using Abaqus/CAE.**
ADCB2powerLaw(dict)	**Create and submit Asymmetric Double Cantilever Beam (ADCB) test with plain strain boundary conditions using Abaqus/CAE.**
ASLB(dict)	**Create and submit Asymmetric Single Leg Bending (ASLB) test with plain strain boundary conditions using Abaqus/CAE.**
ASLB2(dict)	**Create and submit Asymmetric Single Leg Bending (ASLB) test with plain strain boundary conditions using Abaqus/CAE.**
ReDefCE(Name, CzMat, CzIntMat)	**Redefine abaqus cohesive sections in the `.inp` file to user defined elements.**
historyOutput(dict)	**Fetch history output from `.odb` file and save to `.csv` file.**

czmtestkit.abaqus_modules.ADCB

czmtestkit.abaqus_modules.**ADCB**(*dict*)

Create and submit Asymmetric Double Cantilever Beam (ADCB) test with plain strain boundary conditions using Abaqus/CAE.

Note: The function *ADCB()* is only different from *ADCB2()* in the material definition of the bulk. While *ADCB()* defines both top and bottom adherands or plies in Fig. 3.4 with the same engineering constants, *ADCB2()* defines these regions separately.

The ADCB specimen with geometry from Fig. 3.4 is generated with unit width ($B = 1$). The mixed-mode damage is modelled using the *BK criteria* Additionally, along with boundary conditions from Fig. 3.4, the translation along *E2* of all the nodes on faces perpendicular to *E2* are fixed to replicate plain-strain boundary conditions. Further, the displacement on the load edge is applied in an implicit dynamic step with nonlinear geometry option turned on.

Parameters

dict (*dict*):

'JobID' name of the `.odb` file.

'Length' Length of the specimen $2L$.

'tTop' thickness of the top adherand/ply h_u.

'tBot' thickness of the bottom adherand/ply h_l.

'tCz' thickness of the cohesive zone t.

'Crack' Crack length a_0.

'DensityBulk' Density of the bulk material.

'E' Tuple of engineering constants for the elastic behaviour of the bulk. (E1, E2, E3, ν_{12}, ν_{13}, ν_{23}, G12, G13, G23)

'DensityCz' Density of the cohesive zone

'StiffnessCz' Element stiffness or penality stiffness K.

'GcNormal' Fracture toughness in opening mode G_{C_I}. See Fig. 3.5

'GcShear' Fracture toughness in shear mode $G_{C_{sh}}$. See Fig. 3.5

'gFailureNormal' Final or failure displacement gap in opening mode Δ_I^f. See Fig. 3.5

'gFailureShear' Final or failure displacement gap in shear mode Δ_{sh}^f. See Fig. 3.5

'bkPower' η of the *BK criteria*.

'MeshCrack' Mesh size of edges along direction *E1* in the crack.

'MeshX' Mesh size of edges along direction *E1* ahead of crack tip.

'MeshZ' Mesh size of edges along direction *E3*.

'Displacement' Magnitude of the displacement to be applied along *U3* at the load edge.

'nCpu' Number of CPUs to be used when submitting the job.

'nGpu' Number of GPUs to be used when submitting the job.

'userSub' Dictionary with user subroutine specifications

'type' `'None'`: Energy based linear softening traction separation law as implemented by Abaqus/CAE is used for cohesive elements.

`'UEL'`: Redefines the cohesive elements to user elements using *ReDefCE()* and submits with the subroutine from `dict['userSub']['path']`.

```
ReDefCE(JobID+'.inp',
    [StiffnessCz,
        NominalNormal,
        NominalShear,
        GcNormal,
        GcShear,
        bkPower],
    userSub['intProp'])
```

'path' Path to the fortran based user subroutine (`.for` file).

'intProp' *int* list of element properties.

'submit' `True`: the Abaqus/CAE job is submitted.

`False`: the input file `.inp` is generated but the job is not submitted.

Warning: The input parameters should be consistent in their units of measurement. Following are some commonly used groups of units in engineering:

Table 3.6: Consistent set of units [4].

MASS	LENGTH	TIME	FORCE	STRESS	ENERGY
kg	m	s	N	Pa	J
kg	mm	ms	kN	GPa	kN-mm
g	mm	ms	N	MPa	N-mm

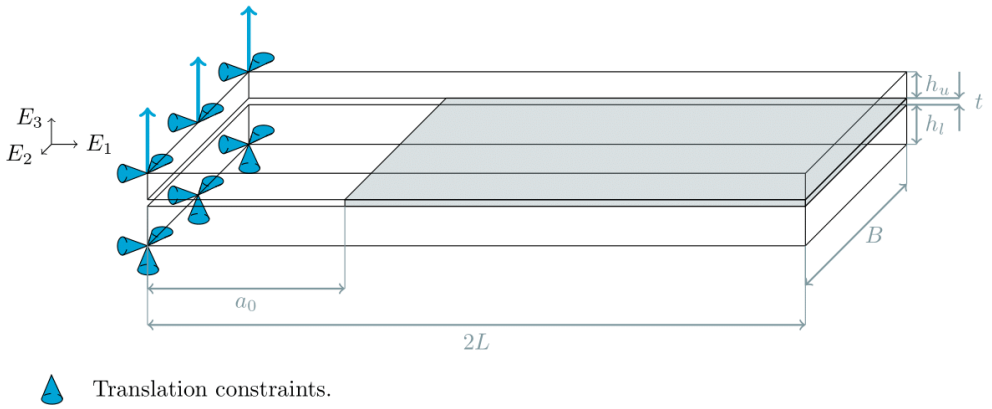


Fig. 3.4: **ADCB schematic** *[1]*.
*Here, the translation degrees of freedom parallel to the axis of the `blue* cones are fixed. Additionally, the shaded region represents the cohesive zone interface while the unshaded region represents the bulk adherands or plies.`

References:

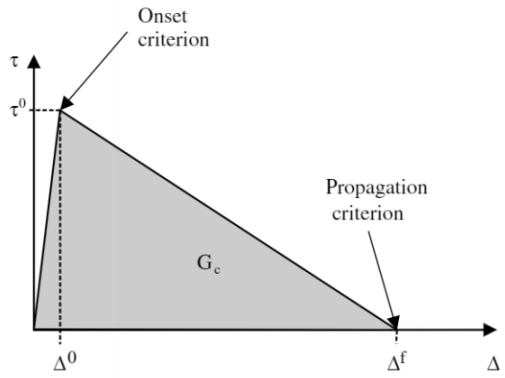


Fig. 3.5: **Bilinear Traction Separation Law** or the linear softening law *[2]*.
Interfaces tend to have different properties for opening (mode-I) and shear modes (mode-II and mode-III) in Fig. 3.6 *, resulting in different traction separation laws represented above using subscripts `I* for opening mode and *sh* for shear modes for the parameters.`

