SpecSWD

Generated by Doxygen 1.13.2

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 specswd::Mesh Struct Reference

SEM mesh class.

`#include <mesh.hpp>`

Public Member Functions

- void read_model (const char *filename)

  read 1D model
- void create_database (float freq, float phi)

  Create SEM database by using input model info.
- void **print_model** () const

  print 1-D model information
- void **print_database** () const
- void **allocate_1D_model** (int nz0, int swd_type, int has_att)
- void **create_model_attributes** ()

  create attributes for elastic/acoustic regions
- void interp_model (const float *param, const std::vector< int > &elmnts, std::vector< float > &md) const

  interpolate elastic/acoustic model by using coordinates
- void project_kl (const float *frekl, float *kl_out) const

  project kernels to original 1-D model
- void **create_material_info_** ()
- void read_model_header_ (const char *filename)

  read header of 1D model, including wave type, attenutation flag, attenuation model flag
- void read_model_love_ (const char *filename)

  read 1D VTI model for Love wave
- void read_model_rayl_ (const char *filename)

  read 1D VTI model for Rayleigh wave
- void read_model_full_aniso_ (const char *filename)

  read 1D full anisotropy model model for Rayleigh wave
- void **compute_minmax_veloc_** (float phi, std::vector< float > &vmin, std::vector< float > &vmax)
- void **create_db_love_** ()

  create database for Love wave
- void **create_db_rayl_** ()

  create database for Love wave
- void **create_db_aniso_** ()

  create database for Love wave

Public Attributes

- int **nspec**
- int **nspec_grl**
- int **nglob**
- std::vector< int > **ibool**
- std::vector< float > **skel**
- std::vector< float > **znodes**
- std::vector< float > **jaco**
- std::vector< float > **zstore**
- int **nspec_ac**
- int **nspec_el**
- int **nspec_ac_grl**
- int **nspec_el_grl**
- std::vector< char > **is_elastic**
- std::vector< char > **is_acoustic**
- std::vector< int > **el_elmnts**
- std::vector< int > **ac_elmnts**
- int **nglob_ac**
- int **nglob_el**
- std::vector< int > **ibool_el**
- std::vector< int > **ibool_ac**
- std::vector< float > **xrho_ac**
- std::vector< float > **xrho_el**
- bool **HAS_ATT**
- int **SWD_TYPE**
- std::vector< float > **xA**
- std::vector< float > **xC**
- std::vector< float > **xL**
- std::vector< float > **xeta**
- std::vector< float > **xN**
- std::vector< float > **xQA**
- std::vector< float > **xQC**
- std::vector< float > **xQL**
- std::vector< float > **xQN**
- int **nQmodel_ani**
- std::vector< float > **xC21**
- std::vector< float > **xQani**
- std::vector< float > **xkappa_ac**
- std::vector< float > **xQk_ac**
- int **nfaces_bdry**
- std::vector< int > **ispec_bdry**
- std::vector< char > **bdry_norm_direc**
- int **nz_tomo**
- int **nregions**
- std::vector< float > **rho_tomo**
- std::vector< float > **vpv_tomo**
- std::vector< float > **vph_tomo**
- std::vector< float > **vsv_tomo**
- std::vector< float > **vsh_tomo**
- std::vector< float > **eta_tomo**
- std::vector< float > **QC_tomo**
- std::vector< float > **QA_tomo**
- std::vector< float > **QL_tomo**
- std::vector< float > **QN_tomo**

- std::vector< float > **c21_tomo**
- std::vector< float > **Qani_tomo**
- std::vector< float > **depth_tomo**
- std::vector< int > **region_bdry**
- std::vector< int > **iregion_flag**
- std::vector< char > **is_el_reg**
- std::vector< char > **is_ac_reg**
- float **PHASE_VELOC_MIN**
- float **PHASE_VELOC_MAX**
- float **freq**

3.1.1 Detailed Description

SEM mesh class.

3.1.2 Member Function Documentation

3.1.2.1 create_database()

```
void specswd::Mesh::create_database (
            float freq0,
            float phi)
```

Create SEM database by using input model info.

Parameters

freq0	current frequency, in Hz
phi	directional angle

3.1.2.2 interp_model()

```
void specswd::Mesh::interp_model (
            const float * param,
            const std::vector< int > & elmnts,
            std::vector< float > & md) const
```

interpolate elastic/acoustic model by using coordinates

Parameters

param	input model parameter, shape(nz_tomo)
elmnts	all elements used, ispec = elmnts[i]
md	model required to interpolate, shape(nspec_el∗NGLL + nspec_el_grl ∗ NGRL)

3.1.2.3 project_kl()

```
void specswd::Mesh::project_kl (
            const float * frekl,
            float * kl_out) const
```

project kernels to original 1-D model

Parameters

frekl	derivatives, shape(nspec∗NGLL+NGRL)
kl_out	derivatives on original 1-Dmodel, shape(nz_)

3.1.2.4 read_model()

```
void specswd::Mesh::read_model (
            const char * filename)
```

read 1D model

Parameters

filename	1D model file

3.1.2.5 read_model_full_aniso_()

```
void specswd::Mesh::read_model_full_aniso_ (
            const char * filename)
```

read 1D full anisotropy model model for Rayleigh wave

Parameters

filename	1D model file

3.1.2.6 read_model_header_()

```
void specswd::Mesh::read_model_header_ (
            const char * filename)
```

read header of 1D model, including wave type, attenutation flag, attenuation model flag

Parameters

filename	model filename

3.1.2.7 read_model_love_()

```
void specswd::Mesh::read_model_love_ (
            const char * filename)
```

read 1D VTI model for Love wave

Parameters

filename	1D model file

3.1.2.8 read_model_rayl_()

```
void specswd::Mesh::read_model_rayl_ (
            const char * filename)
```

read 1D VTI model for Rayleigh wave

Parameters

filename	1D model file

The documentation for this struct was generated from the following files:

- mesh.hpp
- database.cpp
- initialize.cpp
- interpolate.cpp
- io.cpp

3.2 specswd::SolverLove Class Reference

Public Member Functions

- void **prepare_matrices** (const Mesh &M)

  prepare M/K/E matrices for Love wave, an/elastic case
- void compute_egn (const Mesh &M, std::vector< float > &c, std::vector< float > &egn, bool use_qz=false)

  compute Love wave dispersion and eigenfunctions, elastic case
- void compute_egn_att (const Mesh &M, std::vector< scmplx > &c, std::vector< scmplx > &egn, bool use←_qz=false)

  compute rayleigh wave dispersion and eigenfunctions, visco-elastic case
- float **group_vel** (const Mesh &M, float c, const float ∗egn) const

  compute velocity of love wave, elastic case
- scmplx **group_vel_att** (const Mesh &M, scmplx c, const scmplx ∗egn) const

  compute velocity of love wave, anelastic case
- void compute_phase_kl (const Mesh &M, float c, const float ∗egn, std::vector< float > &frekl) const

  compute love wave phase velocity kernels, elastic case
- void compute_phase_kl_att (const Mesh &M, scmplx c, const scmplx ∗egn, std::vector< float > &frekl_c, std::vector< float > &frekl_q) const

  compute love wave phase velocity kernels, visco-elastic case
- void compute_group_kl (const Mesh &M, float c, const float ∗egn, std::vector< float > &frekl) const

  compute group velocity and kernels for love wave group velocity, elastic case
- void compute_group_kl_att (const Mesh &M, scmplx c, scmplx u, const scmplx ∗egn, std::vector< float > &frekl_u, std::vector< float > &frekl_q) const

  compute love wave group velocity kernels, visco-elastic case
- void egn2displ (const Mesh &M, float c, const float ∗egn, float ∗__restrict displ) const

  convert eigenvector to displacement,elastic case
- void egn2displ_att (const Mesh &M, scmplx c, const scmplx ∗egn, scmplx ∗__restrict displ) const

  convert eigenvector to displacement, visco-elastic case
- void transform_kernels (const Mesh &M, std::vector< float > &frekl) const

  transform modulus kernel to velocity kernel, Love wave case

3.2.1 Member Function Documentation

3.2.1.1 compute_egn()

```
void specswd::SolverLove::compute_egn (
            const Mesh & mesh,
            std::vector< float > & c,
            std::vector< float > & egn,
            bool use_qz = false)
```

compute Love wave dispersion and eigenfunctions, elastic case

Parameters

mesh	Mesh class
c	dispersion, shape(nc)
egn	eigen functions(displ at y direction), shape(nc,nglob_el)
use_qz	if false, only compute phase velocities

