# BrakeSqueal Documentation

***Release 0.1***

**Ali H. Kadar**

August 06, 2014

Contents:

CHAPTER

ONE

# BRAKE

## 1.1 __init__

# CHAPTER TWO

# INITIALIZE

## 2.1 logger

This module defines the following functions:

```
- return_info_logger:

  Creates and returns a python logger object for information logging

- return_time_logger:

  Creates and returns a python logger object for time logging
```

logger.**return_info_logger**(*obj*)

INPUT:

- `obj` – object of the class `BrakeClass`

OUTPUT:

- `logger_i` – python logger object for information logging

logger.**return_time_logger**(*obj*)

INPUT:

- `obj` – object of the class `BrakeClass`

OUTPUT:

- `logger_t` – python logger object for time logging

## 2.2 load

This module defines the following functions:

```
- load_matrices:

  This function loads the sparse data matrices in .mat format and adds them to a python list
```

load.**load_matrices**(*obj*)

INPUT:

- `obj` – object of the class BrakeClass

OUTPUT:

- `sparse_list` – a python list of matrices in Compressed Sparse Column format of type '<type 'numpy.float64'>'

The `sparse_list` is obtained by loading the vaious .mat files present in the `data_file_list` attribute of the `BrakeClass` and then appending them into a python list `sparse_list`

## 2.3 assemble

This module defines the following functions:

```
- create_MCK:

  Assembles the various component matrices together(for the given angular frequency
  ``omega``) to form the mass(M), damping(C) and stiffness matrix(K).
```

`assemble.`**`create_MCK`**(*obj*, *sparse_list*, *omega*)

INPUT:

- `obj` – object of the class `BrakeClass`
- `sparse_list` – a python list of matrices in Compressed Sparse Column format of type '<type 'numpy.float64'>'
- `omega` – angular frequency

OUTPUT:

- `M` – Mass Matrix
- `C` – Damping Matrix
- `K` – Stiffness Matrix

The `M` , `C` , `K` are assembled as follows:

- `M` = m
- `C` = c1+c2*(omega/omegaRef)+c3*(omegaRef/omega)
- `K` = k1+k2+k3*math.pow((omega/omegaRef),2)

## 2.4 shift

This module defines the following functions:

```
- shift_matrices:

  Transform the qevp using shift and invert spectral transformations.
```

`shift.`**`shift_matrices`**(*obj*, *m*, *c*, *k*, *tau*)

INPUT:

- `obj` – object of the class `BrakeClass`
- `m` – Mass Matrix
- `c` – Damping Matrix
- `k` – Stiffness Matrix
- `tau` – shift

OUTPUT:

- `M` – Shifted Mass Matrix
- `C` – Shifted Damping Matrix
- `K` – Shifted Stiffness Matrix

The `M` , `C` , `K` are obtained as follows:

- `M` = m
- `C` = 2 * tau * m + c
- `K` = tau_squared * m + tau * c + k

## 2.5 scale

This module defines the following functions:

```
- scale_matrices:

  Scales the M, C, K matrices using 2-scalers before linearization.
```

`scale.`**`scale_matrices`**`(`*obj*, *m*, *c*, *k*`)`

INPUT:

- `obj` – object of the class `BrakeClass`
- `m` – Mass Matrix
- `c` – Damping Matrix
- `k` – Stiffness Matrix

OUTPUT:

- `M` – Scaled Mass Matrix
- `C` – Scaled Damping Matrix
- `K` – Scaled Stiffness Matrix
- `gamma` – scaling parameter
- `delta` – scaling parameter

The `M` , `C` , `K`, `gamma`, `delta` are obtained as follows:

- `M` = gamma*gamma*delta*m;
- `C` = gamma*delta*c;
- `K` = delta*k;
- `gamma` = math.sqrt(k_norm/m_norm);
- `delta` = 2/(k_norm+c_norm*gamma);