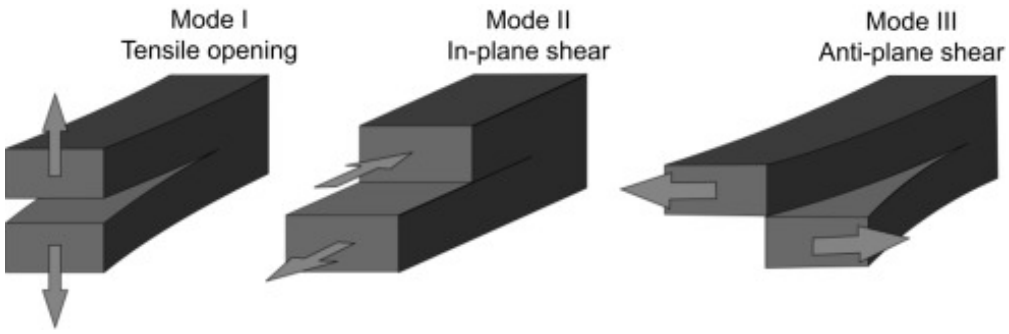


Fig. 3.6: **Fracture Modes** *[3]*.

1) Mudunuru, N. (2022, March 30). Finite Element Model For Interfaces In Compatibilized Polymer Blends. TU Delft Education Repositories. Retrieved on April 21, 2022, from http://resolver.tudelft.nl/uuid:88140513-120d-4a34-b893-b84908fe2373

2) Turon, A., Camanho, P., Costa, J., & Davila, C. (2006). A damage model for the simulation of delamination in advanced composites under variable-mode loading. Mechanics of Materials, 38(11), 1072–1089. https://doi.org/10.1016/j.mechmat.2005.10.003

3) Oterkus, E., Diyaroglu, C., de Meo, D., & Allegri, G. (2016). Fracture modes, damage tolerance and failure mitigation in marine composites. Marine Applications of Advanced Fibre-Reinforced Composites, 79–102. https://doi.org/10.1016/b978-1-78242-250-1.00004-1

4) LS-Dyna. (n.d.). Consistent units. Retrieved April 21, 2022, from https://www.dynasupport.com/howtos/general/consistent-units

5) Benzeggagh, M., & Kenane, M. (1996). Measurement of mixed-mode delamination fracture toughness of unidirectional glass/epoxy composites with mixed-mode bending apparatus. Composites Science and Technology, 56(4), 439–449. https://doi.org/10.1016/0266-3538(96)00005-x

Metadata

Environment

Abaqus/CAE

Version

v1.0.0

Date

2021-12-29

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

czmtestkit.abaqus_modules.ADCB2

`czmtestkit.abaqus_modules.`**ADCB2**(*dict*)

Create and submit Asymmetric Double Cantilever Beam (ADCB) test with plain strain boundary conditions using Abaqus/CAE.

Note: The function *ADCB()* is only different from *ADCB2()* in the material definition of the bulk. While *ADCB()* defines both top and bottom adherands or plies in Fig. 3.3 with the same engineering constants, *ADCB2()* defines these regions separately.

The ADCB specimen with geometry from Fig. 3.3 is generated with unit width ($B = 1$). The mixed-mode damage is modelled using the *BK criteria* Additionally, along with boundary conditions from Fig. 3.3, the translation along *E2* of all the nodes on faces perpendicular to *E2* are fixed to replicate plain-strain boundary conditions. Further, the displacement on the load edge is applied in an implicit dynamic step with nonlinear geometry option turned on.

Parameters

dict (*dict*):

'JobID' name of the `.odb` file.

'Length' Length of the specimen $2L$.

'tTop' thickness of the top adherand/ply h_u.

'tBot' thickness of the bottom adherand/ply h_l.

'tCz' thickness of the cohesive zone t.

'Crack' Crack length a_0.

'DensityBulkBot' Density of the bottom adherand/ply.

'DensityBulkTop' Density of the top adherand/ply.

'EBot' Tuple of engineering constants for the elastic behaviour of the bottom adherand/ply. (E1, E2, E3, ν_{12}, ν_{13}, ν_{23}, G12, G13, G23)

'ETot' Tuple of engineering constants for the elastic behaviour of the top adherand/ply. (E1, E2, E3, ν_{12}, ν_{13}, ν_{23}, G12, G13, G23)

'DensityCz' Density of the cohesive zone

'StiffnessCz' Element stiffness or penality stiffness K.

'GcNormal' Fracture toughness in opening mode G_{C_I}. See Fig. 3.8

'GcShear' Fracture toughness in shear mode $G_{C_{sh}}$. See Fig. 3.8

'gFailureNormal' Final or failure displacement gap in opening mode Δ_I^f. See Fig. 3.8

'gFailureShear' Final or failure displacement gap in shear mode Δ_{sh}^f. See Fig. 3.8

'bkPower' η of the *BK criteria*.

'MeshCrack' Mesh size of edges along direction *E1* in the crack.

'MeshX' Mesh size of edges along direction *E1* ahead of crack tip.

'MeshZ' Mesh size of edges along direction *E3*.

'Displacement' Magnitude of the displacement to be applied along *U3* at the load edge.

'nCpu' Number of CPUs to be used when submitting the job.

'nGpu' Number of GPUs to be used when submitting the job.

'userSub' Dictionary with user subroutine specifications

'type' `'None'`: Energy based linear softening traction separation law as implemented by Abaqus/CAE is used for cohesive elements.

`'UEL'`: Redefines the cohesive elements to user elements using *ReDefCE()* and submits with the subroutine from `dict['userSub']['path']`.

```
ReDefCE(JobID+'.inp',
    [StiffnessCz,
        NominalNormal,
        NominalShear,
        GcNormal,
        GcShear,
        bkPower],
    userSub['intProp'])
```

'path' Path to the fortran based user subroutine (`.for` file).

'intProp' *int* list of element properties.

'submit' `True`: the Abaqus/CAE job is submitted.

`False`: the input file `.inp` is generated but the job is not submitted.

Warning: The input parameters should be consistent in their units of measurement. Following are some commonly used groups of units in engineering:

Table 3.7: Consistent set of units [4].