3.2.1.2 compute_egn_att()

```
void specswd::SolverLove::compute_egn_att (
            const Mesh & mesh,
            std::vector< scmplx > & c,
            std::vector< scmplx > & displ,
            bool use_qz = false)
```

compute rayleigh wave dispersion and eigenfunctions, visco-elastic case

Parameters

mesh	Mesh class
c	dispersion, shape(nc) c = c0(1 + iQL^{-1})
displ	eigen functions(displ at y direction), shape(nc,nglob_el)
use_qz	if true, save QZ matrix

3.2.1.3 compute_group_kl()

```
void specswd::SolverLove::compute_group_kl (
            const Mesh & mesh,
            float c,
            const float * egn,
            std::vector< float > & frekl) const
```

compute group velocity and kernels for love wave group velocity, elastic case

Parameters

mesh	Mesh class
c	current phase velocity
egn	eigen function, shape(nglob_el)
frekl	Frechet kernels (N/L/rho) for elastic parameters, shape(3,nspec∗NGLL + NGRL)

3.2.1.4 compute_group_kl_att()

```
void specswd::SolverLove::compute_group_kl_att (
            const Mesh & mesh,
            scmplx c,
            scmplx u,
            const scmplx * egn,
            std::vector< float > & frekl_u,
            std::vector< float > & frekl_q) const
```

compute love wave group velocity kernels, visco-elastic case

Parameters

mesh	Mesh class
c	current complex phase velocity
u	current complex group velocity
egn	eigen function, shape(nglob_el)
frekl←↩ _u	dRe(u)/d(N/L/QN/QL/rho) shape(5,nspec∗NGLL + NGRL)
frekl←↩ _q	d(qi)/d(N/L/QN/QL/rho) shape(5,nspec∗NGLL + NGRL)

3.2.1.5 compute_phase_kl()

```
void specswd::SolverLove::compute_phase_kl (
            const Mesh & M,
            float c,
            const float * egn,
            std::vector< float > & frekl) const
```

compute love wave phase velocity kernels, elastic case

Parameters

M	Mesh clas
c	current phase velocity
egn	eigen function, shape(nglob_el)
frekl	Frechet kernels (N/L/rho) for elastic parameters, shape(3,nspec∗NGLL + NGRL)

3.2.1.6 compute_phase_kl_att()

```
void specswd::SolverLove::compute_phase_kl_att (
            const Mesh & M,
            scmplx c,
            const scmplx * egn,
            std::vector< float > & frekl_c,
            std::vector< float > & frekl_q) const
```

compute love wave phase velocity kernels, visco-elastic case

Parameters

M	mesh class
c	current complex phase velocity
egn	eigen function, shape(nglob_el)
frekl←↩ _c	dRe(c)/d(N/L/QN/QL/rho) shape(5,nspec∗NGLL + NGRL)
frekl←↩ _q	d(qi)/d(N/L/QN/QL/rho) shape(5,nspec∗NGLL + NGRL)

3.2.1.7 egn2displ()

```
void specswd::SolverLove::egn2displ (
            const Mesh & M,
            float c,
            const float * egn,
            float *__restrict displ) const
```

convert eigenvector to displacement,elastic case

Parameters

M	mesh class
c	current phase velocity
egn	eigenvector
displ	output displacement, shape(nspec∗NGLL+NGRL)

3.2.1.8 egn2displ_att()

```
void specswd::SolverLove::egn2displ_att (
            const Mesh & M,
            scmplx c,
            const scmplx * egn,
            scmplx *__restrict displ) const
```

convert eigenvector to displacement, visco-elastic case

Parameters

M	mesh class
c	current phase velocity
egn	eigenvector
displ	output displacement, shape(nspec∗NGLL+NGRL)

3.2.1.9 transform_kernels()

```
void specswd::SolverLove::transform_kernels (
            const Mesh & M,
            std::vector< float > & frekl) const
```

transform modulus kernel to velocity kernel, Love wave case

Parameters

M	mesh class
frekl	frechet kernels, the shape depends on: • `1`: elastic love wave: N/L/rho -> vsh/vsv/rho • `2`: anelastic love wave: N/L/QNi/QLi/rho -> vsh/vsv/QNi/QLi/rho

The documentation for this class was generated from the following files:

- vti.hpp
- eigenvalues.cpp
- frechet.cpp
- frechet_group.cpp
- group_velocity.cpp
- sem.cpp
- transform.cpp

3.3 specswd::SolverRayl Class Reference

Public Member Functions

- void prepare_matrices (const Mesh &M)

    preparing M/K/E matrices for Rayleigh wave
- void compute_egn (const Mesh &M, std::vector< float > &c, std::vector< float > &ur, std::vector< float > &ul, bool use_qz=false)

    compute rayleigh wave dispersion and eigenfunctions, elastic case
- void compute_egn_att (const Mesh &M, std::vector< scmplx > &c, std::vector< scmplx > &ur, std::vector< scmplx > &ul, bool use_qz=false)

    compute rayleigh wave dispersion and eigenfunctions, visco-elastic case
- float **group_vel** (const Mesh &M, float c, const float ∗ur, const float ∗ul) const

    compute velocity of love wave, elastic case
- scmplx **group_vel_att** (const Mesh &M, scmplx c, const scmplx ∗ur, const scmplx ∗ul) const

    compute velocity of love wave, visco-elastic case
- void compute_phase_kl (const Mesh &M, float c, const float ∗ur, const float ∗ul, std::vector< float > &frekl) const

    compute Rayleigh wave phase kernels, elastic case
- void compute_phase_kl_att (const Mesh &M, scmplx c, const scmplx ∗ur, const scmplx ∗ul, std::vector< float > &frekl_c, std::vector< float > &frekl_q) const

    compute Rayleigh wave phase kernels, visco-elastic case
- void compute_group_kl (const Mesh &M, float c, const float ∗ur, const float ∗ul, std::vector< float > &frekl) const

    compute group velocity and kernels for rayleigh wave group velocity, elastic case
- void compute_group_kl_att (const Mesh &M, scmplx c, scmplx u, const scmplx ∗ur, const scmplx ∗ul, std::vector< float > &frekl_u, std::vector< float > &frekl_q) const

    compute love wave group velocity kernels, visco-elastic case
- void egn2displ (const Mesh &M, float c, const float ∗egn, float ∗__restrict displ) const

    convert right eigenfunction to displacement, elastic case
- void egn2displ_att (const Mesh &M, scmplx c, const scmplx ∗egn, scmplx ∗__restrict displ) const

    convert right eigenfunction to displacement, visco-elastic case
- void transform_kernels (const Mesh &M, std::vector< float > &frekl) const

    transform modulus kernel to velocity kernel, Rayleigh wave case

3.3.1 Member Function Documentation

3.3.1.1 compute_egn()

```
void specswd::SolverRayl::compute_egn (
            const Mesh & mesh,
            std::vector< float > & c,
            std::vector< float > & ur,
            std::vector< float > & ul,
            bool use_qz = false)
```

compute rayleigh wave dispersion and eigenfunctions, elastic case

Parameters

mesh	Mesh class
c	dispersion, shape(nc) c = c0(1 + iQL^{-1})
ur/ul	left/right eigenvectors, shape(nc,nglob_el*2+nglob_ac)
use_qz	if true, save QZ matrix