## 2.6 diagscale

This module defines the following functions:

```
- normalize_cols:

  Returns a diagonal matrix D such that every column of A*D has eucledian norm = 1.

- norm_rc:

  Returns diagonal matrix DL and DR such that every row and every column of DL*Y*DR
  has euclidean norm ~ 1.

- diag_scale_matrices:

  Diagonally scales the shifted scalar-scaled matrices using DL, DR to improve the
  condition number.
```

diagscale.**diag_scale_matrices**(*obj*, *M*, *C*, *K*)

INPUT:

- `obj` – object of the class `BrakeClass`
- `M` – Mass Matrix
- `C` – Damping Matrix
- `K` – Stiffness Matrix

OUTPUT:

- `M` – Diagonally Scaled Mass Matrix
- `C` – Diagonally Scaled Damping Matrix
- `K` – Diagonally Scaled Stiffness Matrix
- `DR` – Matrix that normalize the columns

The `M` , `C` , `K`, `DR` are obtained as follows:

- `M` = `DL * M * DR`
- `C` = `DL * C * DR`
- `K` = `DL * K * DR`

diagscale.**norm_rc**(*Y*)

INPUT:

- `Y` – matrix that needs to be normalized across both columns and rows

OUTPUT:

- `DL` – `DL` = ..... `Drow3` * `Drow2` * `Drow1` * I
- `DR` – `DR` = `I` * `Dcol1` * `Dcol2` * `Dcol3`......

The `DL` and `DR` are obtained as a converging sequence :

- set `Dcol = normalize columns(Y)`
- set `DR = DR * Dcol`
- set `Y = Y * Dcol`
- set `Drow = normalize rows(Y) = normalize columns(Y.T)`
- set `DL = Drow * DL`
- set `Y = Drow * Y`

- when `Dcol` and `Drow` are sufficiently close to `I` or max no of iterations reached STOP and return, else continue with step 1.

`diagscale.`**`normalize_cols`**`(`*A*`)`

INPUT:

- `A` – matrix that needs to be normalized(columnwise)

OUTPUT:

- `D` – diagonal matrix such that every column of `A*D` has eucledian norm = 1.

`D` is obtained as follows:

- square the elements of A and sum each column
- set the diagonal elemnts of D as the inverse square root of the column sums

## 2.7 unlinearize

This module defines the following functions:

```
- unlinearize_matrices:

  To obtain the eigenvectors prior linearization of the QEVP from the
  resulting eigenvectors (after classical companion linearization of the QEVP
  to obtain the generalized eigenvalue propblem).
```

`unlinearize.`**`unlinearize_matrices`**`(`*evec*`)`

INPUT:

- `evec` – eigenvectors of the generalized eigenvalue propblem

OUTPUT:

- `evec_prior` – eigenvectors prior linearization of the QEVP

The `evec_prior` are obtained as follows:

- check for every vector `i` of the GEVP(`evec`) (consider MATLAB convention)
- if `norm(evec[1:n,i]) > norm(evec[n+1:2*n,i]) set evec_prior[:,i] = evec[1:n,i]`
- else `set evec_prior[:,i] = evec[n+1:2*n,i]`

CHAPTER

THREE

# SOLVE

## 3.1 projection

This module defines the following functions:

```
- obtain_projection_matrix:

  This function forms the Projection Matrix by solving the quadratic eigenvalue problem
  for each base angular frequency.
```

`projection.`**`obtain_projection_matrix`**`(`*obj*`)`

INPUT:

- `obj` – object of the class `BrakeClass`

OUTPUT:

- `Q` – projection matrix

The projection matrix is obtained as follows:

- Obtain the measurment matrix `X = [X_real X_imag]`, with `X_real` as a list of real parts of eigenvectors and `X_imag` as a list of imaginary parts of eigenvectors, corresponding to each base angular frequency in `omega_basis`.
- Compute the thin svd of the measurment matrix. `X = U * s * V`
- Set `Q = truncated(U)`, where the truncation is done to take only the significant singular values(based on a certain tolerance) into account