MASS	LENGTH	TIME	FORCE	STRESS	ENERGY
kg	m	s	N	Pa	J
kg	mm	ms	kN	GPa	kN-mm
g	mm	ms	N	MPa	N-mm

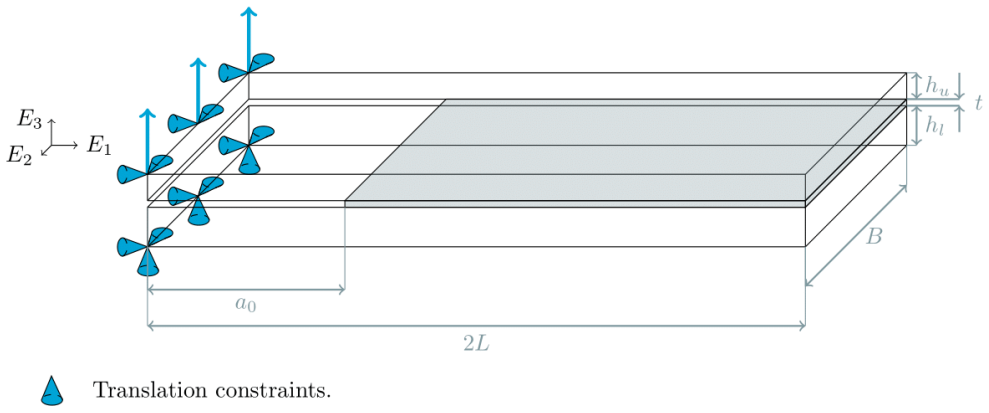


Fig. 3.7: **ADCB schematic** *[1]*.
*Here, the translation degrees of freedom parallel to the axis of the `blue* cones are fixed. Additionally, the shaded region represents the cohesive zone interface while the unshaded region represents the bulk adherands or plies.`

Tip: Analytical results for this test using Timoshenko beam theory and Castigliano theorem as described in appendix B of the master thesis *[1]* can be obtained using methods of the *czmtestkit.py_modules.ADCB* class.

References:

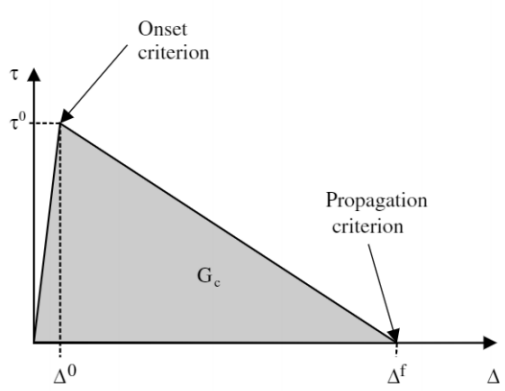


Fig. 3.8: **Bilinear Traction Separation Law** or the linear softening law *[2]*.
Interfaces tend to have different properties for opening (mode-I) and shear modes (mode-II and mode-III) in Fig. 3.9 *, resulting in different traction separation laws represented above using subscripts `I* for opening mode and *sh* for shear modes for the parameters.`

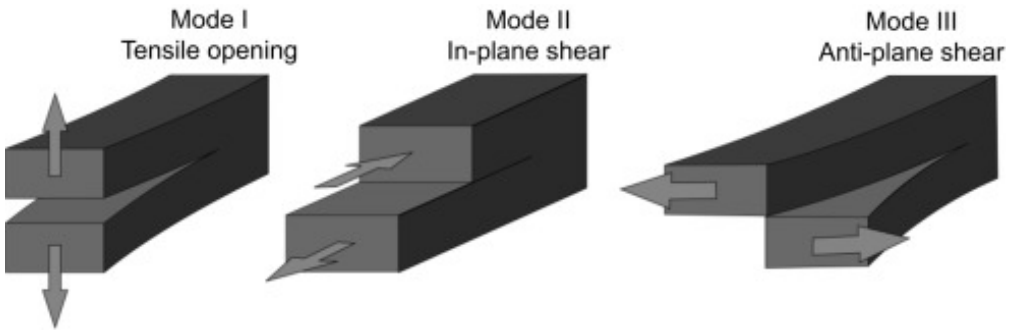


Fig. 3.9: **Fracture Modes** *[3]*.

1) Mudunuru, N. (2022, March 30). Finite Element Model For Interfaces In Compatibilized Polymer Blends. TU Delft Education Repositories. Retrieved on April 21, 2022, from http://resolver.tudelft.nl/uuid:88140513-120d-4a34-b893-b84908fe2373

2) Turon, A., Camanho, P., Costa, J., & Davila, C. (2006). A damage model for the simulation of delamination in advanced composites under variable-mode loading. Mechanics of Materials, 38(11), 1072–1089. https://doi.org/10.1016/j.mechmat.2005.10.003

3) Oterkus, E., Diyaroglu, C., de Meo, D., & Allegri, G. (2016). Fracture modes, damage tolerance and failure mitigation in marine composites. Marine Applications of Advanced Fibre-Reinforced Composites, 79–102. https://doi.org/10.1016/b978-1-78242-250-1.00004-1

4) LS-Dyna. (n.d.). Consistent units. Retrieved April 21, 2022, from https://www.dynasupport.com/howtos/general/consistent-units

5) Benzeggagh, M., & Kenane, M. (1996). Measurement of mixed-mode delamination fracture toughness of unidirectional glass/epoxy composites with mixed-mode bending apparatus. Composites Science and Technology, 56(4), 439–449. https://doi.org/10.1016/0266-3538(96)00005-x

Metadata

Environment

Abaqus/CAE

Version

v1.0.0

Date

2021-12-29

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

czmtestkit.abaqus_modules.ADCB2powerLaw

czmtestkit.abaqus_modules.**ADCB2powerLaw**(*dict*)

Create and submit Asymmetric Double Cantilever Beam (ADCB) test with plain strain boundary conditions using Abaqus/CAE.

Note: The function *ADCB2powerLaw()* is only different from *ADCB2()* in the mixed-mode criteria of the damage law. While *ADCB2powerLaw()* uses the power law, *ADCB2()* uses the *BK criteria*.

The ADCB specimen with geometry from Fig. 3.10 is generated with unit width ($B = 1$). The mixed-mode damage is modelled using the power law Additionally, along with boundary conditions from Fig. 3.10, the translation along *E2* of all the nodes on faces perpendicular to *E2* are fixed to replicate plain-strain boundary conditions. Further, the displacement on the load edge is applied in an implicit dynamic step with nonlinear geometry option turned on.

Parameters

dict (*dict*):

'JobID' name of the `.odb` file.

'Length' Length of the specimen $2L$.

'tTop' thickness of the top adherand/ply h_u.

'tBot' thickness of the bottom adherand/ply h_l.

'tCz' thickness of the cohesive zone t.

'Crack' Crack length a_0.

'DensityBulkBot' Density of the bottom adherand/ply.

'DensityBulkTop' Density of the top adherand/ply.

'EBot' Tuple of engineering constants for the elastic behaviour of the bottom adherand/ply. (E1, E2, E3, ν_{12}, ν_{13}, ν_{23}, G12, G13, G23)

'ETot' Tuple of engineering constants for the elastic behaviour of the top adherand/ply. (E1, E2, E3, ν_{12}, ν_{13}, ν_{23}, G12, G13, G23)

'DensityCz' Density of the cohesive zone

'StiffnessCz' Element stiffness or penality stiffness K.