3.3.1.2 compute_egn_att()

```
void specswd::SolverRayl::compute_egn_att (
            const Mesh & mesh,
            std::vector< scmplx > & c,
            std::vector< scmplx > & ur,
            std::vector< scmplx > & ul,
            bool use_qz = false)
```

compute rayleigh wave dispersion and eigenfunctions, visco-elastic case

Parameters

mesh	Mesh class
c	dispersion, shape(nc) c = c0(1 + iQL^{-1})
ur/ul	left/right eigenvectors, shape(nc,nglob_el*2+nglob_ac)
use_qz	if true, save QZ matrix

3.3.1.3 compute_group_kl()

```
void specswd::SolverRayl::compute_group_kl (
            const Mesh & mesh,
            float c,
            const float * ur,
            const float * ul,
            std::vector< float > & frekl) const
```

compute group velocity and kernels for rayleigh wave group velocity, elastic case

Parameters

mesh	Mesh class
c	current phase velocity
ur/ul	right/left eigen function, shape(nglob_el∗2+nglob_ac)
frekl	Frechet kernels (A/C/L/kappa/rho) for elastic parameters, shape(5,nspec∗NGLL + NGRL)

3.3.1.4 compute_group_kl_att()

```
void specswd::SolverRayl::compute_group_kl_att (
            const Mesh & mesh,
            scmplx c,
            scmplx u,
            const scmplx * ur,
            const scmplx * ul,
            std::vector< float > & frekl_u,
            std::vector< float > & frekl_q) const
```

compute love wave group velocity kernels, visco-elastic case

Parameters

mesh	Mesh class
c	current complex phase velocity
u	current complex group velocity
ur/ul	right/left eigen function, shape(nglob_el∗2+nglob_ac)
frekl_u	dRe(u)/d(N/L/QN/QL/rho) shape(5,nspec∗NGLL + NGRL)
frekl_q	d(qi)/d(N/L/QN/QL/rho) shape(5,nspec∗NGLL + NGRL)

3.3.1.5 compute_phase_kl()

```
void specswd::SolverRayl::compute_phase_kl (
            const Mesh & M,
            float c,
            const float * ur,
            const float * ul,
            std::vector< float > & frekl) const
```

compute Rayleigh wave phase kernels, elastic case

Parameters

M	mesh class
c	current phase velocity
ur/ul	right/left eigen function, shape(nglob_el∗2+nglob_ac)
frekl	Frechet kernels A/C/L/eta/kappa/rho_kl kernels for elastic parameters, shape(6,nspec∗NGLL + NGRL)

3.3.1.6 compute_phase_kl_att()

```
void specswd::SolverRayl::compute_phase_kl_att (
            const Mesh & M,
            scmplx c,
            const scmplx * ur,
            const scmplx * ul,
            std::vector< float > & frekl_c,
            std::vector< float > & frekl_q) const
```

compute Rayleigh wave phase kernels, visco-elastic case

Parameters

M	mesh class
c	current phase velocity
ur/ul	right/left eigen function, shape(nglob_el*2+nglob_ac)
frekl←_c	dRe(c)/d(A/C/L/eta/Qa/Qc/Ql/kappa/Qk/rho) kernels for elastic parameters, shape(10,nspec*NGLL + NGRL)
frekl←_q	dRe(Q_R)/d(A/C/L/eta/Qa/Qc/Ql/kappa/Qk/rho) kernels for elastic parameters, shape(10,nspec*NGLL + NGRL)

3.3.1.7 egn2displ()

```
void specswd::SolverRayl::egn2displ (
            const Mesh & M,
            float c,
            const float * egn,
            float *__restrict displ) const
```

convert right eigenfunction to displacement, elastic case

Parameters

M	Mesh class
c	current phase velocity
egn	eigenfunction,shape(nglob_el*2+nglob_ac)
displ	displacement, shape(2,npts)

3.3.1.8 egn2displ_att()

```
void specswd::SolverRayl::egn2displ_att (
            const Mesh & M,
            scmplx c,
            const scmplx * egn,
            scmplx *__restrict displ) const
```

convert right eigenfunction to displacement, visco-elastic case

Parameters

M	Mesh class
c	current phase velocity
egn	eigenfunction,shape(nglob_el*2+nglob_ac)
displ	displacement, shape(2,npts)

3.3.1.9 prepare_matrices()

```
void specswd::SolverRayl::prepare_matrices (
            const Mesh & M)
```

preparing M/K/E matrices for Rayleigh wave

Parameters

M	Mesh class

3.3.1.10 transform_kernels()

```
void specswd::SolverRayl::transform_kernels (
            const Mesh & M,
            std::vector< float > & frekl) const
```

transform modulus kernel to velocity kernel, Rayleigh wave case

Parameters

M	mesh class
frekl	frechet kernels, the shape depends on: • `1`: elastic rayleigh wave: A/C/L/eta/kappa/rho -> vph/vpv/vsv/eta/vp/rho • `2` anelastic rayleigh wave: A/C/L/eta/QAi/QCi/QLi/kappa/Qki/rho -> vph/vpv/vsv/eta/QAi/QCi/QLi/vp/Qki/rho

The documentation for this class was generated from the following files:

- vti.hpp
- eigenvalues.cpp
- frechet.cpp
- frechet_group.cpp
- group_velocity.cpp
- sem.cpp
- transform.cpp

Chapter 4

File Documentation

4.1 global.hpp

```
#ifndef SPECSWD_LIB_GLOB_H_
#define SPECSWD_LIB_GLOB_H_

#include "mesh/mesh.hpp"
#include "vti/vti.hpp"

#include <memory>
#include <complex>

#define CNAME(a) extern "C" void a

// global vars for solver/mesh

namespace specswd_pylib
{

extern std::unique_ptr<specswd::Mesh> M_;
extern std::unique_ptr<specswd::SolverLove> LoveSol_;
extern std::unique_ptr<specswd::SolverRayl> RaylSol_;

// global vars for eigenvalues/eigenvectors
extern std::vector<float> egnr_,egnl_,c_,u_;
extern std::vector<specswd::scmplx> cegnr_,cegnl_,cc_,cu_;

} // namespace specswd_pylib

#endif
```

4.2 specswd.hpp

```
#ifndef SPECSWD_LIB_UTILS_H_
#define SPECSWD_LIB_UTILS_H_

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

void
specswd_init_GQTable();

void
specswd_init_mesh_love(
    int nz, const float *z,const float *rho,const float *vsh,
    const float *vsv,const float *QN, const float *QL,
    bool HAS_ATT,bool print_tomo_info
);

void
specswd_init_mesh_rayl(
    int nz, const float *z,const float *rho,
    const float *vph,const float* vpv,const float *vsv,
    const float *QA, const float *QC,const float *QL,
    bool HAS_ATT,bool print_tomo_info
```

```
);

void
specswd_egn_love(float freq,bool use_qz);

void
specswd_egn_rayl(float freq,bool use_qz);

void
specswd_group_love();

void
specswd_group_rayl();


#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif
```

