DDQT

Arun Manohar

Sep 19, 2021

CONTENTS

The domain of *Nondestructive Testing (NDT)* and *Structural Health Monitoring (SHM)* comprise of techniques that are used to evaluate the state of health of a structure without causing any damage to the structure being inspected. Typical examples of structures being inspected include aircraft components, bridges, nuclear reactors, etc. *Defect Detection and Quantification Toolbox (DDQT)* aims to provide a framework to automate the routine tasks for researchers in the field of *NDT* and *SHM*.

In the fields of *NDT* and *SHM*, experiments are performed using a variety of modalities like Ultrasound, Infrared Thermography, X-Rays, etc. *Matlab* is often used to perform experiments. Typically, the resulting data is very often a 3D dataset comprising of time axis and 2D spatial axis. Once the data is obtained, a variety of visualization checks are performed. Following this, features are created at each and every spatial location in the time series. Some examples of the types of features are based on time series, gradients, spatial filters, etc. At this stage, to avoid the *curse of dimensionality*, a feature reduction step is performed using *PCA* in case of *unsupervised defect detection*. Once the subset of meaningful features is identified, the defect regions are detected using a statistical distance metric like *Mahalanobis distance*. In *DDQT*, we also propose a realatively newer method to identify defects using *Isolation Forests*. Following this, the performance of the feature space and detection algorithms is quantified using *Receiver Operating Curves (ROC)* curves compared to the raw data. The performance is quantified using commonly used classification metrics such as *True Positive Rate (TPR)*, *False Positive Rate (FPR)* and *Area Under Curve (AUC)*. A visual representation of these metrics is presented using charts to aid the user.

While there are clearly defined steps that researchers in field of *NDT* and *SHM* often pursue, to the best of our knowledge there are no offerings that provide a framework so researchers can mainly focus on the feature space and tweaking the defect detection algorithms. With very minimal edits to the driver program, researchers can visualize the results with ease and focus on the underlying *physics* to improve the performance of defect detection instead of spending much time and effort on setting up the pipeline process.

In Summary, \`DDQT\` can be used for;

- Reading in Matlab data
- Visualizing data
- Creating features in the time and spatial domain
- Feature reduction using PCA
- Identifying defects using Mahalanobis distance and Isolation Forest
- Quantifying results using ROC curves
- Visualizing outcomes

There are numerous avenues to enhance this toolbox. I welcome any contributions to this program. **Some possible areas that could use improvements are;**

- Improvements in feature space
- Improvements to defect detection algorithms
- Coding enhancements
- Documentation enhancements
- Currently, only certain time stamps are used in calculating computationally intensive features. There is scope to write more computationally efficient code to handle more time stamps (if not everything…)
- Possibility of including circular defects - currently, defects are defined using polygon vertices

If you would like to collaborate with me in improving this toolbox or if you would like to provide sample data, please reach out to me at

```
>>>my_first_name = 'arun'
>>>print(str(my_first_name) + 'mano121@outlook.com')
```

Feel free to fork and add any enhancements, and let me know if a pull request is needed to merge the changes.

If you use this work in your research, please cite using;

```
@software{ArunManohar_20210322,
  author       = {Arun Manohar},
  title        = {{Defect Detection and Quantification Toolbox (DDQT)}},
  month        = mar,
  year         = 2021,
  publisher    = {Zenodo},
  version      = {v0.1.0},
  doi          = {10.5281/zenodo.4627984},
  url          = {https://doi.org/10.5281/zenodo.4627984}
}
```

Thank you!

CHAPTER

ONE

README

Defect Detection and Quantification Toolbox (DDQT)

Arun Manohar (2021)

`DDQT` can be used for;

- Reading in Matlab data
- Visualizing data
- Creating features in the time and spatial domain
- Feature reduction using PCA
- Identifying defects using Mahalanobis distance and Isolation Forest
- Quantifying results using ROC curves
- Visualizing outcomes

There are numerous avenues to enhance this toolbox. I welcome any contributions to this program. **Some possible areas that could use improvements are;**

- Improvements in feature space
- Improvements to defect detection algorithms
- Coding enhancements
- Documentation enhancements
- Currently, only certain time stamps are used in calculating computationally intensive features. There is scope to write more computationally efficient code to handle more time stamps (if not everything…)
- Possibility of including circular defects - currently, defects are defined using polygon vertices

If you would like to collaborate with me in improving this toolbox or if you would like to provide sample data, please reach out to me at

```
>>>my_first_name = 'arun'
>>>print(str(my_first_name) + 'mano121@outlook.com')
```

Feel free to fork and add any enhancements, and let me know if a pull request is needed to merge the changes.