'GcNormal' Fracture toughness in opening mode G_{C_I}. See Fig. 3.11

'GcShear' Fracture toughness in shear mode $G_{C_{sh}}$. See Fig. 3.11

'gFailureNormal' Final or failure displacement gap in opening mode Δ_I^f. See Fig. 3.11

'gFailureShear' Final or failure displacement gap in shear mode Δ_{sh}^f. See Fig. 3.11

'powerLaw' γ of the power law.

'MeshCrack' Mesh size of edges along direction *E1* in the crack.

'MeshX' Mesh size of edges along direction *E1* ahead of crack tip.

'MeshZ' Mesh size of edges along direction *E3*.

'Displacement' Magnitude of the displacement to be applied along *U3* at the load edge.

'nCpu' Number of CPUs to be used when submitting the job.

'nGpu' Number of GPUs to be used when submitting the job.

'userSub' Dictionary with user subroutine specifications

'type' `'None'`: Energy based linear softening traction separation law as implemented by Abaqus/CAE is used for cohesive elements.

`'UEL'`: Redefines the cohesive elements to user elements using *ReDefCE()* and submits with the subroutine from `dict['userSub']['path']`.

```
ReDefCE(JobID+'.inp',
    [StiffnessCz,
        NominalNormal,
        NominalShear,
        GcNormal,
        GcShear,
        bkPower],
    userSub['intProp'])
```

'path' Path to the fortran based user subroutine (`.for` file).

'intProp' *int* list of element properties.

'submit' `True`: the Abaqus/CAE job is submitted.

`False`: the input file `.inp` is generated but the job is not submitted.

Warning: The input parameters should be consistent in their units of measurement. Following are some commonly used groups of units in engineering:

Table 3.8: Consistent set of units [4].

MASS	LENGTH	TIME	FORCE	STRESS	ENERGY
kg	m	s	N	Pa	J
kg	mm	ms	kN	GPa	kN-mm
g	mm	ms	N	MPa	N-mm

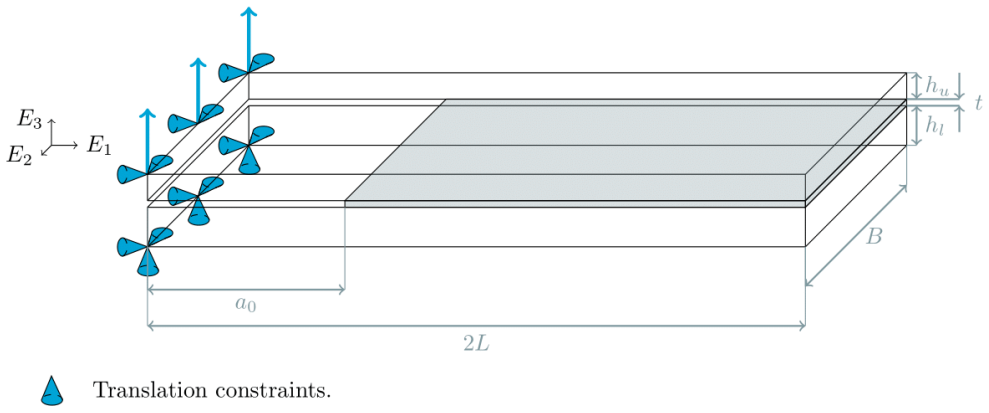


Fig. 3.10: **ADCB schematic** *[1]*.
*Here, the translation degrees of freedom parallel to the axis of the `blue* cones are fixed. Additionally, the shaded region represents the cohesive zone interface while the unshaded region represents the bulk adherands or plies.`

Tip: Analytical results for this test using Timoshenko beam theory and Castigliano theorem as described in appendix B of the master thesis *[1]* can be obtained using methods of the *czmtestkit.py_modules.ADCB* class.

References:

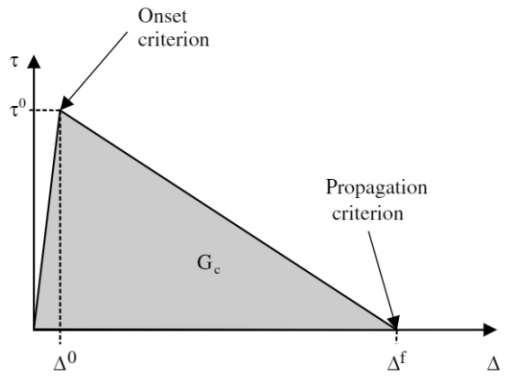


Fig. 3.11: **Bilinear Traction Separation Law** or the linear softening law *[2]*.
Interfaces tend to have different properties for opening (mode-I) and shear modes (mode-II and mode-III) in Fig. 3.12 *, resulting in different traction separation laws represented above using subscripts `I* for opening mode and *sh* for shear modes for the parameters.`

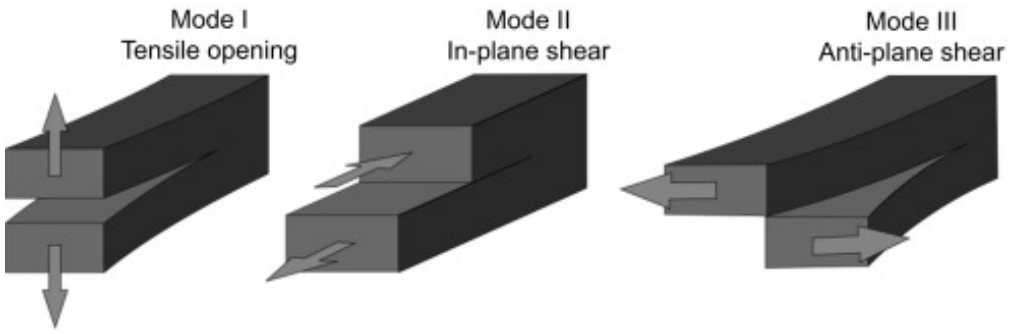


Fig. 3.12: **Fracture Modes** *[3]*.