4.3 mesh.hpp

```
#ifndef SPECSWD_MESH_H_
#define SPECSWD_MESH_H_

#include <complex>
#include <vector>
#include <array>

namespace specswd
{

struct Mesh {

    // SEM Mesh
    int nspec,nspec_grl; // no. of elements for gll/grl layer
    int nglob; // no. of unique points
    std::vector<int> ibool; // connectivity matrix, shape(nspec * NGLL + NGRL)
    std::vector<float> skel;  // skeleton, shape(nspec * 2 + 2)
    std::vector<float> znodes; // shape(nspec * NGLL + NGRL)
    std::vector<float> jaco; // jacobian for GLL, shape(nspec + 1) dz / dxi
    std::vector<float> zstore; // shape(nglob)

    // element type for each medium
    int nspec_ac,nspec_el;
    int nspec_ac_grl,nspec_el_grl;
    std::vector<char> is_elastic, is_acoustic;
    std::vector<int> el_elmnts,ac_elmnts; // elements for each media, shape(nspec_? + nspec_?_grl)

    // unique array for acoustic/elastic
    int nglob_ac, nglob_el;
    std::vector<int> ibool_el, ibool_ac; // connectivity matrix, shape shape(nspec_? + nspec_?_grl)

    // density and elastic parameters
    std::vector<float> xrho_ac; // shape(nspec_ac * NGLL + nspec_ac_grl * NGRL)
    std::vector<float> xrho_el; // shape (nsepc_el * NGLL + nspec_el_grl * NGRL)

    // attenuation/type flag
    bool HAS_ATT;
    int SWD_TYPE; // =0 Love wave, = 1 for Rayleigh = 2 full aniso

    // vti media
    std::vector<float> xA,xC,xL,xeta,xN; // shape(nspec_el * NGLL+ nspec_el_grl * NGRL)
    std::vector<float> xQA,xQC,xQL,xQN; // shape(nspec_el * NGLL+ nspec_el_grl * NGRL), Q model

    // full anisotropy
    int nQmodel_ani; // no. of Q used for anisotropy
    std::vector<float> xC21; // shape(21,nspec_el * NGLL+ nspec_el_grl * NGRL)
    std::vector<float> xQani; // shape(nQmodel_ani,nspec_el * NGLL+ nspec_el_grl * NGRL)

    // fluid vti
    std::vector<float> xkappa_ac,xQk_ac;

    // fluid-elastic boundary
    int nfaces_bdry;
    std::vector<int> ispec_bdry; // shape(nfaces_bdry,2) (i,:) = [ispec_ac,ispec_el]
    std::vector<char> bdry_norm_direc; //  shape(nfaces_bdry), = 1 point from acoustic -> z direc
elastic

    int nz_tomo, nregions;
    std::vector<float> rho_tomo;
    std::vector<float> vpv_tomo,vph_tomo,vsv_tomo,vsh_tomo,eta_tomo;
```

```
    std::vector<float> QC_tomo,QA_tomo,QL_tomo,QN_tomo;
    std::vector<float> c21_tomo,Qani_tomo;
    std::vector<float> depth_tomo;
    std::vector<int> region_bdry; // shape(nregions,2)
    std::vector<int> iregion_flag; // shape(nspec + 1), return region flag

    // interface with layered model
    std::vector<char> is_el_reg, is_ac_reg; // shape(nregions)

    float PHASE_VELOC_MIN,PHASE_VELOC_MAX;

    // current frequency
    float freq;

    // public functions
    void read_model(const char *filename);
    void create_database(float freq,float phi);
    void print_model() const;
    void print_database() const;
    void allocate_1D_model(int nz0,int swd_type,int has_att);
    void create_model_attributes();

    // interpolate model
    void interp_model(const float *param,const std::vector<int> &elmnts,std::vector<float> &md) const;
    void project_kl(const float *frekl, float *kl_out) const;

    // private functions below
    // ==============================
    //
    void create_material_info_();

    // 1-D model
    void read_model_header_(const char *filename);
    void read_model_love_(const char *filename);
    void read_model_rayl_(const char *filename);
    void read_model_full_aniso_(const char *filename);

    // create SEM database
    void compute_minmax_veloc_(float phi,std::vector<float> &vmin,std::vector<float> &vmax);
    void create_db_love_();
    void create_db_rayl_();
    void create_db_aniso_();
};

} // namespace specswd




#endif
```

4.4 attenuation.hpp

```

#ifndef SPECSWD_ATT_TABLE_H_
#define SPECSWD_ATT_TABLE_H_

#include <complex>

namespace specswd
{

const int NSLS = 5;

std::complex<float> get_sls_modulus_factor(float freq,float Q);
void
get_sls_Q_derivative(float freq,float Qm,std::complex<float> &s,
                        std::complex<float> &dsdqi);

void get_C21_att(float freq,const float *Qm,int nQmodel,
                        std::complex<float> * c21,
                        int funcid=1);


}

#endif
```

4.5 GQTable.hpp

```
#ifndef SPECSWD_GQTABLE_H_
#define SPECSWD_GQTABLE_H_

#include <array>

namespace GQTable
{

const int NGLL = 7, NGRL = 20;
extern std::array<float,NGLL> xgll,wgll;
extern std::array<float,NGRL> xgrl,wgrl;
extern std::array<float,NGLL*NGLL> hprimeT,hprime; // hprimeT(i,j) = l'_i(xi_j)
extern std::array<float,NGRL*NGRL> hprimeT_grl,hprime_grl;

void initialize();

} // GQTable


#endif
```

4.6 iofunc.hpp

```
#ifndef SPECSWD_IOFUNC_H_
#define SPECSWD_IOFUNC_H_

#include <iostream>

namespace specswd
{


inline void __myfwrite(const void *__ptr, size_t __size, size_t __nitems, FILE *__stream)
{
    size_t size = fwrite(__ptr,__size,__nitems,__stream);
    if(size != __nitems) {
        printf("cannot write to binary!\n");
        exit(1);
    }
}


template<typename T>
void
write_binary_f(FILE *fp, const T *data, size_t n)
{
    // write integers of the size
    int size = (int)(n * sizeof(T));

    // integer front
    __myfwrite(&size,sizeof(int),1,fp);

    // data
    __myfwrite(data,sizeof(T),n,fp);

    // integer back
    __myfwrite(&size,sizeof(int),1,fp);
}

} // namespace specswd


#endif
```

4.7 schur.hpp

```
#ifndef SPECSWD_SCHUR_H_
#define SPECSWD_SCHUR_H_

#include <Eigen/Core>
#include <Eigen/Eigenvalues>

#ifdef SPECSWD_EGN_DOUBLE
typedef double realw;
#define LAPACKE_REAL(name) LAPACKE_d ## name
```

```
#define LAPACKE_CMPLX(name) LAPACKE_z ## name
#define LCREALW lapack_complex_double

#else
typedef float realw;
#define LAPACKE_REAL(name) LAPACKE_s ## name
#define LAPACKE_CMPLX(name) LAPACKE_c ## name
#define LCREALW lapack_complex_float
#endif

typedef std::complex<realw> crealw;


namespace specswd {

template<typename COMMTP=double,typename SAVETP=float>  void
schur_qz(int ng,
    Eigen::MatrixX<COMMTP> &A,
    Eigen::MatrixX<COMMTP> &B,
    std::vector<SAVETP> &Qmat,std::vector<SAVETP> &Zmat,
    std::vector<SAVETP> &Smat,std::vector<SAVETP> &Spmat
)
{
    static_assert(std::is_same_v<SAVETP,float> ||
                    std::is_same_v<SAVETP,scmplx>);

    // resize all matrices
    Qmat.resize(ng*ng); Zmat.resize(ng*ng);
    Smat.resize(ng*ng); Spmat.resize(ng*ng);

    // run QZ
    int sdim = 0;
    if constexpr (std::is_same_v<SAVETP,float>){
        Eigen::VectorX<COMMTP> alphar(ng),alphai(ng),beta(ng);
        if constexpr (std::is_same_v<realw,float>) {
            LAPACKE_REAL(gges)(
                LAPACK_COL_MAJOR,'V','V','N',nullptr,
                ng,A.data(),ng,B.data(),ng,&sdim,alphar.data(),
                alphai.data(),beta.data(),Qmat.data(),ng,
                Zmat.data(),ng
            );
            memcpy(Smat.data(),A.data(),A.size()*sizeof(A(0,0)));
            memcpy(Spmat.data(),B.data(),B.size()*sizeof(B(0,0)));
        }
        else {
            Eigen::MatrixX<COMMTP> Q1(ng,ng),Z1(ng,ng);
            LAPACKE_REAL(gges)(
                LAPACK_COL_MAJOR,'V','V','N',nullptr,
                ng,A.data(),ng,B.data(),ng,&sdim,alphar.data(),
                alphai.data(),beta.data(),Q1.data(),ng,
                Z1.data(),ng
            );

            // copy to Smat/Spmat
            for(int j = 0; j < ng; j ++) {
            for(int i = 0; i < ng; i ++) {
                int idx = j * ng + i;
                Smat[idx] = A(i,j);
                Spmat[idx] = B(i,j);
                Qmat[idx] = Q1(i,j);
                Zmat[idx] = Z1(i,j);
            }}
        }
    }
    else { // save type is scmplex
        Eigen::VectorX<COMMTP> alpha(ng),beta(ng);

        if constexpr (std::is_same_v<realw,float>) {
            LAPACKE_CMPLX(gges)(
                LAPACK_COL_MAJOR,'V','V','N',nullptr,
                ng,(LCREALW*)A.data(),ng,(LCREALW*)B.data(),
                ng,&sdim,(LCREALW*)alpha.data(),(LCREALW*)beta.data(),
                (LCREALW*)Qmat.data(),ng,
                (LCREALW*)Zmat.data(),ng
            );
            memcpy(Smat.data(),A.data(),A.size()*sizeof(A(0,0)));
            memcpy(Spmat.data(),B.data(),B.size()*sizeof(B(0,0)));
        }
        else {
            Eigen::MatrixX<COMMTP> Q1(ng,ng),Z1(ng,ng);
            LAPACKE_CMPLX(gges)(
                LAPACK_COL_MAJOR,'V','V','N',nullptr,
                ng,(LCREALW*)A.data(),ng,(LCREALW*)B.data(),
                ng,&sdim,(LCREALW*)alpha.data(),(LCREALW*)beta.data(),
                (LCREALW*)Q1.data(),ng,
                (LCREALW*)Z1.data(),ng
            );
```

```
            // copy to Smat/Spmat
            for(int j = 0; j < ng; j ++) {
            for(int i = 0; i < ng; i ++) {
                int idx = j * ng + i;
                Smat[idx] = A(i,j);
                Spmat[idx] = B(i,j);
                Qmat[idx] = Q1(i,j);
                Zmat[idx] = Z1(i,j);
            }}
        }
    }
}

}

#endif
```

4.8 frechet_op.hpp

```
#ifndef SPECSWD_FRECHET_OP_H_
#define SPECSWD_FRECHET_OP_H_


#include "shared/attenuation.hpp"
#include "shared/GQTable.hpp"

#define GET_REAL(dc_dm,loc) frekl_r[loc*size+id1] = dc_dm.real(); \
                            frekl_i[loc*size+id1] = dc_dm.imag();

namespace specswd
{

void inline
get_fQ_kl(int npts,std::complex<float> f_cmplx,
          const float *frekl_r,
          float *__restrict frekl_i)
{
    float f_real = f_cmplx.real();
    float f_imag = f_cmplx.imag();
    float fQi = 2. * f_imag / f_real;

    for(int ipt = 0; ipt < npts; ipt ++) {
        float dQidm = (frekl_i[ipt] * 2. - fQi * frekl_r[ipt]) / f_real;
        frekl_i[ipt] = dQidm;
    }
}

template<typename T = float > void
love_op_matrix(float freq,T c_M, T c_K,T c_E, const T *y,const T *x,
               int nspec_el,int nglob_el, const int *ibool_el,
               const float *jaco, const float *xN,
               const float *xL,const float *xQN,
               const float *xQL,float * __restrict frekl_r,
               float * __restrict frekl_i)
{
    // check template type
    static_assert(std::is_same_v<float,T> || std::is_same_v<std::complex<float>,T>);

    using namespace GQTable;
    std::array<T,NGRL> rW,lW;
    size_t size = nspec_el*NGLL + NGRL;
    for(int ispec = 0; ispec < nspec_el + 1; ispec ++) {
        int id = ispec * NGLL;
        float J = jaco[ispec]; // jacobians in this layers

        const bool is_gll = (ispec != nspec_el);
        const float *w = is_gll? wgll.data(): wgrl.data();
        const float *hp = is_gll? hprime.data(): hprime_grl.data();
        const int NGL = is_gll? NGLL : NGRL;;

        // cache displ in a element
        for(int i = 0; i < NGL; i ++) {
            int iglob = ibool_el[id+i];
            rW[i] = x[iglob];
            lW[i] = y[iglob];
            if constexpr (std::is_same_v<T,std::complex<float») {
                lW[i] = std::conj(lW[i]);
            }
        }

        // compute kernels
        T dc_drho{}, dc_dN{}, dc_dL{};
```

```
        T dc_dqni{}, dc_dqli{};
        T sn = 1., sl = 1.;
        T dsdqni{}, dsdqli{};
        for(int m = 0; m < NGL; m ++) {
            dc_drho = w[m] * J * rW[m] * lW[m] * c_M;

            // get sls derivative if required
            if constexpr (std::is_same_v<T,std::complex<float») {
                get_sls_Q_derivative(freq,xQN[id+m],sn,dsdqni);
                get_sls_Q_derivative(freq,xQL[id+m],sl,dsdqli);
                dsdqni *= xN[id+m];
                dsdqli *= xL[id+m];
            }

            // N kernel
            T temp = rW[m] * lW[m]  * J * w[m] * c_K;
            dc_dN = temp * sn;
            dc_dqni = temp * dsdqni;

            // L kernel
            T sx{},sy{};
            for(int i = 0; i < NGL; i ++) {
                sx += hp[m*NGL+i] * rW[i];
                sy += hp[m*NGL+i] * lW[i];
            }
            temp = sx * sy * w[m] / J * c_E;
            dc_dL = temp * sl;
            dc_dqli = temp * dsdqli;

            // copy to frekl
            int id1 = id + m;
            if constexpr (std::is_same_v<T,std::complex<float») {
                GET_REAL(dc_dN,0); GET_REAL(dc_dL,1);
                GET_REAL(dc_dqni,2); GET_REAL(dc_dqli,3);
                GET_REAL(dc_drho,4);
            }
            else {
                frekl_r[0*size+id1] = dc_dN;
                frekl_r[1*size+id1] = dc_dL;
                frekl_r[2*size+id1] = dc_drho;
            }
        }
    }
}

template<typename T = float >
void
rayl_op_matrix(float freq,T c_M,T c_K, T c_E,const T *y, const T *x,
                int nspec_el,int nspec_ac,int nspec_el_grl,int nspec_ac_grl,int nglob_el,
                int nglob_ac, const int *el_elmnts,const int *ac_elmnts,
                const int* ibool_el, const int* ibool_ac,
                const float *jaco,const float *xrho_el,const float *xrho_ac,
                const float *xA, const float *xC,const float *xL,const float *xeta,
                const float *xQA, const float *xQC,const float *xQL,
                const float *xkappa_ac, const float *xQk_ac,
                float *__restrict frekl_r,
                float *__restrict frekl_i)
{
    // check template type
    static_assert(std::is_same_v<float,T> || std::is_same_v<std::complex<float>,T>);

    // constants
    using namespace GQTable;
    size_t size = nspec_el * NGLL + nspec_el_grl * NGRL +
                  nspec_ac * NGLL + nspec_ac_grl * NGRL;

    // loop elastic elements
    std::array<T,NGRL> U,V,lU,lV; //left/right eigenvectors in on element
    for(int ispec = 0; ispec < nspec_el + nspec_el_grl; ispec ++) {
        int iel = el_elmnts[ispec];
        int id = ispec * NGLL;

        // jacobian
        float J = jaco[iel];

        // get const arrays
        const bool is_gll = (ispec != nspec_el);
        const float *weight = is_gll? wgll.data(): wgrl.data();
        const float *hp = is_gll? hprime.data(): hprime_grl.data();
        const int NGL = is_gll? NGLL : NGRL;

        // cache U,V and lU,lV
        for(int i = 0; i < NGL; i ++) {
            int iglob = ibool_el[id + i];
            U[i] = x[iglob];
            V[i] = x[iglob + nglob_el];
            lU[i] = y[iglob];
```

```
            lV[i] = y[iglob + nglob_el];
            if  constexpr (std::is_same_v<T,std::complex<float») {
                lU[i] = std::conj(lU[i]);
                lV[i] = std::conj(lV[i]);
            }
        }

        // compute kernel
        T dc_drho{}, dc_dA{}, dc_dC{}, dc_dL{};
        T dc_deta{}, dc_dQci{},dc_dQai{},dc_dQli{};
        const T two = 2.;
        for(int m = 0; m < NGL; m ++) {
            T temp = weight[m] * J * c_M;
            dc_drho = temp * (U[m] * lU[m] + V[m] * lV[m]);

            // get sls factor if required
            T sa = 1.,sl = 1.,sc = 1.;
            T dsdqai{},dsdqci{},dsdqli{};
            T C = xC[id+m], A = xA[id+m], L = xL[id+m], eta = xeta[m];
            if constexpr (std::is_same_v<T,std::complex<float») {
                get_sls_Q_derivative(freq,xQA[id+m],sa,dsdqai);
                get_sls_Q_derivative(freq,xQC[id+m],sc,dsdqci);
                get_sls_Q_derivative(freq,xQL[id+m],sl,dsdqli);
                dsdqai *= A;
                dsdqci *= C;
                dsdqli *= L;
            }

            // K matrix
            // dc_dA
            temp = weight[m] * J * U[m] * lU[m] * c_K;
            dc_dA = temp * sa; dc_dQai = temp * dsdqai;

            // dc_dL
            temp = weight[m] * J * V[m] * lV[m] * c_K;
            dc_dL = temp * sl; dc_dQli = temp * dsdqli;

            // Ematrix
            T sx{},sy{},lsx{},lsy{};
            for(int i = 0; i < NGL; i ++) {
                sx += hp[m*NGL+i] * U[i];
                sy += hp[m*NGL+i] * V[i];
                lsx += hp[m*NGL+i] * lU[i];
                lsy += hp[m*NGL+i] * lV[i];
            }

            // E1
            temp = weight[m] / J * sx * lsx * c_E;
            dc_dL += temp * sl; dc_dQli += temp * dsdqli;

            // E3
            temp = weight[m] / J * sy * lsy * c_E;
            dc_dC = temp * sc; dc_dQci = temp * dsdqci;

            // K2,  d / dm_k sum_{ij} w_j F_j hpT(i,j) U_j lV_i
            // = \sum_{i} w_k dF/dm_k hpT(i,k) U_k lV_i = lsy * w_k * U_k * dF/dm_k
            temp = weight[m] * U[m] * lsy * c_K;
            dc_deta = temp * (A*sa - two*L*sl);
            temp *= eta;
            dc_dA += temp * sa; dc_dQai += temp * dsdqai;
            dc_dL += - temp * two * sl; dc_dQli += -temp * two * dsdqli;

            // K2, -d / dm_k sum_{ij} w_i L_i hp(i,j) U_j lV_i
            // = - \sum_{j} w_k dL/dm_k hp(j,k) U_j lV_k = -sx * w_k * lV_k dL/dm_k
            temp = -weight[m] * lV[m] * sx * c_K;
            dc_dL += - temp * two * sl; dc_dQli += -temp * two * dsdqli;

            //E2 \sum_{j} w_k dF/dm_k hp(j,k) V_j lU_k = -sx * w_k * lV_k dF/dm_k
            temp = weight[m] * lU[m] * sy * c_E;
            dc_deta = temp * (A*sa - two*L*sl);
            temp *= eta;
            dc_dA += temp * sa; dc_dQai += temp * dsdqai;
            dc_dL += - temp * two * sl; dc_dQli += -temp * two * dsdqli;

            // E2 -lsx * w_k * V_k * dL/dm_k
            temp = -weight[m] * V[m] * lsx * c_E;
            dc_dL += - temp * two * sl; dc_dQli += -temp * two * dsdqli;

            // copy them to frekl
            int id1 = iel * NGLL + m;
            if constexpr (std::is_same_v<T,float>) {
                frekl_r[0*size+id1] = dc_dA;
                frekl_r[1*size+id1] = dc_dC;
                frekl_r[2*size+id1] = dc_dL;
                frekl_r[3*size+id1] = dc_deta;
                frekl_r[5*size+id1] = dc_drho;
            }
```

```
            else {
                GET_REAL(dc_dA,0);
                GET_REAL(dc_dC,1);
                GET_REAL(dc_dL,2);
                GET_REAL(dc_deta,3);
                GET_REAL(dc_dQai,4);
                GET_REAL(dc_dQci,5);
                GET_REAL(dc_dQli,6);
                GET_REAL(dc_drho,9);
            }
        }
    }

    // acoustic eleemnts
    std::array<T,NGRL> chi,lchi;
    for(int ispec = 0; ispec < nspec_ac + nspec_ac_grl; ispec ++) {
        int iel = ac_elmnts[ispec];
        int id = ispec * NGLL;

        // const arrays
        const bool is_gll = (ispec != nspec_ac);
        const float *weight = is_gll? wgll.data(): wgrl.data();
        const float *hp = is_gll? hprime.data(): hprime_grl.data();
        const int NGL = is_gll? NGLL : NGRL;

        // jacobians
        float J = jaco[iel];

        // cache chi and lchi in one element
        for(int i = 0; i < NGL; i ++) {
            int iglob = ibool_ac[id + i];
            chi[i] = (iglob == -1) ? 0: x[iglob+nglob_el*2];
            lchi[i] = (iglob == -1) ? 0.: y[iglob+nglob_el*2];
            if  constexpr (std::is_same_v<T,std::complex<float») {
                lchi[i] = std::conj(lchi[i]);
            }
        }

        // derivatives
        T dc_dkappa{},dc_drho{}, dc_dqki{};
        T sk = 1., dskdqi = 0.;
        for(int m = 0; m < NGL; m ++ ){
            // copy material
            float rho = xrho_ac[id+m];
            float kappa = xkappa_ac[id+m];
            if constexpr (std::is_same_v<T,std::complex<float») {
                get_sls_Q_derivative(freq,xQk_ac[id+m],sk,dskdqi);
                dskdqi *= kappa;
            }

            // kappa kernel
            T temp = -c_M * weight[m]* J*chi[m] * lchi[m] / (sk * kappa) / (sk * kappa);
            dc_dkappa = temp * sk;
            dc_dqki =  temp * dskdqi;

            dc_drho = - c_K * weight[m]* J * chi[m] * lchi[m] / rho / rho;

            T sx{},sy{};
            for(int i = 0; i < NGL; i ++) {
                sx += hp[m*NGL+i] * chi[i];
                sy += hp[m*NGL+i] * lchi[i];
            }
            dc_drho += -c_E * weight[m] / J / (rho*rho) * sx * sy;

            // copy to frekl
            int id1 = iel * NGLL + m;
            if constexpr (std::is_same_v<T,float>) {
                frekl_r[4*size+id1] = dc_dkappa;
                frekl_r[5*size+id1] = dc_drho;
            }
            else {
                GET_REAL(dc_dkappa,7);
                GET_REAL(dc_dqki,8);
                GET_REAL(dc_drho,9);
            }
        }
    }
}

// /**
//  * @brief compute coef * y^dag @ d( (w^2 M -E) - k^2 K)/dm_i @ x dm_i
//  * @param freq current frequency
//  * @param c current phase velocity
//  * @param coef derivative scaling coefs
//  * @param y/x  dot vector,shape(nglob_el*2+nglob_ac)
//  * @param nspec_el/nglob_el mesh nelemnts/unique points for elastic
//  * @param nspec_ac/nglob_ac mesh nelemnts/unique points for acoustic
```

```
//  * @param nspec_el/ac_grl no. of GRL elements
//  * @param ibool_el elastic connectivity matrix, shape(nspec_el*NGLL+nspec_el_grl*NGRL)
//  * @param ibool_ac elastic connectivity matrix, shape(nspec_ac*NGLL+nspec_ac_grl*NGRL)
//  * @param jaco jacobian matrix, shape (nspec_el + 1)
//  * @param xA/xC/xL/xeta/xQA/xQC/xQL/xrho elastic model parameters,ibool_el.shape
//  * @param xkappa_ac/xQk_ac/xrho_ac acoustic model parameters, ibool_ac.shape
//  * @param frekl_r dc/d(A/C/L/kappa/rho) (elastic) or dc/d(A/C/L/QA/QC/QL/kappa/Qk/rho) (anelstic)
//  * @param frekl_i nullptr or dqc/d(A/C/L/QA/QC/QL/kappa/Qk/rho) (anelstic)
//  * @note frekl_r and frekl_i should be set to 0 before calling this routine
//  */
// template<typename T = float >
// void
// rayl_deriv_op(float freq,T c,T coef,const T *y, const T *x,
//                 int nspec_el,int nspec_ac,int nspec_el_grl,int nspec_ac_grl,int nglob_el,
//                 int nglob_ac, const int *el_elmnts,const int *ac_elmnts,
//                 const int* ibool_el, const int* ibool_ac,
//                 const float *jaco,const float *xrho_el,const float *xrho_ac,
//                 const float *xA, const float *xC,const float *xL,const float *xeta,
//                 const float *xQA, const float *xQC,const float *xQL,
//                 const float *xkappa_ac, const float *xQk_ac,
//                 float *__restrict frekl_r,
//                 float *__restrict frekl_i)
// {
//     // check template type
//     static_assert(std::is_same_v<float,T> || std::is_same_v<std::complex<float>,T>);

//     // constants
//     using namespace GQTable;
//     size_t size = nspec_el *  NGLL + nspec_el_grl * NGRL +
//                   nspec_ac * NGLL + nspec_ac_grl * NGRL;
//     T om = 2 * M_PI * freq;
//     T k2 = std::pow(om / c,2);

//     // loop elastic elements
//     std::array<T,NGRL> U,V,lU,lV;
//     for(int ispec = 0; ispec < nspec_el + nspec_el_grl; ispec ++) {
//         int iel = el_elmnts[ispec];
//         int id = ispec * NGLL;

//         const float *weight = wgll.data();
//         const float *hp = hprime.data();
//         int NGL = NGLL;

//         // jacobian
//         float J = jaco[iel];

//         // grl case
//         if(ispec == nspec_el) {
//             weight = wgrl.data();
//             hp = hprime_grl.data();
//             NGL = NGRL;
//         }

//         // cache U,V and lU,lV
//         for(int i = 0; i < NGL; i ++) {
//             int iglob = ibool_el[id + i];
//             U[i] = x[iglob] * coef;
//             V[i] = x[iglob + nglob_el] * coef;
//             lU[i] = y[iglob];
//             lV[i] = y[iglob + nglob_el];
//             if  constexpr  (std::is_same_v<T,std::complex<float») {
//                 lU[i] = std::conj(lU[i]);
//                 lV[i] = std::conj(lV[i]);
//             }
//         }

//        // compute kernel
//         T dc_drho{}, dc_dA{}, dc_dC{}, dc_dL{};
//         T dc_deta{}, dc_dQci{},dc_dQai{},dc_dQli{};
//         const T two = 2.;
//         for(int m = 0; m < NGL; m ++) {
//             T temp = weight[m] * J;
//             dc_drho = temp * om * om *
//                 (U[m] * lU[m] + V[m] * lV[m]);

//             // get sls factor if required
//             T sa = 1.,sl = 1.,sc = 1.;
//             T dsdqai{},dsdqci{},dsdqli{};
//             float C = xC[id+m], A = xA[id+m],
//                   L = xL[id+m], eta = xeta[m];
//             if constexpr (std::is_same_v<T,std::complex<float») {
//                 get_sls_Q_derivative(freq,xQA[id+m],sa,dsdqai);
//                 get_sls_Q_derivative(freq,xQC[id+m],sc,dsdqai);
//                 get_sls_Q_derivative(freq,xQL[id+m],sl,dsdqai);
//                 dsdqai *= A;
//                 dsdqci *= C;
//                 dsdqli *= L;
```

```
//             }

//             // K matrix
//             // dc_dA
//             temp = -weight[m] * J * k2 * U[m] * lU[m];
//             dc_dA = temp * sa; dc_dQai = temp * dsdqai;

//             // dc_dL
//             temp = -weight[m] * J * k2 * V[m] * lV[m];
//             dc_dL = temp * sl; dc_dQli = temp * dsdqli;

//             // Ematrix
//             T sx{},sy{},lsx{},lsy{};
//             for(int i = 0; i < NGL; i ++) {
//                 sx += hp[m*NGL+i] * U[i];
//                 sy += hp[m*NGL+i] * V[i];
//                 lsx += hp[m*NGL+i] * lU[i];
//                 lsy += hp[m*NGL+i] * lV[i];
//             }
//             temp = -weight[m] / J * sx * lsx;
//             dc_dL += temp * sl; dc_dQli += temp * dsdqli;

//             temp = - weight[m] / J * sy * lsy;
//             dc_dC = temp * sc; dc_dQci = temp * dsdqci;

//             // eta
//             temp = - (k2 * weight[m] * U[m] * lsy +
//                         weight[m] * lU[m] * sy);
//             dc_deta = temp * (A*sa - two*L*sl);

//             temp *= eta;
//             dc_dA += temp * sa; dc_dQai += temp * dsdqai;
//             dc_dL += - temp * two * sl; dc_dQli += -temp * two * dsdqli;

//             temp = k2 * weight[m] * lV[m] * sx + weight[m] * V[m] * lsx;
//             temp *= coef;
//             dc_dL +=  temp * sl;
//             dc_dQli += temp * dsdqli;

//             // copy them to frekl
//             int id1 = iel * NGLL + m;
//             if constexpr (std::is_same_v<T,float>) {
//                 frekl_r[0*size+id1] = dc_dA;
//                 frekl_r[1*size+id1] = dc_dC;
//                 frekl_r[2*size+id1] = dc_dL;
//                 frekl_r[3*size+id1] = dc_deta;
//                 frekl_r[5*size+id1] = dc_drho;
//             }
//             else {
//                 GET_REAL(dc_dA,0);
//                 GET_REAL(dc_dC,1);
//                 GET_REAL(dc_dL,2);
//                 GET_REAL(dc_deta,3);
//                 GET_REAL(dc_dQai,4);
//                 GET_REAL(dc_dQci,5);
//                 GET_REAL(dc_dQli,6);
//                 GET_REAL(dc_drho,9);
//             }
//         }
//     }

//     // acoustic eleemnts
//     std::array<T,NGRL> chi,lchi;
//     for(int ispec = 0; ispec < nspec_ac + nspec_ac_grl; ispec ++) {
//         int iel = ac_elmnts[ispec];
//         int id = ispec * NGLL;
//         const float *weight = wgll.data();
//         const float *hp = hprime.data();
//         int NGL = NGLL;

//         // jacobians
//         float J = jaco[iel];

//         // grl case
//         if(ispec == nspec_ac) {
//             weight = wgrl.data();
//             hp = hprime_grl.data();
//             NGL = NGRL;
//         }

//         // cache chi and lchi in one element
//         for(int i = 0; i < NGL; i ++) {
//             int iglob = ibool_ac[id + i];
//             chi[i] = (iglob == -1) ? 0: x[iglob+nglob_el*2] * coef;
//             lchi[i] = (iglob == -1) ? 0.: y[iglob+nglob_el*2];
//             if  constexpr (std::is_same_v<T,std::complex<float>>) {
//                 lchi[i] = std::conj(lchi[i]);
```

```
//              }
//          }

//          // derivatives
//          T dc_dkappa{},dc_drho{}, dc_dqki{};
//          T sk = 1., dskdqi = 0.;
//          for(int m = 0; m < NGL; m ++ ){
//              // copy material
//              float rho = xrho_ac[id+m];
//              float kappa = xkappa_ac[id+m];
//              if constexpr (std::is_same_v<T,std::complex<float») {
//                  get_sls_Q_derivative(freq,xQk_ac[id+m],sk,dskdqi);
//                  dskdqi *= kappa;
//              }

//              // kappa kernel
//              T temp = std::pow(om/(sk * kappa),2) *weight[m]* J*
//                          chi[m] * lchi[m];
//              dc_dkappa = temp * sk;
//              dc_dqki =  temp * dskdqi;

//              dc_drho = -k2 * std::pow(om/rho,2) *weight[m]* J*
//                          chi[m] * lchi[m];

//              T sx{},sy{};
//              for(int i = 0; i < NGL; i ++) {
//                  sx += hp[m*NGL+i] * chi[i];
//                  sy += hp[m*NGL+i] * lchi[i];
//              }
//              dc_drho += weight[m] / J / (rho*rho) * sx * sy;

//              // copy to frekl
//              int id1 = iel * NGLL + m;
//              if constexpr (std::is_same_v<T,float>) {
//                  frekl_r[4*size+id1] = dc_dkappa;
//                  frekl_r[5*size+id1] = dc_drho;
//              }
//              else {
//                  GET_REAL(dc_dkappa,7);
//                  GET_REAL(dc_dqki,8);
//                  GET_REAL(dc_drho,9);
//              }
//          }
//      }
// }


} // namespace specswd

#undef GET_REAL

#endif
```

4.9 vti.hpp

```
#ifndef SPECSWD_SOLVER_H_
#define SPECSWD_SOLVER_H_

#include "mesh/mesh.hpp"

#include <complex>
#include <vector>


namespace specswd
{

typedef std::complex<float>  scmplx;

class SolverLove {

private:
    // solver matrices
    std::vector<float> Mmat,Emat,Kmat;
    std::vector<scmplx> CMmat,CEmat,CKmat;

    // QZ matrix all are column major
    std::vector<float> Qmat_,Zmat_,Smat_,Spmat_; // column major!
    std::vector<scmplx> cQmat_,cZmat_,cSmat_,cSpmat_;

public:

    // eigenfunctions/values
```

```
    void prepare_matrices(const Mesh &M);
    void compute_egn(const Mesh &M,
                    std::vector<float> &c,
                    std::vector<float> &egn,
                    bool use_qz=false);
    void compute_egn_att(const Mesh &M,
                        std::vector<scmplx> &c,
                        std::vector<scmplx> &egn,
                        bool use_qz=false);

    // group velocity
    float group_vel(const Mesh &M,float c,const float *egn) const;
    scmplx group_vel_att(const Mesh &M,scmplx c, const scmplx *egn) const ;

    // phase velocity kernels
    void compute_phase_kl(const Mesh &M,
                        float c,const float *egn,
                        std::vector<float> &frekl) const;
    void compute_phase_kl_att(const Mesh &M,
                        scmplx c, const scmplx *egn,
                        std::vector<float> &frekl_c,
                        std::vector<float> &frekl_q) const;
    // group kernel
    void compute_group_kl(const Mesh &M,
                        float c,const float *egn,
                        std::vector<float> &frekl) const;

    void compute_group_kl_att(const Mesh &M,
                        scmplx c, scmplx u, const scmplx *egn,
                        std::vector<float> &frekl_u,
                        std::vector<float> &frekl_q) const;
    // tranforms
    void egn2displ(const Mesh &M,float c,
                    const float *egn, float * __restrict displ) const;
    void egn2displ_att(const Mesh &M,scmplx c,const scmplx *egn,
                      scmplx * __restrict displ) const;
    void transform_kernels(const Mesh &M,std::vector<float> &frekl) const;
};

class SolverRayl {

private:
    // solver matrices
    std::vector<float> Mmat,Emat,Kmat;
    std::vector<scmplx> CMmat,CEmat,CKmat;

    // QZ matrix all are column major
    std::vector<float> Qmat_,Zmat_,Smat_,Spmat_; // column major!
    std::vector<scmplx> cQmat_,cZmat_,cSmat_,cSpmat_;

public:
    void prepare_matrices(const Mesh &M);
    void compute_egn(const Mesh &M,
                    std::vector<float> &c,
                    std::vector<float> &ur,
                    std::vector<float> &ul,
                    bool use_qz=false);
    void compute_egn_att(const Mesh &M,
                        std::vector<scmplx> &c,
                        std::vector<scmplx> &ur,
                        std::vector<scmplx> &ul,
                        bool use_qz=false);

    // group velocity
    float group_vel(const Mesh &M,
                    float c,const float *ur,
                    const float *ul) const;
    scmplx group_vel_att(const Mesh &M,
                        scmplx c, const scmplx *ur,
                        const scmplx *ul) const;

    // phase velocity kernels
    void compute_phase_kl(const Mesh &M,
                        float c,const float *ur,
                        const float *ul,
                        std::vector<float> &frekl) const;
    void compute_phase_kl_att(const Mesh &M,
                        scmplx c, const scmplx *ur,
                        const scmplx *ul,
                        std::vector<float> &frekl_c,
                        std::vector<float> &frekl_q) const;

    // group velocity kernels
    void compute_group_kl(const Mesh &M,
                        float c,const float *ur,
                        const float *ul,
                        std::vector<float> &frekl) const;
```

```
    void compute_group_kl_att(const Mesh &M,
                        scmplx c, scmplx u,
                        const scmplx *ur,const scmplx *ul,
                        std::vector<float> &frekl_u,
                        std::vector<float> &frekl_q) const;


    // transforms
    void egn2displ(const Mesh &M,float c,
                    const float*egn, float * __restrict displ) const;
    void egn2displ_att(const Mesh &M,scmplx c,const scmplx *egn,
                        scmplx * __restrict displ) const;
    void transform_kernels(const Mesh &M,std::vector<float> &frekl) const;
};


} // namespace specswd




#endif
```

Index