## 3.2 qevp

This module defines the following functions:

```
- brake_squeal_qevp:

  For a particuar base angular frequency this function assembles the eigenvalues
  and eigenvectors for different shift points in the target region.

-  Obtain_eigs:

  For a particuar base angular frequency and for a particular shift point this
  function evaluates the eigenvalues and eigenvectors
```

qevp.**Obtain_eigs**(*obj*, *freq_i*, *qevp_j*, *omega*, *next_shift*)

INPUT:

- `obj` – object of the class `BrakeClass`
- `freq_i` – the index of the base angular freq
- `qevp_j` – the index of the shift point
- `omega` – ith base angular freq
- `next_shift` – jth shift point in the target region

OUTPUT:

- `la` – eigenvalues
- `evec` – eigenvectors

The `la` and `evec` are obtained as follows:

- load the various component matrices
- assemble the various component matrices together(for the given angular frequency `omega`) to form the mass(M), damping(C) and stiffness matrix(K).
- because we are interested in inner eigenvalues around certain shift points `next_shift`, so we transform the qevp using shift and invert spectral transformations.
- 

qevp.**brake_squeal_qevp**(*obj*, *freq_i*, *omega*)

INPUT:

- `obj` – object of the class `BrakeClass`
- `freq_i` – the index of the base angular freq in
- `omega` – ith base angular freq

OUTPUT:

- `assembled_la` – assembled eigenvalues
- `assembled_evec` – assembled eigenvectors

The `assembled_la` and `assembled_evec` are obtained as follows:

- calculate the next shift point in the target region
- obtain eigenvalues and eigenvectors for that particular shift point
- add the eigenvalues and eigenvectors to `assembled_la` and `assembled_evec` respectively
- check if the required area fraction of the target region has been covered. If `yes` return `assembled_la` and `assembled_evec` else calculate the next shift point in the target region and repeat

## 3.3 solver

Function for the generalized eigenvalue problem

Input A x = lamda B x evs_per_shift: no of eigenvalues required kind: largest or smallest in magnitude. parameters 'LM', 'SM' respectively flag: flag should be passed true when B is positive definite

**Output**

1. la (Array of evs_per_shift eigenvalues.)
2. **v (An array of evs_per_shift eigenvectors. v[:, i] is the eigenvector** corresponding to the eigenvalue la[i])

Additional Info Example of eigs eigs(A, k=6, M=None, sigma=None, which='LM', v0=None, ncv=None, maxiter=None, tol=0, return_eigenvectors=True, Minv=None, OPinv=None, OPpart=None)

The eigs function of PYTHON can calculate the eigenvalues of the generalized eigenvalue problem A*x=lamda*M*x with the following conditions. M must represent a real, symmetric matrix if A is real, and must represent a complex, hermitian matrix if A is complex. If sigma is None, M has to be positive definite If sigma is specified, M has to be positive semi-definite

When sigma is specified 'say 0' then eigs function will calculate the eigenvalues nearest to sigma. The 'LM' clause along with sigma = 0 can be used to calculate the reciprocal eigenvalues of Largest Magnitude.

## 3.4 cover

Implementation of the MonteCarlo Algorithm for choosing shift points to cover the target region

cover.next_shift Input 1. target (target region for the shift points) 2. previous_shifts (python list for the previous shift points already calculated) 3. previous_radius (corresponding radius of the previous shift points)

Output 1. next_shift (next shift point in the target region)

`cover.`**`next_shift`**(*obj*, *previous_shifts*=[ ], *previous_radius*=[ ])
    Obtain the next shift in the target region

CHAPTER

FOUR

# ANALYZE

## 4.1 residual

Function definitions for obtaining the residual

# CHAPTER FIVE

# INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

# PYTHON MODULE INDEX

## Symbols



## A



## B



## C



## D



## L



## N



## O



## P



## Q



## R



## S



## U