1) Mudunuru, N. (2022, March 30). Finite Element Model For Interfaces In Compatibilized Polymer Blends. TU Delft Education Repositories. Retrieved on April 21, 2022, from http://resolver.tudelft.nl/uuid:88140513-120d-4a34-b893-b84908fe2373

2) Turon, A., Camanho, P., Costa, J., & Davila, C. (2006). A damage model for the simulation of delamination in advanced composites under variable-mode loading. Mechanics of Materials, 38(11), 1072–1089. https://doi.org/10.1016/j.mechmat.2005.10.003

3) Oterkus, E., Diyaroglu, C., de Meo, D., & Allegri, G. (2016). Fracture modes, damage tolerance and failure mitigation in marine composites. Marine Applications of Advanced Fibre-Reinforced Composites, 79–102. https://doi.org/10.1016/b978-1-78242-250-1.00004-1

4) LS-Dyna. (n.d.). Consistent units. Retrieved April 21, 2022, from https://www.dynasupport.com/howtos/general/consistent-units

5) Benzeggagh, M., & Kenane, M. (1996). Measurement of mixed-mode delamination fracture toughness of unidirectional glass/epoxy composites with mixed-mode bending apparatus. Composites Science and Technology, 56(4), 439–449. https://doi.org/10.1016/0266-3538(96)00005-x

Metadata

Environment

Abaqus/CAE

Version

v1.0.0

Date

2021-12-29

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

czmtestkit.abaqus_modules.ASLB

czmtestkit.abaqus_modules.**ASLB**(*dict*)

Create and submit Asymmetric Single Leg Bending (ASLB) test with plain strain boundary conditions using Abaqus/CAE.

Note: The function *ASLB()* is only different from *ASLB2()* in the material definition of the bulk. While *ASLB()* defines both top and bottom adherands or plies in Fig. 3.13 with the same engineering constants, *ASLB2()* defines these regions separately.

The ASLB specimen with geometry from Fig. 3.13 is generated with unit width ($B = 1$). The mixed-mode damage is modelled using the *BK criteria* Additionally, along with boundary conditions from Fig. 3.13, the translation along *E2* of all the nodes on faces perpendicular to *E2* are fixed to replicate plain-strain boundary conditions. Further, the displacement on the load edge is applied in an implicit dynamic step with nonlinear geometry option turned on.

Parameters

dict (*dict*):

'JobID' name of the `.odb` file.

'Length' Length of the specimen $2L$.

'tTop' thickness of the top adherand/ply h_u.

'tBot' thickness of the bottom adherand/ply h_l.

'tCz' thickness of the cohesive zone t.

'Crack' Crack length a_0.

'DensityBulk' Density of the bulk material.

'E' Tuple of engineering constants for the elastic behaviour of the bulk. (E1, E2, E3, ν_{12}, ν_{13}, ν_{23}, G12, G13, G23)

'DensityCz' Density of the cohesive zone

'StiffnessCz' Element stiffness or penality stiffness K.

'GcNormal' Fracture toughness in opening mode G_{C_I}. See Fig. 3.14

'GcShear' Fracture toughness in shear mode $G_{C_{sh}}$. See Fig. 3.14

'gFailureNormal' Final or failure displacement gap in opening mode Δ_I^f. See Fig. 3.14

'gFailureShear' Final or failure displacement gap in shear mode Δ_{sh}^f. See Fig. 3.14

'bkPower' η of the *BK criteria*.

'MeshCrack' Mesh size of edges along direction *E1* in the crack.

'MeshX' Mesh size of edges along direction *E1* ahead of crack tip.

'MeshZ' Mesh size of edges along direction *E3*.

'Displacement' Magnitude of the displacement to be applied along *U3* at the load edge.

'nCpu' Number of CPUs to be used when submitting the job.

'nGpu' Number of GPUs to be used when submitting the job.

'userSub' Dictionary with user subroutine specifications

'type' `'None'`: Energy based linear softening traction separation law as implemented by Abaqus/CAE is used for cohesive elements.

`'UEL'`: Redefines the cohesive elements to user elements using *`ReDefCE()`* and submits with the subroutine from `dict['userSub']['path']`.

```
ReDefCE(JobID+'.inp',
    [StiffnessCz,
        NominalNormal,
        NominalShear,
        GcNormal,
        GcShear,
        bkPower],
    userSub['intProp'])
```

'path' Path to the fortran based user subroutine (`.for` file).

'intProp' *int* list of element properties.

'submit' `True`: the Abaqus/CAE job is submitted.

`False`: the input file `.inp` is generated but the job is not submitted.

Warning: The input parameters should be consistent in their units of measurement. Following are some commonly used groups of units in engineering:

Table 3.9: Consistent set of units [4].

MASS	LENGTH	TIME	FORCE	STRESS	ENERGY
kg	m	s	N	Pa	J
kg	mm	ms	kN	GPa	kN-mm
g	mm	ms	N	MPa	N-mm

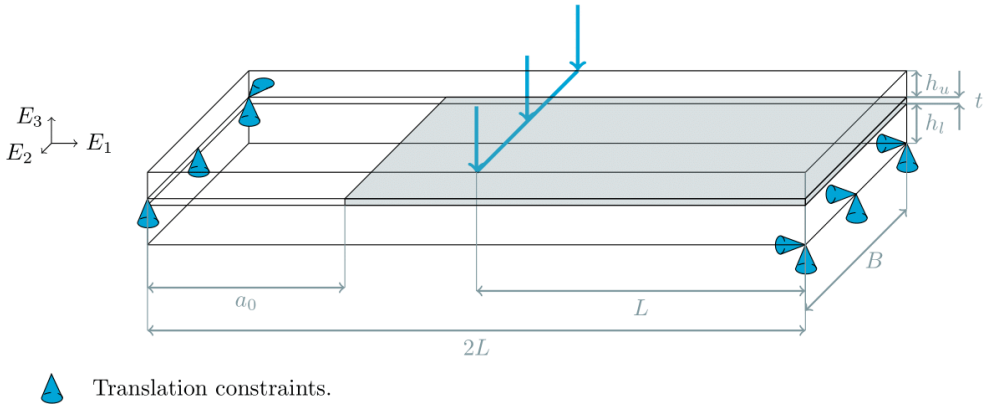


Fig. 3.13: **ASLB schematic** *[1]*.
*Here, the translation degrees of freedom parallel to the axis of the `blue* cones are fixed. Additionally, the shaded region represents the cohesive zone interface while the unshaded region represents the bulk adherands or plies.`

Tip: Analytical results for this test using Timoshenko beam theory and Castigliano theorem as described in appendix B of the master thesis *[1]* can be obtained using methods of the *`czmtestkit.py_modules.ASLB`* class.