If you use this work in your research, please cite using;

```
@software{ArunManohar_20210322,
  author       = {Arun Manohar},
  title        = {{Defect Detection and Quantification Toolbox (DDQT)}},
  month        = mar,
```

(continues on next page)

(continued from previous page)

```
  year          = 2021,
  publisher     = {Zenodo},
  version       = {v0.1.0},
  doi           = {10.5281/zenodo.4627984},
  url           = {https://doi.org/10.5281/zenodo.4627984}
}
```

Thank you!

CHAPTER

TWO

GETTING STARTED

2.1 Dependencies

In order to run the program, you need Python3 and the following dependencies.

- SciPy
- Matplotlib
- NumPy
- PyWavelets
- sklearn

2.2 Installing

Either use *git-clone* using the following command;

```
git clone https://github.com/arunmano121/DDQT.git MyDDQT
```

or manually download the two python files into your desired working directory. In the example *MyDDQT* is an example. You can use any name of your choice.

2.3 Running the program

cd into your working directory and run the program.

```
cd MyDDQT
./DefectDetection.py
```

2.4 Configuring Parameters

The program is designed so that all parameter settings need to be only edited within the *main()* module.

The following block is used to load Matlab data, this assume a dataset named *sample.mat* containing a table name *rawData1*. The output data is stored as *ndarray* and is named *mat*. This will be used for further processing.

```
print('Reading in raw matlab data using scipy IO modules...')
# Example - this assumes a matlab dataset named defect.mat and the
# table named rawData inside the dataset
dataset = 'sample.mat'
tablename = 'rawData1'
mat = read_matlab_data(dataset=dataset, table=tablename)
```

The data is assumed in the format containing time (*axis=0*) followed by spatial axis 1 (*axis=1*) and spatial axis 2 (*axis=2*) respectively. If the Matlab dataset contains data in a different axis order, re-arrange using numpy.moveaxis before proceeding to subsequent steps.

Data is described by the following block.

```
[t_max, s1_max, s2_max] = mat.shape
print('Shape of the data matrix')
print('t_max: %d  s1_max: %d s2_max: %d' % (t_max, s1_max, s2_max))
```

The range of the three different axis is set.

```
# scanning parameters
# in this sample code, time axes ranges from t_lb to t_ub over t_max
t_lb = 0*1e-1
t_ub = t_max*1e-1
t = np.linspace(t_lb, t_ub, t_max)

# s1 axis range from s1_lb to s1_ub divided over s1_max steps
s1_lb = 0
s1_ub = 200
s1 = np.linspace(s1_lb, s1_ub, s1_max)

# s2 axis range from s2_lb to s2_ub divided over s2_max steps
s2_lb = 0
s2_ub = 250
s2 = np.linspace(s2_lb, s2_ub, s2_max)
```

Units along the three different axis is held in a dictionary named *units*. In this example, the time axis is defined in *micro seconds*, while the two spatial axis are in *mm*. Set the appropriate units based on the experiment.

```
# dictionary object to hold the units along the different axis
units = {'t_units': '$\\mu$S', 's1_units': 'mm', 's2_units': 'mm'}
```

For ease of plotting, the *s1* and *s2* axis are converted to 2D meshgrid.

```
# meshgrid conversion in 2D
s1_2d, s2_2d = np.meshgrid(s1, s2, indexing='ij')
```

Raw data is visualized at four random spatial points by charting the time series.

```
# raw data visualization
print('Pick 4 random spatial coordinates and chart the time-series...')
visualize_time_series(mat, t, s1, s2, units)
```

The spatial data is visualized at different time stamps as needed. In the example below, the spatial data is visualized between time indices of 450 (*t_min_idx*) to 500 (*t_max_idx*) in steps of 25 (*del_t_idx*).

```
print('Visualize spatial slices of data at certain time stamps...')
t_min_idx = 450
t_max_idx = 500
del_t_idx = 25
visualize_spatial_data(mat, t, s1_2d, s2_2d,
                       t_min_idx, t_max_idx, del_t_idx, units)
```

The raw time series is very noisy and often a low-pass filter is desired. In this example, the time series is filtered using a simple *mean* filter. The filter avergages using the *size* parameter. The bigger the number, the more aggressive the filtering is.

```
# time series filtering of data
print('performing mean filtering at each spatial location...')
mat = mean_filter(mat, t, s1, s2, units, size=20, plot_sample=True)
```