References:

1) Mudunuru, N. (2022, March 30). Finite Element Model For Interfaces In Compatibilized Polymer Blends. TU Delft Education Repositories. Retrieved on April 21, 2022, from http://resolver.tudelft.nl/uuid:88140513-120d-4a34-b893-b84908fe2373

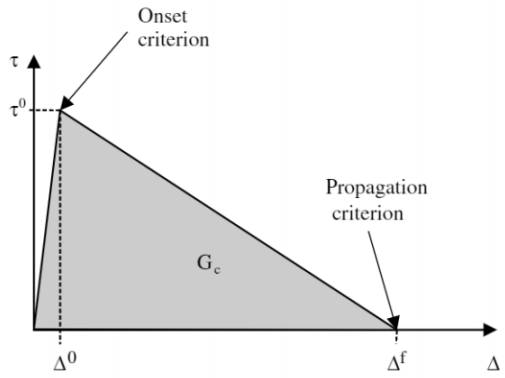


Fig. 3.14: **Bilinear Traction Separation Law** or the linear softening law *[2]*.
Interfaces tend to have different properties for opening (mode-I) and shear modes (mode-II and mode-III) in Fig. 3.15 *, resulting in different traction separation laws represented above using subscripts `I* for opening mode and *sh* for shear modes for the parameters.`

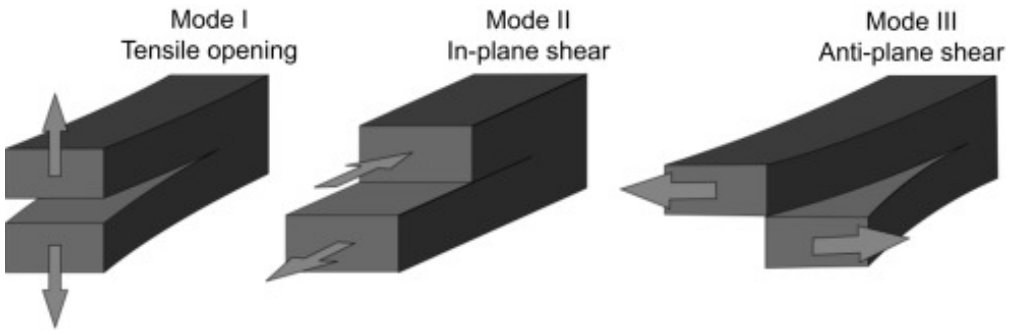


Fig. 3.15: **Fracture Modes** *[3]*.

2) Turon, A., Camanho, P., Costa, J., & Davila, C. (2006). A damage model for the simulation of delamination in advanced composites under variable-mode loading. Mechanics of Materials, 38(11), 1072–1089. https://doi.org/10.1016/j.mechmat.2005.10.003

3) Oterkus, E., Diyaroglu, C., de Meo, D., & Allegri, G. (2016). Fracture modes, damage tolerance and failure mitigation in marine composites. Marine Applications of Advanced Fibre-Reinforced Composites, 79–102. https://doi.org/10.1016/b978-1-78242-250-1.00004-1

4) LS-Dyna. (n.d.). Consistent units. Retrieved April 21, 2022, from https://www.dynasupport.com/howtos/general/consistent-units

5) Benzeggagh, M., & Kenane, M. (1996). Measurement of mixed-mode delamination fracture toughness of unidirectional glass/epoxy composites with mixed-mode bending apparatus. Composites Science and Technology, 56(4), 439–449. https://doi.org/10.1016/0266-3538(96)00005-x

Metadata

Environment

Abaqus/CAE

Version

v1.0.0

Date

2021-12-29

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

czmtestkit.abaqus_modules.ASLB2

czmtestkit.abaqus_modules.**ASLB2**(*dict*)

Create and submit Asymmetric Single Leg Bending (ASLB) test with plain strain boundary conditions using Abaqus/CAE.

Note: The function *ASLB()* is only different from *ASLB2()* in the material definition of the bulk. While *ASLB()* defines both top and bottom adherands or plies in Fig. 3.16 with the same engineering constants, *ASLB2()* defines these regions separately.

The ASLB specimen with geometry from Fig. 3.16 is generated with unit width ($B = 1$). The mixed-mode damage is modelled using the *BK criteria* Additionally, along with boundary conditions from Fig. 3.16, the translation along *E2* of all the nodes on faces perpendicular to *E2* are fixed to replicate plain-strain boundary

conditions. Further, the displacement on the load edge is applied in an implicit dynamic step with nonlinear geometry option turned on.

Parameters

dict (*dict*):

'JobID' name of the `.odb` file.

'Length' Length of the specimen $2L$.

'tTop' thickness of the top adherand/ply h_u.

'tBot' thickness of the bottom adherand/ply h_l.

'tCz' thickness of the cohesive zone t.

'Crack' Crack length a_0.

'DensityBulkBot' Density of the bottom adherand/ply.

'DensityBulkTop' Density of the top adherand/ply.

'EBot' Tuple of engineering constants for the elastic behaviour of the bottom adherand/ply. (E1, E2, E3, ν_{12}, ν_{13}, ν_{23}, G12, G13, G23)

'ETot' Tuple of engineering constants for the elastic behaviour of the top adherand/ply. (E1, E2, E3, ν_{12}, ν_{13}, ν_{23}, G12, G13, G23)

'DensityCz' Density of the cohesive zone

'StiffnessCz' Element stiffness or penality stiffness K.

'GcNormal' Fracture toughness in opening mode G_{C_I}. See Fig. 3.17

'GcShear' Fracture toughness in shear mode $G_{C_{sh}}$. See Fig. 3.17

'gFailureNormal' Final or failure displacement gap in opening mode Δ_I^f. See Fig. 3.17

'gFailureShear' Final or failure displacement gap in shear mode Δ_{sh}^f. See Fig. 3.17

'bkPower' η of the *BK criteria*.

'MeshCrack' Mesh size of edges along direction *E1* in the crack.

'MeshX' Mesh size of edges along direction *E1* ahead of crack tip.

'MeshZ' Mesh size of edges along direction *E3*.

'Displacement' Magnitude of the displacement to be applied along *U3* at the load edge.

'nCpu' Number of CPUs to be used when submitting the job.

'nGpu' Number of GPUs to be used when submitting the job.

'userSub' Dictionary with user subroutine specifications

'type' `'None'`: Energy based linear softening traction separation law as implemented by Abaqus/CAE is used for cohesive elements.

`'UEL'`: Redefines the cohesive elements to user elements using *ReDefCE()* and submits with the subroutine from `dict['userSub']['path']`.

```
ReDefCE(JobID+'.inp',
    [StiffnessCz,
        NominalNormal,
        NominalShear,
        GcNormal,
        GcShear,
        bkPower],
    userSub['intProp'])
```

'path' Path to the fortran based user subroutine (`.for` file).

'intProp' *int* list of element properties.

'submit' `True`: the Abaqus/CAE job is submitted.

`False`: the input file `.inp` is generated but the job is not submitted.

Warning: The input parameters should be consistent in their units of measurement. Following are some commonly used groups of units in engineering:

Table 3.10: Consistent set of units [4].

MASS	LENGTH	TIME	FORCE	STRESS	ENERGY
kg	m	s	N	Pa	J
kg	mm	ms	kN	GPa	kN-mm
g	mm	ms	N	MPa	N-mm

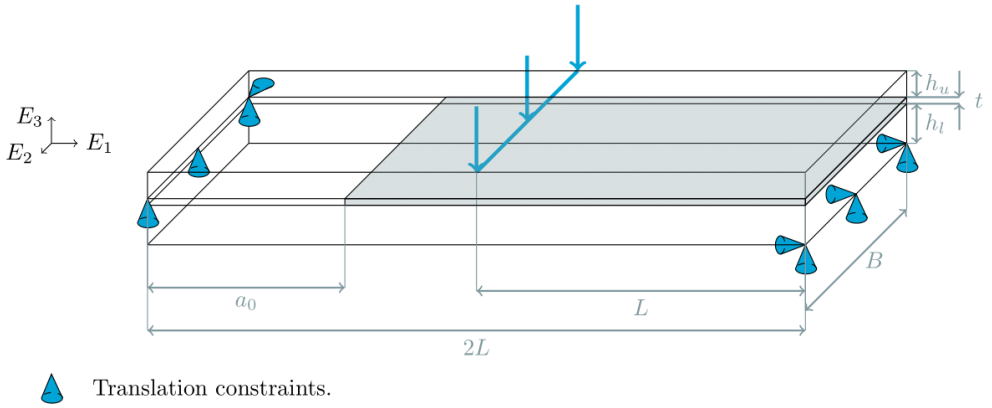


Fig. 3.16: **ASLB schematic** *[1]*.
*Here, the translation degrees of freedom parallel to the axis of the `blue* cones are fixed. Additionally, the shaded region represents the cohesive zone interface while the unshaded region represents the bulk adherands or plies.`

Tip: Analytical results for this test using Timoshenko beam theory and Castigliano theorem as described in appendix B of the master thesis *[1]* can be obtained using methods of the *`czmtestkit.py_modules.ASLB`* class.

References:

1) Mudunuru, N. (2022, March 30). Finite Element Model For Interfaces In Compatibilized Polymer Blends. TU Delft Education Repositories. Retrieved on April 21, 2022, from http://resolver.tudelft.nl/uuid:88140513-120d-4a34-b893-b84908fe2373

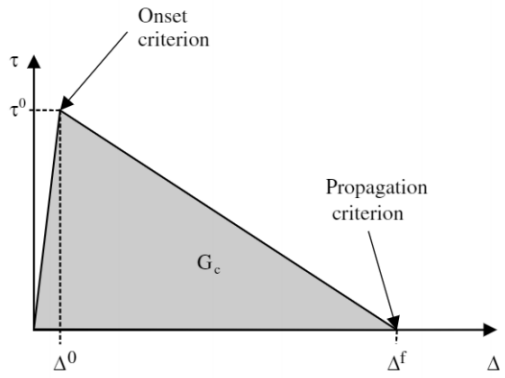


Fig. 3.17: **Bilinear Traction Separation Law** or the linear softening law *[2]*.
Interfaces tend to have different properties for opening (mode-I) and shear modes (mode-II and mode-III) in Fig. 3.18 *, resulting in different traction separation laws represented above using subscripts `I* for opening mode and *sh* for shear modes for the parameters.`

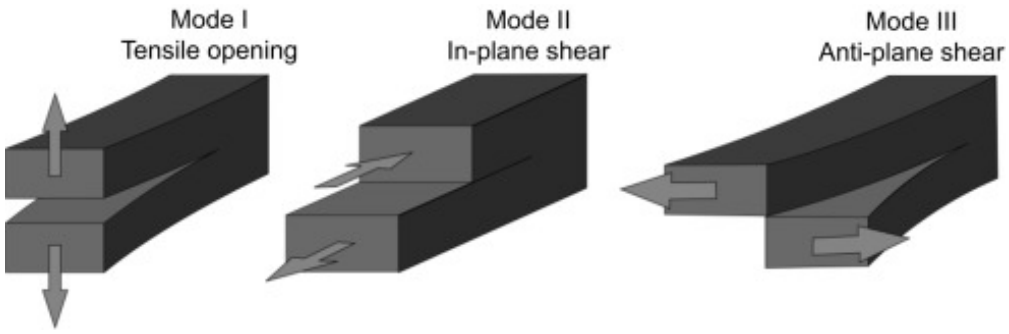


Fig. 3.18: **Fracture Modes** *[3]*.

2) Turon, A., Camanho, P., Costa, J., & Davila, C. (2006). A damage model for the simulation of delamination in advanced composites under variable-mode loading. Mechanics of Materials, 38(11), 1072–1089. https://doi.org/10.1016/j.mechmat.2005.10.003

3) Oterkus, E., Diyaroglu, C., de Meo, D., & Allegri, G. (2016). Fracture modes, damage tolerance and failure mitigation in marine composites. Marine Applications of Advanced Fibre-Reinforced Composites, 79–102. https://doi.org/10.1016/b978-1-78242-250-1.00004-1

4) LS-Dyna. (n.d.). Consistent units. Retrieved April 21, 2022, from https://www.dynasupport.com/howtos/general/consistent-units

5) Benzeggagh, M., & Kenane, M. (1996). Measurement of mixed-mode delamination fracture toughness of unidirectional glass/epoxy composites with mixed-mode bending apparatus. Composites Science and Technology, 56(4), 439–449. https://doi.org/10.1016/0266-3538(96)00005-x

Metadata

Environment

Abaqus/CAE

Version

v1.0.0

Date

2021-12-29

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

czmtestkit.abaqus_modules.ReDefCE

`czmtestkit.abaqus_modules.`**`ReDefCE`**(*Name*, *CzMat*, *CzIntMat*)

Redefine abaqus cohesive sections in the `.inp` file to user defined elements.

Parameters **Name** (*str*): `.inp` file name.

CzMat (*list*): *float* list of element properties for the user elements.

CzIntMat (*list*): *int* list of element properties for the user elements.