The defects are defined using the *list* structure. As many defects can be setup. The defects can be defined using as many vertices as needed. Each defect is a *list* of *tuples*. The defect names or labels are a *list* containing *strings*.

```
# define defects
print('Defining coordinates of defects...')
# define as many defects as needed
# each defect should contain the coordinates of the vertices
# the structure is list of tuples
def1 = [(20, 20), (50, 10), (30, 40), (20, 30)]
def2 = [(120, 120), (180, 120), (150, 180)]
def3 = [(60, 60), (80, 60), (80, 80), (60, 80)]

# list contains all the defects
defs_coord = [def1, def2, def3]
def_names = ['D1', 'D2', 'D3']  # names of defects
defs = define_defects(s1, s2, defs_coord, def_names)
```

Calculation of features at every time index is computationally intensive. A sample of time stamps in defined. *t_stamps* defines the indices at which features are calculated, and where performance is finally measured.

```
# sample time indices where computationally intentionally features
# will be calculated.
t_stamps = range(500, 800, 100)
```

Feature engineering is very important and is based on problem at hand and creativity of the researcher. Feel free to define additional features as necessary. In the sample, the following family of features are calculated.

Identity features.

```
# identity features
features_id = {}
features_id['id'] = mat
```

Gradient based features.

```
# compute gradient features
print('Calculating spatial and temporal gradients...')
features_grad = {}
features_grad = compute_features_grad(mat)
```

Spatial domain features are calculated at desired time indices defined above.

```
# compute spatial domain features
print('Calculating spatial features at every location and time...')
features_sd = {}
features_sd = compute_features_sd(mat, t_stamps)
```

Time domain features are calculated at desired time indices defined above.

```
# compute time domain features
print('Calculating temporal features at every spatial location...')
features_td = {}
features_td = compute_features_td(mat, t_stamps)
```

Wavelet decomposition features are calculated at desired time indices defined above.

```
# compute wavelet decomposition features
print('Calculating wavelet transformed features at every location...')
features_wav = {}
features_wav = compute_features_wav(mat, t_stamps)
```

Once features are calculated, it is often desired to visualize the feature. The *visualize_features* accomplishes this as shown below. In the examples, *s1_grad* and *s2_grad* features belonging to *features_grad* are visualized.

```
# visualize feature
print('Visualizing computed features...')
t_idx = 650
visualize_features(mat, features_grad, s1_2d, s2_2d, 's1_grad',
                   t_idx, t, units)
visualize_features(mat, features_grad, s1_2d, s2_2d, 's2_grad',
                   t_idx, t, units)
```

The input features across all families are now combined into a single *feature* family for further processing. *combine_features* function combines the family of features as defined in the list named *feature_list*.

```
# combine features
print('Combining all features from different methods into a dict...')
feature_list = [features_id, features_grad, features_sd,
                features_td, features_wav]
features = {}
features = combine_features(feature_list)
print('Total number of features is %d' % (len(features)))
```

The features are scaled using the minimum and maximum values, so that the resulting features lie between 0-1. Scaling features has proven to be useful in Machine Learning.

```
# normalize features
print('Normalize features...')
features = normalize_features(features, t_stamps)
```

Outlier analysis is perfomed using two methods - Mahalanobis distance and Isolation Forest. If PCA is desired to reduce input dimensionality, set *pca_var* to the *Desired Variance* level. For example, if *pca_var* is set to 0.9, then it is implied that 90% variance is desired. Accordingly, PCA will choose the number of dimensions that are needed to achieve this. The result of Mahalanobis distance is output to the *ndarray* named *mah*.

```
# Outlier analysis using Mahalanobis distance
# if PCA is required to trim features, set pca_var to the desired
# explained varaince level - in this example, 90% variance is desired
print('Mahalanobis distance to identify outliers...')
mah = {}
mah = outlier_mah(features, t_stamps, pca_var=0.9)
```

Another popular method to detect outliers uses *Isolation Forest* method. The result is output to the *ndarray* named *iso*.

```
# fit Isolation Forest model
# if PCA is required to trim features, set pca_var to the desired
# explained variance level - in this example, 90% variance is desired
print('Fit Isolation Forest model...')
iso = {}
iso = fit_isolationforest_model(features, t_stamps, pca_var=0.9)
```

In order to better visualize the results contained in *mah* and *iso*, the frames are scaled between 0-1 using the minimum and maximum values of the arrays.

```
# scale frames between 0-1
print('Scaling frames between 0-1 for better interpretability...')
mat = scale_frames(mat, t_stamps)
mah = scale_frames(mah, t_stamps)
iso = scale_frames(iso, t_stamps)
```

defect_detection_metrics will compute the performance of the algorithms using *True Positive Rate (TPR)*, *False Positive Rate (FPR)* and *Area Under Curve (AUC)* metrics. The function will also output the *TPR* at *FPR* rates of 2%, 5% and 10%. If *plot* parameter is set to *True*, the *Reciever Operating Characteristic (ROC)* curves are plotted to show the improvement obtained over the raw data.

```
# Defect detection metrics
print('Quantification of defect detection and plotting the results...')
defect_detection_metrics(mat, mah, iso, s1_2d, s2_2d,
                         defs, t_stamps, t, units, plot=True)
```

CHAPTER THREE

HOW TO CITE

if you use this work in your research, please cite using;

```
@software{ArunManohar_20210322,
  author       = {Arun Manohar},
  title        = {{Defect Detection and Quantification Toolbox (DDQT)}},
  month        = mar,
  year         = 2021,
  publisher    = {Zenodo},
  version      = {v0.1.0},
  doi          = {10.5281/zenodo.4627984},
  url          = {https://doi.org/10.5281/zenodo.4627984}
}
```

Thank you!

CHAPTER

FOUR

CODE REFERENCE

4.1 DefectDetection module

`DefectDetection.`**`annotate_plots`**`(`*ax*, *defs*`)`

Annotate charts with locations of defects

Parameters

ax: axis object plot axis

defs: dict defect parameters

Returns

None

`DefectDetection.`**`combine_features`**`(`*feature_list*`)`

Combine all features from different methods into one single dict

Parameters

feature_list: list list containing all entries of input features that need to be concatenated

Returns

features: dict feature dictionary containing all the input features

`DefectDetection.`**`compute_features_grad`**`(`*mat*`)`

Calculates spatial and temporal gradients

Parameters

mat: ndarray raw data

Returns

features_grad: dict dictionary containing spatial and temporal gradient features

`DefectDetection.`**`compute_features_sd`**`(`*mat*, *t_stamps*`)`

Calculates spatial features at every location and time stamp

Parameters

mat: ndarray raw data

t_stamps: list time stamps at which time domain features are calculated

Returns

features_sd: dict dictionary containing spatial domain features

`DefectDetection.`**`compute_features_td`**`(`*mat*, *t_stamps*`)`

Calculating temporal features at every spatial location

Parameters

mat: ndarray raw data

t_stamps: list time stamps at which time domain features are calculated

Returns

features_td: dict dictionary containing time domain features

`DefectDetection.`**`compute_features_wav`**`(`*mat*, *t_stamps*`)`

Calculates wavelet transformed features at every location

Parameters

mat: ndarray raw data

t_stamps: list time stamps at which wavelet features are calculated

Returns

features_wav: dict dictionary containing wavelet features

`DefectDetection.`**`defect_detection_metrics`**`(`*mat*, *mah*, *iso*, *s1_2d*, *s2_2d*, *defs*, *t_stamps*, *t*, *units*, *plot=True*`)`

Quantification of defect detection, and plotting the results

True-Positive Rate (TPR), False-Positive Rate (FPR), Receiver Operating Curves (ROC) are calculated for the raw data, Mahalanobis distance and result of Isolation Forest method. In addition, Area Under Curve (AUC) is also calculated to quantify the performance. Often, performance in terms of higher TPR is desired at lower FPR. To aid this, TPR values are calculated at 2%, 5% and 10% FPR. Further, the results are presented graphically if needed.

Parameters

mat: ndarray raw data - 3D *float* array

mah: ndarray result of performing Mahalanobis distance - 3D *float* array

iso: ndarray result of performing Isolation Forest algorithm - 3D *float* array

s1_2d: ndarray 2D meshgrid representation of s1 axis

s2_2d: ndarray 2D meshgrid representation of s2 axis

defs: dict defect parameters

t_stamps: list time stamps at which features were calculated and where results are desired

t: list time coordinates

units: dict units of the different dimensions

plot: Bool Boolean to indicate if plots are needed to visualize

Returns

None

`DefectDetection.`**`define_defects`**`(`*s1*, *s2*, *defs_coord*, *def_names*`)`

Define coordinates of defects

Parameters

s1: list spatial axis 1

s2: list spatial axis 2

defs_coord: list list containing all defects - each defect contains a list of tuples containing the vertices of defect

def_names: dict dictionary containing the names of defects

Returns

defs: dict dictionary containing all the necessary parameters of all the defined defects

`DefectDetection.`**`fit_isolationforest_model`**`(`*features*, *t_stamps*, *pca_var*`)`
Fit Isolation Forest model

Parameters

features: dict dictionary containing all input features

t_stamps: list time stamps at which features were calculated and where results are desired

pca_var: float contains the desired explained variance parameter, if less than 1.0, PCA will be performed

Returns

iso: ndarray result of Isolation Forest model over the data

`DefectDetection.`**`main`**`()`
All the subroutines will be called from here

`DefectDetection.`**`mean_filter`**`(`*mat*, *t*, *s1*, *s2*, *units*, *size*, *plot_sample*`)`
Performs mean filtering at each location

Parameters

mat: ndarray raw data

t: list time axis

s1: list spatial axis 1

s2: list spatial axis 2

units: dict units of the different dimensions

size: int number of elements to use in the mean filter. The higher, the more aggresive the filtering

plot_sample: Bool Boolean to indicate if time series plots are needed to compare raw and filtered data

Returns

filt_mat: ndarray mean filtered raw data based on kernel size

`DefectDetection.`**`normalize_features`**`(`*features*, *t_stamps*`)`
Normalize features

Parameters

features: dict dictionary containing all input features

t_stamps: list time stamps at which features were calculated and where results are desired

Returns

features: dict dictionary containing all normalized features

`DefectDetection.`**`outlier_mah`**`(`*features*, *t_stamps*, *pca_var*`)`
Mahalanobis distance to identify outliers

Parameters

features: dict dictionary containing all input features

t_stamps: list time stamps at which features were calculated and where results are desired

pca_var: float contains the desired explained variance parameter, if less than 1.0, PCA will be performed

Returns

mah: ndarray contains the result of computing Mahalanobis distance over the data

`DefectDetection.`**`read_matlab_data`**`(`*dataset*, *table*`)`

Reads in raw matlab data using scipy IO modules

Parameters

dataset: ndarray name of the Matlab dataset

table: str name of table within Matlab

Returns

mat: ndarray matlab data that has been converted to numpy array

`DefectDetection.`**`scale_frames`**`(`*arr*, *t_stamps*`)`

Scale frames between 0-1 for better interpretability

Parameters

arr: ndarray input array that needs to be scaled

t_stamps: list time stamps at which features were calculated and where results are desired

Returns

outarr: ndarray scaled array where the elements lie between 0-1

`DefectDetection.`**`visualize_features`**`(`*mat*, *features*, *s1_2d*, *s2_2d*, *feature*, *t_idx*, *t*, *units*`)`

Visualize computed features

Parameters

mat: ndarray raw data

features: dict dictionary containing input features

s1_2d: ndarray 2D meshgrid representation of s1 axis

s2_2d: ndarray 2D meshgrid representation of s2 axis

feature: str desired feature that needs to be visualized

t_idx: int time index at which visualization is needed

units: dict units of the different dimensions

Returns

None

`DefectDetection.`**`visualize_spatial_data`**`(`*mat*, *t*, *s1_2d*, *s2_2d*, *t_min_idx*, *t_max_idx*, *del_t_idx*, *units*`)`

Visualize spatial slices of data at certain time stamps

Parameters

mat: ndarray raw data

t: list time axis

s1_2d: ndarray 2D meshgrid representation of s1 axis

s2_2d: ndarray 2D meshgrid representation of s2 axis

t_min_idx: int lower bound time index for visualization

t_max_idx: int upper bound time index for visualization

del_t_idx: int time index steps for visualization

units: dict units of the different dimensions

Returns

None

`DefectDetection.`**`visualize_time_series`**`(`*mat*, *t*, *s1*, *s2*, *units*`)`

Pick 4 random spatial coordinates and chart the time-series

Parameters

mat: ndarray raw data

t: list time axis

s1: list spatial axis 1

s2: list spatial axis 2

units: dict units of the different dimensions

Returns

None

4.2 point_in_convex_polygon module

Helper module to determine if a point lies within a polygon

Script is based on Ref1 and Ref2.

`class` `point_in_convex_polygon.`**`Point`**`(`*s1*, *s2*`)`

Bases: `object`

Point class to define a point

`point_in_convex_polygon.`**`is_within_polygon`**`(`*polygon*, *point*`)`

Determine if a point lies within the polygon

Parameters

polygon: list of points polygon definition using a set of points

point: class:'Point' a single point

Returns

True/False: Bool Depending on if point lies within polygon

CHAPTER

FIVE

LICENSE

CHAPTER

SIX

ACKNOWLEDGEMENTS

Thanks to Prof. Michael Todd, Dr. See Yenn Chong and Mr. Zihan Wu at University of California, San Diego, for valuable discussions and inputs.

CHAPTER

SEVEN

CONTACT

Arun Manohar

```
>>>my_first_name = 'arun'
>>>print(str(my_first_name) + 'mano121@outlook.com')
```

PYTHON MODULE INDEX

INDEX