Example

Convert elements of `type = COH3D8` or `type = COH3D6` used with cohesive sections in Abaqus/CAE input file `fileName.inp`:

fileName.inp:

```
...
*Element, type=COH3D8
...
** Section: Section-2
*Cohesive Section, elset=Cz, material=Material-2, response=TRACTION SEPARATION
,
...
```

to user defined elements with only float variables as element properties:

```
ReDefCE('fileName', [1.000, 31.003, 6.7894], [])
```

fileName.inp:

```
...
-#  *Element, type=COH3D8
+#  *USER ELEMENT, NODES=8, Type= U1, PROPERTIES=3, COORDINATES=3,
+#   UNSYMM, VARIABLES=21
+#   1, 2, 3
+#  *UEL PROPERTY, elset=Cz
+#   1.000, 31.003, 6.7894
+#  *ELEMENT, TYPE=U1, elset=Cz
...
-#  ** Section: Section-2
-#  *Cohesive Section, elset=Cz, material=Material-2, response=TRACTION SEPARATION
-#  ,
+#  **** Section: Section-2
+#  ***Cohesive Section, elset=Cz, material=Material-2, response=TRACTION SEPARATION
+#  **,
...
```

where 3 in PROPERTIES=3 is the length of *CzMat* list, or user defined elements with float and int variables as element properties:

```
ReDefCE('fileName', [1.000, 31.003, 6.7894], [1, 2])
```

fileName.inp:

```
...
-#  *Element, type=COH3D8
+#  *USER ELEMENT, NODES=8, Type= U1, PROPERTIES=3, COORDINATES=3,
+#   UNSYMM, I PROPERTIES=2, VARIABLES=21
+#   1, 2, 3
+#  *UEL PROPERTY, elset=Cz
+#   1.000, 31.003, 6.7894,
+#   1, 2
+#  *ELEMENT, TYPE=U1, elset=Cz
...
-#  ** Section: Section-2
-#  *Cohesive Section, elset=Cz, material=Material-2, response=TRACTION SEPARATION
-#  ,
+#  **** Section: Section-2
+#  ***Cohesive Section, elset=Cz, material=Material-2, response=TRACTION SEPARATION
```

(continues on next page)

(continued from previous page)

```
+#  **,
...
```

where 3 in `PROPERTIES=3` is the length of *CzMat* list and 2 in `I PROPERTIES=2` is the length of *CzIntMat* list

---

Metadata

Environment

Python

Version

v1.0.0

Date

2020-12-20

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

---

czmtestkit.abaqus_modules.historyOutput

`czmtestkit.abaqus_modules.`**`historyOutput`**`(`*dict*`)`

Fetch history output from `.odb` file and save to `.csv` file.

Parameters

dict (*dict*):

'JobID' name of the `.odb` file.

Example

If reaction force and displacement at Node 2 of the assembly are requested as history output to `ExampleJob.odb`, then executing the following:

```
historyOutput({'JobID':'ExampleJob'})
```

results in the creation of `ExampleJob.csv` with the history output:

ExampleJob.csv:

Table 3.11: ExampleJob.csv

Node AS-SEMBLY.2	Node AS-SEMBLY.2	Node AS-SEMBLY.2	Node AS-SEMBLY.2	Node AS-SEMBLY.2	Node AS-SEMBLY.2
RF	RF	RF	U	U	U
1	2	3	1	2	3
-0.0	0.0	-0.0	0.0	0.0	0.0
0.0018873203080147	5	0.6899987459182704	0.0	0.0	2.0
-0.0001103667163988	0.0 58	1.3232073783874	5 0.0	0.0	4.0
-3.97297917515971e-05	0.0	1.895939111709599	0.0	0.0	6.0
9.44650037126848e-06	0.0	2.364769220352177	0.0	0.0	8.0
0.0002365063992328	94	2.503397226333627	0.0	0.0	10.0
5.893718480365351e-05	0.0	2.499580383300788	0.0	0.0	12.0
-0.0006786909070797	0.0 26	2.423223257064827	0.0	0.0	14.0
2.220768510596831e-05	0.0	2.313085794448855	0.0	0.0	16.0
-6.120345460658	0.0 43e-06	2.183591127395637	0.0	0.0	18.0
-0.0028251307085156	0.0 4	1.5845751762390	1 0.0	0.0	20.0

Metadata

Environment

Abaqus/CAE

Version

v1.0.0

Date

2020-12-20

Authors

Nanditha Mudunuru

Contribution: v1.0.0

Email: nanditha.mudunuru@gmail.com

Guidelines for contributing to abaqus_modules

To use your own abaqus-python functions with the `czmtestkit`, start by testing your function in the Abaqus/CAE Python Development Environment (PDE). Assume that `abqPy_Func1` is your function based on abaqus-python scripting language with *param_1*, *param_2* and *param_3* as input parameters.

```
def abqPy_Func1(param_1, param_2, param_3):
    # The function code to be tested goes below
```

Ensure the command is exectable without errors in the `PDE`, using the following command after assigning values of corresponding data types:

```
abqPy_Func1(param_1=value_param1, param_2=value_param2, param_3=value_param3)
```

Then, convert the function to make it compatible with `czmtestkit` with the following changes.

```
def abqPy_Func1(dict): # Change input parameter
    for k in dict.keys(): exec("{0} = dict[\'{0}\']".format(k)) # New line to be added␣
↪to the script
    # Rest of the tested function code goes below
```

Now you can pass a dictionary of variables to `abqPy_Func1` with keys *param_1*, *param_2* and *param_3* and values of corresponding data types. For example:

```
In_dict_Func1 = {
    'param_1': value_param1
    'param_2': value_param2
    'param_3': value_param3
}
abqPy_Func1(In_dict_Func1)
```

After testing the funciton in the `PDE`, create a `moduleName.py` file in the `abaqus_modules` subdirectory and copy the converted function code (`def abqPy_Func1(dict):...`) to this file. Further, import the funciton to the `czmtestkit.abaqus_modules` module by appending the following line to the `abaqus_modules\__init__.py` file.

```
from moduleName.py import abqPy_Func1
```

```
    $ Local czmtestkit repository
    ├── ...
    ├── czmtestkit
    |   ├── abaqus_modules
m#  |   |   ├── __init__.py
    |   |   |   └── ...
+#  |   |   |       from moduleName import abqPy_Func1
+#  |   |   ├── moduleName.py
```

(continues on next page)

(continued from previous page)

```
    |   |   └── ...
    |   ├── py_modules
    |   └── __init__.py
    ├── ...
    └── setup.py
```

Make sure the names `moduleName` and `abqPy_Func1` are unique and have not already been used. Further, if you have more than one function in `moduleName.py` file, use the following command to import all the funcitons at once.

```
from moduleName.py import *
```

Finally, reinstall the `czmtestkit` locally (see) and test your module with *czmtestkit.py_modules.abqFun()*.

```
# Create dictionary
In_dict_Func1 = {
    'param_1': value_param1
    'param_2': value_param2
    'param_3': value_param3
}

# Write data to a .json file
import json
filePath = 'dataIn.json'
with open(filePath, 'a') as file:
    json.dump(In_dict_Func1, file)
    file.write("\\n")

# Execute the abaqus-python script with czmtestkit
import os
czmtestkit.py_modules.abqFun(filePath, 'czmtestkit.abaqus_modules.abqPy_Func1',os.
↪getcwd())
```

Now that you verified your script with the `czmtestkit`, go ahead and send us a pull request to share your code with the world. While this step is optional, we highly encourage you to do this.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX