spectral-wave-data Documentation

Release 1.0.0-rc1

Jens B. Helmers

Apr 12, 2020

CONTENTS:

This is the documentation for the GitHub organization SpectralWaveData.

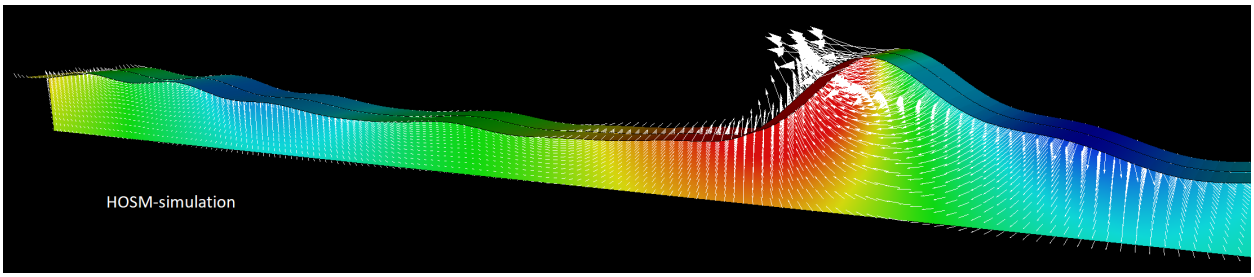


Fig. 1: Long crested deep water freak wave. Hs=13.5m and Tp=14.5s. Potential and velocity distribution.

CHAPTER

ONE

INTRODUCTION

The open-source GitHub organization SpectralWaveData hosts several repositories related to the spectral_wave_data API.

The main purpose of the **spectral_wave_data** API is to establish an open interface for how to exchange spectral ocean wave kinematics between computer programs.

Calculation of ocean wave kinematics is not only relevant for environmental assessments, but is also highly important for predicting wave induced motions and loads on marine structures.

In lack of proper tools, knowledge and computer capacity, realistic critical wave trains have typical been replaced by idealized regular waves or linear sea states, when assessing responses of marine structures. However, extensive recent research documents important physical effects not present in these simplistic wave models.

The simplistic wave models can be described by a small set of parameters. Consequently, it has not been important to have standardized field interfaces between programs. Ad-hoc input transformations, like adjustment of phase definitions, were sufficient prior to actual calculations.

The more realistic critical wave trains are complicated to describe.

- What are the important characteristics of such wave trains?
- What is the appropriate mathematical model(s)?
- How should my software deal with such wave fields?

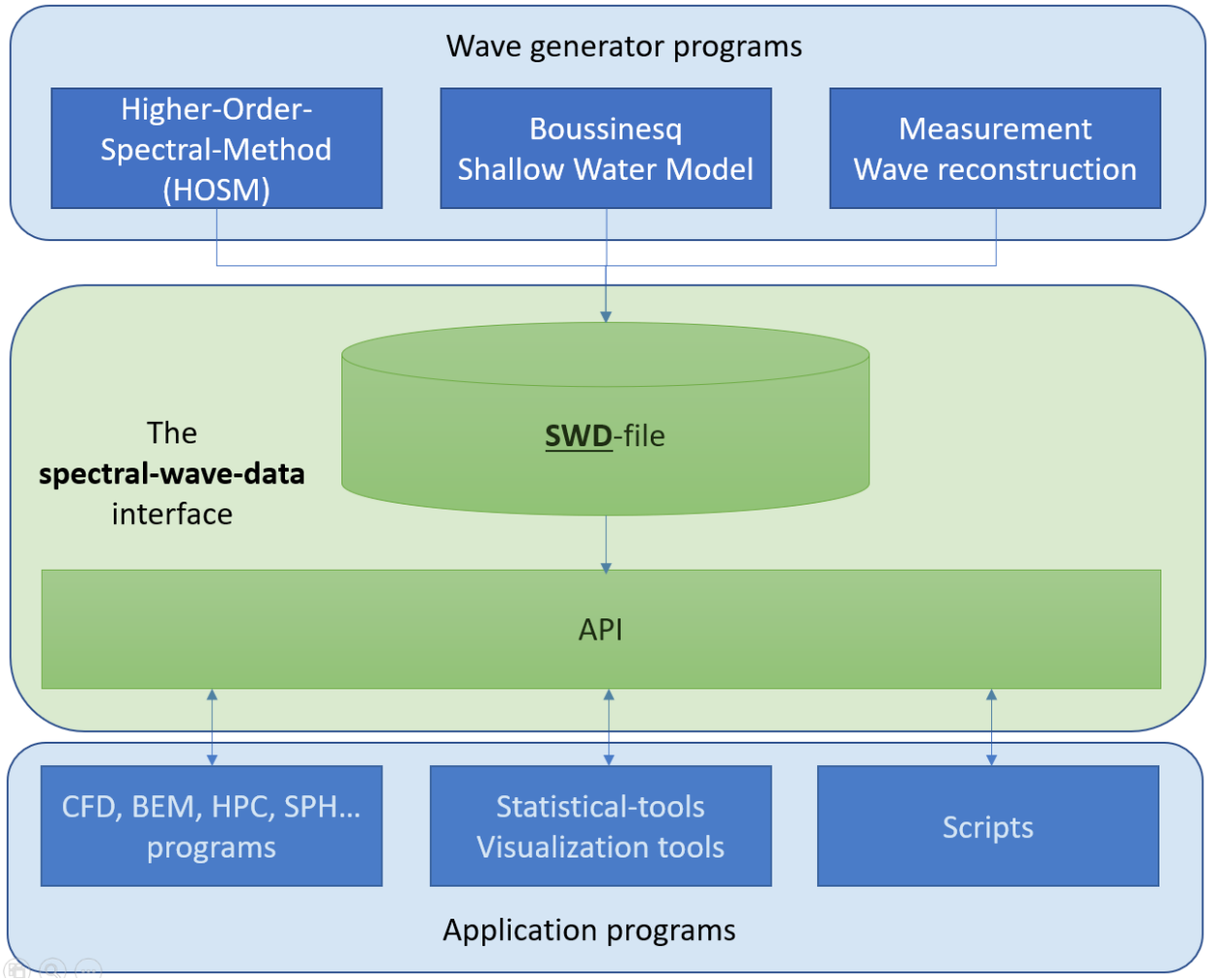
These questions are still challenging, and the answers may differ depending on the considered phenomena. However, the **spectral_wave_data** API is an attempt to push forward in this respect. Not to provide the answers to all these questions, but to facilitate such investigations.

CHAPTER

TWO

TERMINOLOGY

In this documentation we apply a vocabulary based on the following figure.



2.1 Wave generator

A wave generator is any program or script capable of constructing a sound SWD-file. There are no restrictions on theoretical models applied in wave generators.

2.2 Application program

Any program or scripts applying a SWD-file and the associated API for evaluating the spectral wave kinematics is referred to as an application program.

2.3 The Spectral-Wave-Data (SWD) file

All SWD files comply with the file format as *specified* in this documentation. It contains the spectral description of a particular wave field as explained in the theory section.

2.4 API

This documentation describes an object oriented interface for evaluation of wave kinematics based on the contents of a SWD file. This interface is considered the SWD-API (the Application-Programming-Interface).

This API, including an implementation, is distributed as part of the spectral_wave_data repository. Source code and precompiled libraries are available in several promgramming languages. C++, C, Fortrran-2008, Python-2 and Python-3.

Site specific or vendor optimized implementations complying with this API are considered SWD compliant if numerical results do not differ significantly from results using the code in this repository.

CHAPTER

THREE

RATIONALES

3.1 Why spectral waves?

There are several advanced wave models in academic and industrial applications. All with pros and cons. A particular challenge is that a wide range of length and time scales are relevant for ocean wave kinematics. Some effects like modulation instabilities may develop over a long range of time and space, while formation of breaking waves may occur very locally and quickly. Consequently, there are no model practical for all assessments.

This API is exclusively dedicated to spectral wave formulations for the following reasons

- It provides a continuous and smooth description of the wave field. Hence, *application programs* using different numerical models and discretizations may evaluate the same wave field at arbitrary locations without field interpolation.
- The wave field is completely described by a relative small set of data. This makes it possible to do large scale computations in time and space.
- Spectral wave fields are typical applied as boundary conditions in advanced *application programs*. Such programs may simulate overturning and breaking waves and/or responses of marine structures.
- Spectral waves can describe all relevant classical wave models (Airy, Stokes, etc) and linear and non-linear sea states. Capturing several important non-linear physical effects are documented for all levels of water depths including bathymetry, for both long and short-crested waves.
- The required set of kinematics to evaluate may differ between *application programs*. E.g. perturbation based programs may need to evaluate higher-order gradients of the velocity potential while a CFD program may only need surface elevation and particle velocities. Without any such considerations when first constructing the field, the spectral_wave_data API can provide all relevant kinematics without introducing further mathematical approximations.
- *Wave generators* are not limited to e.g. Higher-Order-Spectral-Method HOS(M) or Boussinesq solvers. Output from other numerical or experimental wave fields may be fitted to a spectral formulation. This is e.g. relevant when doing wave reconstruction based on experimental data from surface elevation probes. Such wave fields may then be analyzed using any *application program*.

3.2 Why introduce the SWD file?

The main reasons for introducing the SWD file and not to couple programs directly at the source code level are

- The spectral wave field analyzed by *application programs* may come from very different sources. E.g.
  - Different institutions (academic and industry)
  - Different theories (e.g. HOSM vs Boussinesq)
  - Fitting of wave elevation probes (different model basins)
- It is technically complicated to couple *wave generators* and *application programs* at the source code level. Especially if you need wave fields from several sources as described above.
- It takes much less effort to assess different sources using the file approach, which we think benefit both the research and industry communities.
- There might be severe copyright issues when coupling software at the source code.
- If you need to assess the same wave field several times (e.g. assess two different loading conditions of a ship) it is faster to decode the SWD file than to resolve the same ambient wave field for every case.
- Reassessing short critical time windows from from a long simulation, using a more advanced solver (e.g. a CFD-solver instead of a BEM-solver applied for the long simultion) is complicated if not using the interface file approach.
- No, the SWD file is usually not huge. Typical size for a 20-min simulation of irregular seas is 10-100 Mb.

3.3 Why use a binary C stream format for the SWD file?

The reasons for this decision are

- Standard Fortran, C, C++, Python and Matlab support I/O of binary C streams.
- No additional libraries are required to deal with the spectral_wave_data API.
- The file format is simple and I/O is efficiently handled with minimum storage requirements.
- Nothing wrong with HDF5 and other advanced libraries, some people just don't like the idea of compiling and linking very large and complicated libraries for smaller tasks. This API should be simple to apply in all *wave generators* and *application programs*. Small is beautiful.

3.4 Why don't you implement vector operations?

For the following reasons most kinematics is evaluated for individual locations only.

- The cost of function calls is small compared to kinematic calculations for irregular seas.
- Where to vectorize or parallelize the code, is highly dependent on the actual application program.
- Parallel computations are more complicated in multi-language software
- Simplicity is important in the open source.

You may always optimize the implementation for your specific application as you like, and still enjoy the benefits of the open API.

3.5 Will there be FFT based evaluations?

For a small set of specific features, like global grid evaluation of surface elevation, there will be FFT based methods. Different API and implementations are considered.

For other kinematics the implemented recursive schemes are superior with respect to speed and accuracy when doing evaluation at arbitrary locations.

3.6 Why Fortran with C/C++/Python wrappers?

This API is comprehensive with many methods supporting many spectral formulations that may use several numerical implementations. That is why the initial release only provides one native implementation and apply wrappers for the other languages.

Fortran was selected because it is fast and has an ISO standardized interface to C which again has well established interfaces to C++ and Python. It would be difficult to support Fortran based on other native implementations.

If somebody is willing to make and maintain a complete native implementation in other languages than Fortran-2008 that is great, please contribute. Not only C/C++/Python but other languages like Matlab may be relevant.

For your own implementations you may of course apply any language you like and still enjoy the benefits of the open API.

CHAPTER

FOUR

THEORY

In this chapter we define the wave formulation as applied in the spectral_wave_data API.

The *wave generator* may apply other formulations but is required to do eventual output transformations to comply with the definitions described in this document. Such modifications are typical minor, limited to apply sign shifts, complex conjugate etc. in output statements.

The *wave generator* producing the wave field do not consider the location, orientation and speed of eventual structures. As a consequence, the produced ambient wave field can be reused for assessing different structural configurations.

4.1 The SWD coordinate system

The spectral_wave_data (SWD) wave field is described in an earth fixed right-handed Cartesian coordinate system $\mathbf{x} = (x, y, z)$ where $z = 0$ coincides with the calm free surface and the z-axis is pointing upwards. This is the coordinate system applied in the *SWD file*.

For long crested seas, waves are assumed to propagate in the positive x-direction.

For short crested seas, waves may propagate in all directions. Consequently, the orientation of the x-direction is not specified by the API. However, if there is a time independent main propagation direction it is expected to be in the positive x-direction. In case of varying bathymetry this expectation may not apply.

4.2 The application defined wave coordinate system

The spectral_wave_data API is designed to be applied in a wide range of *application programs*. Consequently, an earth fixed, right-handed, Cartesian wave coordinate systems $\bar{\mathbf{x}} = (\bar{x}, \bar{y}, \bar{z})$ related to the *application program* may differ from the SWD coordinate system. It is assumed that $\bar{z} = 0$ coincides with the calm free surface and the $\bar{z}$-axis is pointing upwards.

The relation between the SWD and the wave coordinate system applied in the *application program* is defined by the 3 spatial shift parameters x_0, y_0 and β. The location of the application origin is $(x_0, y_0, 0)$ observed from the SWD system. The x-axis is rotated β relative to the $\bar{x}$-axis.

These relations are expressed as

$$x - x_0 = \bar{x}\cos\beta + \bar{y}\sin\beta, \qquad y - y_0 = -\bar{x}\sin\beta + \bar{y}\cos\beta$$

$$z = \bar{z}, \qquad t - t_0 = \bar{t}, \qquad t_0 \geq 0$$

The shift parameter t_0 relates the time t defined in the SWD system relative to the time $\bar{t}$ in the *application program*.

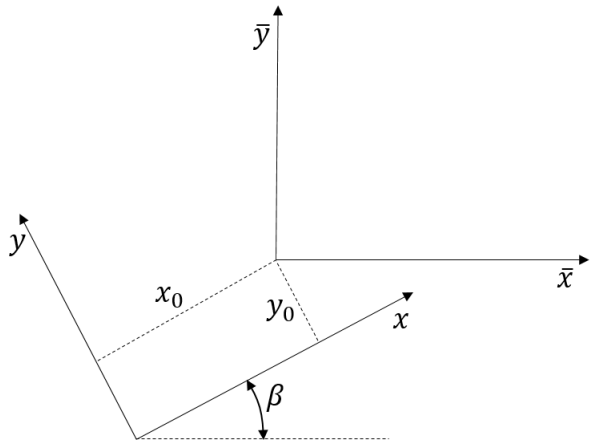


Fig. 1: The relation between the SWD $\mathbf{x}$ and the earth fixed *application program* coordinate system $\bar{\mathbf{x}}$. In this example x_0, y_0 and β are all positive quantities.

4.3 The spectral formulation

Assuming an ocean of ideal fluid, the incident velocity potential $\phi(\mathbf{x}, t)$ and the corresponding single valued surface elevation $\zeta(x, y, t)$ at time t are expressed with respect to the SWD coordinate system as

$$\phi(\mathbf{x}, t) = \sum_{j=1}^{n} \mathcal{R}\rceil\Big\{c_j(t)\,\psi_j(\mathbf{x})\Big\}, \qquad \zeta(x, y, t) = \sum_{j=1}^{n} \mathcal{R}\rceil\Big\{h_j(t)\,\xi_j(x, y)\Big\}$$

where both sets of n explicit shape functions $\psi_j(\mathbf{x})$ and $\xi_j(x, y)$ are orthogonal on the global considered space. The functions $\psi_j(\mathbf{x})$ are harmonic. The number of shape functions n defines a time independent spatial resolution of the wave field. Consequently, constrained by the given resolution n a wide range of wave fields can be described given the shape functions and the temporal amplitudes $c_j(t)$ and $h_j(t)$. Obvious exceptions are overturning waves and viscous flows, but a large class of important nonlinear wave trains may be described in addition to all the classical perturbation wave models (Airy, Stokes, Stream waves, etc.).

The shape functions and temporal amplitudes may in general be complex functions. We let $\mathcal{R}\rceil\{\alpha\}$ and $\mathcal{I}\Updownarrow\{\alpha\}$ denote the real and imaginary part of a complex number α.

4.4 Spectral kinematics

Given the wave potential as defined above, the API may evaluate kinematics as described in the following table.

Symbol	API	Note
ϕ	`phi()`	The velocity potential is often applied as Dirichlet conditions in potential flow solvers.
φ	`stream()`	The stream function is orthogonal to the velocity potential. $\frac{\partial \phi}{\partial x} = \frac{\partial \varphi}{\partial z}$ and $\frac{\partial \phi}{\partial z} = -\frac{\partial \varphi}{\partial x}$ The stream function is only relevant for long crested waves. For short crested seas we define $\varphi \equiv 0$ for all $\mathbf{x}$ and t.
$\frac{\partial \phi}{\partial t}$	`phi_t()`	is the time rate of change of ϕ at an earth fix location. It is sometimes applied as Dirichlet conditions in potential flow solvers. This term is an important part of the pressure field. For linear models it is proportional to the dynamic pressure.
$\nabla\phi$	`grad_phi()`	The fluid particle velocity
$\nabla\nabla\phi$	`grad_phi_2nd()`	The second order spatial gradients of ϕ are sometimes applied as boundary conditions in potential flow solvers. They are also needed for calculating particle accelerations.
$\frac{\partial \nabla \phi}{\partial t}$	`acc_euler()`	The Euler acceleration. (Earth fixed location)
$\frac{d \nabla \phi}{dt}$	`acc_particle()`	The fluid particle acceleration
p	`pressure()`	The fluid pressure including all terms in Bernoulli's equation.
ζ	`elev()`	The surface elevation
$\frac{\partial \zeta}{\partial t}$	`elev_t()`	The time rate of change of surface elevation (At fixed horizontal location)
$\nabla\zeta$	`grad_elev()`	Spatial gradients of the surface elevation is often applied in boundary conditions
$\nabla\nabla\zeta$	`grad_elev_2nd()`	The second order spatial gradients of the surface elevation are

4.5 Shape classes

We define a shape class as a specific set of parameterized shape functions $\psi_j(\mathbf{x})$ and $\xi_j(x, y)$ as explained above. A wave field is assumed to be described by a single shape class.

Because $\psi_j(\mathbf{x})$ is harmonic we may in general write

$$\psi_j(\mathbf{x}) = X_j(x)\, Y_j(y)\, Z_j(z)$$

In the current version, the official API implements the following shape classes

- *Shape 1* For long crested waves propagating in infinite water depth.
- *Shape 2* For long crested waves propagating in constant finite water depth.
- *Shape 3* For long crested waves propagating in infinite, constant or varying water depth.
- *Shape 4* For short crested waves propagating in infinite water depth.
- *Shape 5* For short crested waves propagating in constant finite water depth.
- *Shape 6* For a general set of Airy waves. Infinite or constant water depth.

4.5.1 Shape class 1

This shape class describes long crested waves propagating in infinite water depth.

$$\phi(x, z, t) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{c_j(t)\, X_j(x)\Big\} Z_j(z)$$

$$\zeta(x, t) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{h_j(t)\, X_j(x)\Big\}$$

$$X_j(x) = e^{-ik_j x}, \quad Z_j(z) = e^{k_j z}, \quad k_j = j \cdot \Delta k, \quad i = \sqrt{-1}$$

The set of real constants k_j resemble wave numbers. It follows that the kinematics is periodic in space

$$\phi(x + \lambda_{\max}, z, t) = \phi(x, z, t), \qquad \zeta(x + \lambda_{\max}, t) = \zeta(x, t)$$

$$\lambda_{\max} = \frac{2\pi}{\Delta k}, \qquad \lambda_{\min} = \frac{\lambda_{\max}}{n}$$

where $\lambda_{\min}$ and $\lambda_{\max}$ are the shortest and longest wave lengths resolved respectively.

The actual set of shape functions is uniquely defined by the two input parameters Δk and n.

Note: The fields related to $j = 0$ are uniform in space (DC bias). Non-zero values of $h_0(t)$ violates mass conservation. The amplitude $c_0(t)$ adds a uniform time varying ambient pressure field not influencing the flow field. Consequently, these components will by default be suppressed in the kinematic calculations. However, there is an option in the API for including all DC values provided by the *wave generator*.

The fields related to $j = n$ are expected to correspond to the Nyquist frequency of the physical resolution applied in the *wave generator*. Hence, typical $n = \lfloor n_{fft}/2 \rfloor$ where n_{fft} is the physical spatial resolution applied in the *wave generator*.

Kinematics

Given the definitions above we obtain the following explicit kinematics:

$$\phi(\bar{x},\bar{y},\bar{z},\bar{t}) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{c_j(t)\,X_j(x)\Big\}Z_j(z)$$

$$\varphi(\bar{x},\bar{y},\bar{z},\bar{t}) = \sum_{j=0}^{n} \mathcal{I}\Updownarrow\Big\{c_j(t)\,X_j(x)\Big\}Z_j(z)$$

$$\frac{\partial\phi}{\partial\bar{t}}(\bar{x},\bar{y},\bar{z},\bar{t}) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{\frac{dc_j(t)}{dt}\,X_j(x)\Big\}Z_j(z)$$

$$\zeta(\bar{x},\bar{y},\bar{t}) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{h_j(t)\,X_j(x)\Big\}$$

$$\frac{\partial\zeta}{\partial\bar{t}}(\bar{x},\bar{y},\bar{t}) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{\frac{dh_j(t)}{dt}\,X_j(x)\Big\}$$

$$\frac{\partial\zeta}{\partial\bar{x}}(\bar{x},\bar{y},\bar{t}) = \zeta_x\cos\beta, \qquad \frac{\partial\zeta}{\partial\bar{y}}(\bar{x},\bar{y},\bar{t}) = \zeta_x\sin\beta$$

$$\zeta_x = \sum_{j=0}^{n} k_j\mathcal{I}\Updownarrow\Big\{h_j(t)\,X_j(x)\Big\}$$

$$\bar{\nabla}\phi(\bar{x},\bar{y},\bar{z},\bar{t}) = [\phi_x\cos\beta, \phi_x\sin\beta, \phi_z]^T$$

$$\phi_x = \sum_{j=0}^{n} k_j\mathcal{I}\Updownarrow\Big\{c_j(t)\,X_j(x)\Big\}\,Z_j(z)$$

$$\phi_z = \sum_{j=0}^{n} k_j\mathcal{R}\rceil\Big\{c_j(t)\,X_j(x)\Big\}\,Z_j(z)$$

$$\frac{\partial\bar{\nabla}\phi}{\partial\bar{t}}(\bar{x},\bar{y},\bar{z},\bar{t}) = [\phi_{xt}\cos\beta, \phi_{xt}\sin\beta, \phi_{zt}]^T$$

$$\phi_{xt} = \sum_{j=0}^{n} k_j\mathcal{I}\Updownarrow\Big\{\frac{dc_j(t)}{dt}\,X_j(x)\Big\}Z_j(z)$$

$$\phi_{zt} = \sum_{j=0}^{n} k_j\mathcal{R}\rceil\Big\{\frac{dc_j(t)}{dt}\,X_j(x)\Big\}Z_j(z)$$

$$\frac{d\bar{\nabla}\phi}{d\bar{t}}(\bar{x},\bar{y},\bar{z},\bar{t}) = \frac{\partial\bar{\nabla}\phi}{\partial\bar{t}} + \bar{\nabla}\phi\cdot\bar{\nabla}\bar{\nabla}\phi$$

$$\bar{\nabla}\bar{\nabla}\phi(\bar{x},\bar{y},\bar{z},\bar{t}) = \begin{bmatrix} \phi_{\bar{x},\bar{x}} & \phi_{\bar{x},\bar{y}} & \phi_{\bar{x},\bar{z}} \\ \phi_{\bar{x},\bar{y}} & \phi_{\bar{y},\bar{y}} & \phi_{\bar{y},\bar{z}} \\ \phi_{\bar{x},\bar{z}} & \phi_{\bar{y},\bar{z}} & \phi_{\bar{z},\bar{z}} \end{bmatrix}$$

$$\phi_{\bar{x},\bar{x}} = \phi_{xx}\cos^2\beta, \qquad \phi_{\bar{x},\bar{y}} = \phi_{xx}\sin\beta\cos\beta, \qquad \phi_{\bar{x},\bar{z}} = \phi_{xz}\cos\beta$$

$$\phi_{\bar{y},\bar{y}} = \phi_{xx}\sin^2\beta, \qquad \phi_{\bar{y},\bar{z}} = \phi_{xz}\sin\beta, \qquad \phi_{\bar{z},\bar{z}} = \phi_{zz} = -\phi_{xx}$$

$$\phi_{xx} = -\sum_{j=0}^{n} k_j^2\mathcal{R}\rceil\Big\{c_j(t)\,X_j(x)\Big\}Z_j(z)$$

$$\phi_{zz} = \sum_{j=0}^{n} k_j^2 \mathcal{R}\rceil\Big\{c_j(t)\, X_j(x)\Big\} Z_j(z) = -\phi_{xx}$$

$$\phi_{xz} = \sum_{j=0}^{n} k_j^2 \mathcal{I}\Updownarrow\Big\{c_j(t)\, X_j(x)\Big\} Z_j(z)$$

$$\frac{\partial^2 \zeta}{\partial \bar{x}^2}(\bar{x},\bar{y},\bar{t}) = \zeta_{xx}\cos^2\beta \qquad \frac{\partial^2 \zeta}{\partial \bar{y}^2}(\bar{x},\bar{y},\bar{t}) = \zeta_{xx}\sin^2\beta$$

$$\frac{\partial^2 \zeta}{\partial \bar{x}\partial \bar{y}}(\bar{x},\bar{y},\bar{t}) = \zeta_{xx}\sin\beta\cos\beta$$

$$\zeta_{xx} = -\sum_{j=0}^{n} k_j^2 \mathcal{R}\rceil\Big\{h_j(t)\, X_j(x)\Big\}$$

$$p = -\rho\frac{\partial \phi}{\partial \bar{t}} - \frac{1}{2}\rho\bar{\nabla}\phi\cdot\bar{\nabla}\phi - \rho g\bar{z}$$

where $\bar{\nabla}$ denotes gradients with respect to $\bar{x}$, $\bar{y}$ and $\bar{z}$. The particle acceleration is labeled $\frac{d\bar{\nabla}\phi}{d\bar{t}}$.

The stream function φ is related to the velocity potential ϕ. Hence $\partial\phi/\partial x = \partial\varphi/\partial z$ and $\partial\phi/\partial z = -\partial\varphi/\partial x$.

Implementation notes

Evaluation of costly transcendental functions ($\cos$, $\sin$, $\exp$, $\ldots$) are almost eliminated by exploiting the following recursive relations

$$X_j(x) = X_1(x)\, X_{j-1}(x), \qquad Z_j(z) = Z_1(z)\, Z_{j-1}(z), \qquad j > 1$$

In case the *wave generator* applies a perturbation theory of order q we apply the following Taylor expansion above the calm free surface.

$$Z_j(z) = 1 + \sum_{p=1}^{q-1} \frac{(k_j z)^p}{p!}, \qquad z > 0$$

4.5.2 Shape class 2

This shape class describes long crested waves propagating in constant water depth d.

$$\phi(x,z,t) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{c_j(t)\, X_j(x)\Big\} Z_j(z)$$

$$\zeta(x,t) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{h_j(t)\, X_j(x)\Big\}$$

$$X_j(x) = e^{-ik_j x}, \quad Z_j(z) = \frac{\cosh k_j(z+d)}{\cosh k_j d}, \quad k_j = j\cdot\Delta k, \quad i = \sqrt{-1}$$

The set of real constants k_j resemble wave numbers. It follows that the kinematics is periodic in space

$$\phi(x+\lambda_{\max},z,t) = \phi(x,z,t), \qquad \zeta(x+\lambda_{\max},t) = \zeta(x,t)$$

$$\lambda_{\max} = \frac{2\pi}{\Delta k}, \qquad \lambda_{\min} = \frac{\lambda_{\max}}{n}$$

where $\lambda_{\min}$ and $\lambda_{\max}$ are the shortest and longest wave lengths resolved respectively.

The actual set of shape functions is uniquely defined by the three input parameters Δk and n and d.

Note: The fields related to $j = 0$ are uniform in space (DC bias). Non-zero values of $h_0(t)$ violates mass conservation. The amplitude $c_0(t)$ adds a uniform time varying ambient pressure field not influencing the flow field. Consequently, these components will by default be suppressed in the kinematic calculations. However, there is an option in the API for including all DC values provided by the *wave generator*.

The fields related to $j = n$ are expected to correspond to the Nyquist frequency of the physical resolution applied in the *wave generator*. Hence, typical $n = \lfloor n_{fft}/2 \rfloor$ where n_{fft} is the physical spatial resolution applied in the *wave generator*.

Kinematics

Given the definitions above we obtain the following explicit kinematics:

$$\phi(\bar{x},\bar{y},\bar{z},\bar{t}) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{c_j(t)\,X_j(x)\Big\}Z_j(z)$$

$$\varphi(\bar{x},\bar{y},\bar{z},\bar{t}) = \sum_{j=0}^{n} \mathcal{I}\Updownarrow\Big\{c_j(t)\,X_j(x)\Big\}\hat{Z}_j(z)$$

$$\frac{\partial\phi}{\partial\bar{t}}(\bar{x},\bar{y},\bar{z},\bar{t}) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{\frac{dc_j(t)}{dt}\,X_j(x)\Big\}Z_j(z)$$

$$\zeta(\bar{x},\bar{y},\bar{t}) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{h_j(t)\,X_j(x)\Big\}$$

$$\frac{\partial\zeta}{\partial\bar{t}}(\bar{x},\bar{y},\bar{t}) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{\frac{dh_j(t)}{dt}\,X_j(x)\Big\}$$

$$\frac{\partial\zeta}{\partial\bar{x}}(\bar{x},\bar{y},\bar{t}) = \zeta_x\cos\beta, \qquad \frac{\partial\zeta}{\partial\bar{y}}(\bar{x},\bar{y},\bar{t}) = \zeta_x\sin\beta$$

$$\zeta_x = \sum_{j=0}^{n} k_j\mathcal{I}\Updownarrow\Big\{h_j(t)\,X_j(x)\Big\}$$

$$\bar{\nabla}\phi(\bar{x},\bar{y},\bar{z},\bar{t}) = [\phi_x\cos\beta, \phi_x\sin\beta, \phi_z]^T$$

$$\phi_x = \sum_{j=0}^{n} k_j\mathcal{I}\Updownarrow\Big\{c_j(t)\,X_j(x)\Big\}\,Z_j(z)$$

$$\phi_z = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{c_j(t)\,X_j(x)\Big\}\,\frac{dZ_j(z)}{dz}$$

$$\frac{\partial\bar{\nabla}\phi}{\partial\bar{t}}(\bar{x},\bar{y},\bar{z},\bar{t}) = [\phi_{xt}\cos\beta, \phi_{xt}\sin\beta, \phi_{zt}]^T$$

$$\phi_{xt} = \sum_{j=0}^{n} k_j\mathcal{I}\Updownarrow\Big\{\frac{dc_j(t)}{dt}\,X_j(x)\Big\}Z_j(z)$$

$$\phi_{zt} = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{\frac{dc_j(t)}{dt}\,X_j(x)\Big\}\frac{dZ_j(z)}{dz}$$

$$\frac{d\bar{\nabla}\phi}{d\bar{t}}(\bar{x},\bar{y},\bar{z},\bar{t}) = \frac{\partial\bar{\nabla}\phi}{\partial\bar{t}} + \bar{\nabla}\phi\cdot\bar{\nabla}\bar{\nabla}\phi$$

$$\bar{\nabla}\bar{\nabla}\phi(\bar{x},\bar{y},\bar{z},\bar{t}) = \begin{bmatrix}\phi_{\bar{x},\bar{x}} & \phi_{\bar{x},\bar{y}} & \phi_{\bar{x},\bar{z}}\\ \phi_{\bar{x},\bar{y}} & \phi_{\bar{y},\bar{y}} & \phi_{\bar{y},\bar{z}}\\ \phi_{\bar{x},\bar{z}} & \phi_{\bar{y},\bar{z}} & \phi_{\bar{z},\bar{z}}\end{bmatrix}$$

$$\phi_{\bar{x},\bar{x}} = \phi_{xx}\cos^2\beta, \qquad \phi_{\bar{x},\bar{y}} = \phi_{xx}\sin\beta\cos\beta, \qquad \phi_{\bar{x},\bar{z}} = \phi_{xz}\cos\beta$$

$$\phi_{\bar{y},\bar{y}} = \phi_{xx}\sin^2\beta, \qquad \phi_{\bar{y},\bar{z}} = \phi_{xz}\sin\beta, \qquad \phi_{\bar{z},\bar{z}} = \phi_{zz} = -\phi_{xx}$$

$$\phi_{xx} = -\sum_{j=0}^{n} k_j^2\mathcal{R}\rceil\Big\{c_j(t)\,X_j(x)\Big\}Z_j(z)$$

$$\phi_{zz} = \sum_{j=0}^{n} k_j^2\mathcal{R}\rceil\Big\{c_j(t)\,X_j(x)\Big\}Z_j(z) = -\phi_{xx}$$

$$\phi_{xz} = \sum_{j=0}^{n} k_j\mathcal{I}\Updownarrow\Big\{c_j(t)\,X_j(x)\Big\}\frac{dZ_j(z)}{dz}$$

$$\frac{\partial^2\zeta}{\partial\bar{x}^2}(\bar{x},\bar{y},\bar{t}) = \zeta_{xx}\cos^2\beta \qquad \frac{\partial^2\zeta}{\partial\bar{y}^2}(\bar{x},\bar{y},\bar{t}) = \zeta_{xx}\sin^2\beta$$

$$\frac{\partial^2\zeta}{\partial\bar{x}\partial\bar{y}}(\bar{x},\bar{y},\bar{t}) = \zeta_{xx}\sin\beta\cos\beta$$

$$\zeta_{xx} = -\sum_{j=0}^{n} k_j^2\mathcal{R}\rceil\Big\{h_j(t)\,X_j(x)\Big\}$$

$$p = -\rho\frac{\partial\phi}{\partial\bar{t}} - \frac{1}{2}\rho\bar{\nabla}\phi\cdot\bar{\nabla}\phi - \rho g\bar{z}$$

where $\bar{\nabla}$ denotes gradients with respect to $\bar{x}$, $\bar{y}$ and $\bar{z}$. The particle acceleration is labeled $\frac{d\bar{\nabla}\phi}{d\bar{t}}$.

The stream function φ is related to the velocity potential ϕ. Hence $\partial\phi/\partial x = \partial\varphi/\partial z$ and $\partial\phi/\partial z = -\partial\varphi/\partial x$. Note that for the stream function evaluation we apply the function

$$\hat{Z}_j(z) = \frac{\sinh k_j(z+d)}{\cosh k_j d}$$

Implementation notes

Evaluation of costly transcendental functions ($\cos$, $\sin$, $\exp$, $\ldots$) are almost eliminated by exploiting the following recursive relations

$$X_j(x) = X_1(x)\,X_{j-1}(x), \qquad Z_j(z) = U_jS_j + V_jT_j, \qquad \frac{dZ_j(z)}{dz} = k_j(U_jS_j - V_jT_j)$$

$$U_j = \frac{1+R_j}{2}, \qquad V_j = 1-U_j, \qquad R_j \equiv \tanh k_j d = \frac{R_1+R_{j-1}}{1+R_1\,R_{j-1}}$$

$$S_j \equiv e^{k_j z} = S_1S_{j-1}, \qquad T_j \equiv e^{-k_j z} = T_1T_{j-1} = 1/S_j, \qquad \hat{Z}_j(z) = U_jS_j - V_jT_j$$

When $1-R_j < 100\epsilon_m$, where ϵ_m is the machine precision, we apply the deep water approximations $Z_j = S_j$ and $\frac{dZ_j(z)}{dz} = k_jS_j$. Consequently, only the first $\hat{j}$ spectral components should be treated as shallow water components

$$\hat{j} = \mathrm{int}(\frac{\tanh^{-1}(1-100\epsilon_m)}{\Delta k\cdot d})$$

This number can be precomputed prior to any kinematical calculations to avoid testing and numerical issues.

In case the *wave generator* applies a perturbation theory of order q we apply the following Taylor expansion above the calm free surface.

$$S_j(z) = 1 + \sum_{p=1}^{q-1} \frac{(k_j z)^p}{p!}, \qquad z > 0$$

4.5.3 Shape class 3

This shape class describes long crested waves propagating in infinite, constant or varying water depth. Consequently, it may be interpreted as a generalization of *Shape 1* and *Shape 2*. Because the storage and CPU requirements are similar to those specialized classes we may consider to only apply this class for all long crested waves independent of the actual sea floor formulation.

$$\phi(x,z,t) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{c_j(t)\, X_j(x)\Big\} Z_j(z) + \sum_{j=0}^{\hat{n}} \mathcal{R}\rceil\Big\{\hat{c}_j(t)\, X_j(x)\Big\} \hat{Z}_j(z)$$

$$\zeta(x,t) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{h_j(t)\, X_j(x)\Big\}$$

$$X_j(x) = e^{-ik_j x}, \quad Z_j(z) = e^{k_j z}, \quad \hat{Z}_j(z) = e^{-k_j z}, \quad k_j = j \cdot \Delta k, \quad i = \sqrt{-1}$$

The set of real constants k_j resemble wave numbers. It follows that the kinematics is periodic in space

$$\phi(x + \lambda_{\max}, z, t) = \phi(x,z,t), \qquad \zeta(x + \lambda_{\max}, t) = \zeta(x,t)$$

$$\lambda_{\max} = \frac{2\pi}{\Delta k}, \qquad \lambda_{\min} = \frac{\lambda_{\max}}{n}$$

where $\lambda_{\min}$ and $\lambda_{\max}$ are the shortest and longest wave lengths resolved respectively.

The actual set of shape functions is uniquely defined by the three input parameters Δk, n and $\hat{n}$.

Note: The fields related to $j = 0$ are uniform in space (DC bias). Non-zero values of $h_0(t)$ violates mass conservation. The amplitude $c_0(t)$ and $\hat{c}_0(t)$ adds a uniform time varying ambient pressure field not influencing the flow field. Consequently, these components will by default be suppressed in the kinematic calculations. However, there is an option in the API for including all DC values provided by the *wave generator*.

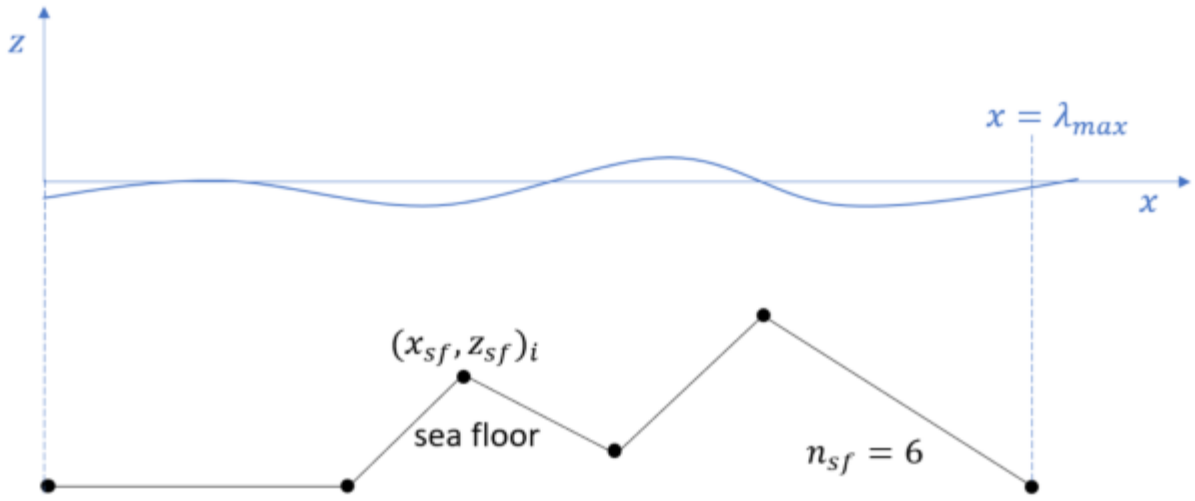
The fields related to $j = n$ are expected to correspond to the Nyquist frequency of the physical resolution applied in the *wave generator*. Hence, typical $n = \lfloor n_{fft}/2 \rfloor$ where n_{fft} is the physical spatial resolution applied in the *wave generator*.

The sea floor

The sea floor $z_{sf}(x)$ is assumed single valued and is periodic with respect to the x-location

$$z_{sf}(x + \lambda_{\max}) = z_{sf}(x)$$

Hence, the seafloor steepness is finite but may be arbitrary large. In the current version of the API the sea floor is continous and piecewise linear. As illustrated in the figure below, the sea floor is defined by linear interpolation between n_{sf} offset points $(x_{sf}, z_{sf})_i$ for $i \in \{1, 2, \ldots, n_{sf}\}$. The sequence $x_{sf}(i)$ is monotonic increasing and covering the range $x_{sf}(i) \in [0, \lambda_{\max}]$



Infinite water depth

The infinite depth formulation follows directly by specifying $n_{sf} = 0$ and $\hat{n} = -1$. Consequently the formulation is equivalent to *Shape 1* and there are no spectral amplitudes $\hat{c}_j(t)$.

Constant water depth

Constant water depth d is defined by assigning $n_{sf} = 1$. Hence $d \equiv -z_{sf}(1)$. The classical shallow water equations (ref *Shape 2*) are reformulated using exponential terms

$$C_j(t)\frac{\cosh k_j(z+d)}{\cosh k_j d} \equiv c_j(t)Z_j(z) + \hat{c}_j(t)\hat{Z}_j(z)$$

$$c_j(t) = \gamma_j C_j(t), \qquad \hat{c}_j(t) = \hat{\gamma}_j c_j(t), \qquad \gamma_j = \frac{1+\tanh k_j d}{2}, \qquad \hat{\gamma}_j = e^{-2k_j d}$$

Consequently, we only store $c_j(t)$ in the *SWD-file*. At the beginning of each new time step we construct $\hat{c}_j(t)$ using $c_j(t)$ and the coefficient $\hat{\gamma}_j$. Because this relation is valid for all time instances the same relations are valid for the temporal derivatives too

$$\frac{dc_j(t)}{dt} = \gamma_j \frac{dC_j(t)}{dt}, \qquad \frac{d\hat{c}_j(t)}{dt} = \hat{\gamma}_j \frac{dc_j(t)}{dt}$$

This formulation is potential faster and more numerical stable than direct evaluation of the hyperbolic functions.

Hint: In practice $\hat{n} < n$ because the corresponding neglected high-frequency components do not contribute significantly to any boundary conditions. For the free surface conditions this follows from the property

$$|\hat{c}_j(t)\hat{Z}_j(z)| = |\hat{\gamma}_j c_j(t) e^{-k_j z}| = e^{-2k_j(d+z)}|c_j(t)Z_j(z)| \ll |c_j(t)Z_j(z)|$$

$$j > \hat{n}, \qquad z \in [-\zeta_{\max}, \zeta_{\max}], \qquad d > \zeta_{\max}$$

where $\zeta_{\max}$ is the maximum wave elevation. Near the sea floor the exponential coefficient $e^{-2k_j(d+z)} \to 1^-$. Consequently, the contribution to the boundary conditions at the sea floor vanish too, because $|c_j(t)Z_j(z)|$ vanish for large j.

Due to finite precision arithmetic the following upper limit is recommended

$$\hat{n} \leq \frac{\tanh^{-1}(1 - 100\epsilon_m)}{\Delta k \cdot d}$$

where ϵ_m is the machine precision ($1 + \epsilon_m = 1$).

Varying water depth

Varying water depth (bathymetry) is assumed if $n_{sf} > 1$. In this case also the coefficients $\hat{c}_j(t)$ needs to be stored in the SWD file.

Hint: For varying water depth, the additional $\hat{n}$ complex valued coefficients $\hat{c}_j(t)$ at time t are expected to be determined in the *wave generator* by adding zero-flux boundary conditions at $2\hat{n}$ distributed collocation points on the sea floor.

Due to finite precision arithmetic the following upper limit is recommended

$$\hat{n} \leq \frac{\tanh^{-1}(1 - 100\epsilon_m)}{-\Delta k \cdot \min\{z_{zf}\}}$$

where ϵ_m is the machine precision ($1 + \epsilon_m = 1$).

Kinematics

Given the definitions above we obtain the following explicit kinematics:

$$\phi(\bar{x}, \bar{y}, \bar{z}, \bar{t}) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{c_j(t)\, X_j(x)\Big\} Z_j(z) + \sum_{j=0}^{\hat{n}} \mathcal{R}\rceil\Big\{\hat{c}_j(t)\, X_j(x)\Big\} \hat{Z}_j(z)$$

$$\varphi(\bar{x}, \bar{y}, \bar{z}, \bar{t}) = \sum_{j=0}^{n} \mathcal{I}\Updownarrow\Big\{c_j(t)\, X_j(x)\Big\} Z_j(z) - \sum_{j=0}^{\hat{n}} \mathcal{I}\Updownarrow\Big\{\hat{c}_j(t)\, X_j(x)\Big\} \hat{Z}_j(z)$$

$$\frac{\partial \phi}{\partial \bar{t}}(\bar{x}, \bar{y}, \bar{z}, \bar{t}) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{\frac{dc_j(t)}{dt}\, X_j(x)\Big\} Z_j(z) + \sum_{j=0}^{\hat{n}} \mathcal{R}\rceil\Big\{\frac{d\hat{c}_j(t)}{dt}\, X_j(x)\Big\} \hat{Z}_j(z)$$

$$\zeta(\bar{x}, \bar{y}, \bar{t}) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{h_j(t)\, X_j(x)\Big\}$$

$$\frac{\partial \zeta}{\partial \bar{t}}(\bar{x}, \bar{y}, \bar{t}) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{\frac{dh_j(t)}{dt}\, X_j(x)\Big\}$$

$$\frac{\partial \zeta}{\partial \bar{x}}(\bar{x}, \bar{y}, \bar{t}) = \zeta_x \cos\beta, \qquad \frac{\partial \zeta}{\partial \bar{y}}(\bar{x}, \bar{y}, \bar{t}) = \zeta_x \sin\beta$$

$$\zeta_x = \sum_{j=0}^{n} k_j \mathcal{I}\Updownarrow\Big\{h_j(t)\, X_j(x)\Big\}$$

$$\bar{\nabla}\phi(\bar{x}, \bar{y}, \bar{z}, \bar{t}) = [\phi_x \cos\beta, \phi_x \sin\beta, \phi_z]^T$$

$$\phi_x = \sum_{j=0}^{n} k_j \mathcal{I}\Updownarrow\Big\{c_j(t)\, X_j(x)\Big\}\, Z_j(z) + \sum_{j=0}^{\hat{n}k_j \mathcal{I}\Updownarrow\left\{\hat{c}_j(t)\, X_j(x)\right\} \hat{Z}_j(z)}$$

$$\phi_z = \sum_{j=0}^{n} k_j \mathcal{R}\rceil\Big\{c_j(t)\, X_j(x)\Big\}\, Z_j(z) - \sum_{j=0}^{\hat{n} k_j \mathcal{R}\rceil\left\{\hat{c}_j(t)\, X_j(x)\right\} \hat{Z}_j(z)}$$

$$\frac{\partial \bar{\nabla}\phi}{\partial \bar{t}}(\bar{x}, \bar{y}, \bar{z}, \bar{t}) = [\phi_{xt}\cos\beta, \phi_{xt}\sin\beta, \phi_{zt}]^T$$

$$\phi_{xt} = \sum_{j=0}^{n} k_j \mathcal{I}\Updownarrow\Big\{\frac{dc_j(t)}{dt}\, X_j(x)\Big\} Z_j(z) + \sum_{j=0}^{\hat{n} k_j \mathcal{I}\Updownarrow\left\{\frac{d\hat{c}_j(t)}{dt}\, X_j(x)\right\} \hat{Z}_j(z)}$$

$$\phi_{zt} = \sum_{j=0}^{n} k_j \mathcal{R}\rceil\Big\{\frac{dc_j(t)}{dt}\, X_j(x)\Big\} Z_j(z) - \sum_{j=0}^{\hat{n} k_j \mathcal{R}\rceil\left\{\frac{d\hat{c}_j(t)}{dt}\, X_j(x)\right\} \hat{Z}_j(z)}$$

$$\frac{d\bar{\nabla}\phi}{d\bar{t}}(\bar{x}, \bar{y}, \bar{z}, \bar{t}) = \frac{\partial \bar{\nabla}\phi}{\partial \bar{t}} + \bar{\nabla}\phi \cdot \bar{\nabla}\bar{\nabla}\phi$$

$$\bar{\nabla}\bar{\nabla}\phi(\bar{x}, \bar{y}, \bar{z}, \bar{t}) = \begin{bmatrix} \phi_{\bar{x},\bar{x}} & \phi_{\bar{x},\bar{y}} & \phi_{\bar{x},\bar{z}} \\ \phi_{\bar{x},\bar{y}} & \phi_{\bar{y},\bar{y}} & \phi_{\bar{y},\bar{z}} \\ \phi_{\bar{x},\bar{z}} & \phi_{\bar{y},\bar{z}} & \phi_{\bar{z},\bar{z}} \end{bmatrix}$$

$$\phi_{\bar{x},\bar{x}} = \phi_{xx}\cos^2\beta, \qquad \phi_{\bar{x},\bar{y}} = \phi_{xx}\sin\beta\cos\beta, \qquad \phi_{\bar{x},\bar{z}} = \phi_{xz}\cos\beta$$

$$\phi_{\bar{y},\bar{y}} = \phi_{xx}\sin^2\beta, \qquad \phi_{\bar{y},\bar{z}} = \phi_{xz}\sin\beta, \qquad \phi_{\bar{z},\bar{z}} = \phi_{zz} = -\phi_{xx}$$

$$\phi_{xx} = -\sum_{j=0}^{n} k_j^2 \mathcal{R}\rceil\Big\{c_j(t)\, X_j(x)\Big\} Z_j(z) - \sum_{j=0}^{\hat{n} k_j^2 \mathcal{R}\rceil\left\{\hat{c}_j(t)\, X_j(x)\right\} \hat{Z}_j(z)}$$

$$\phi_{zz} = \sum_{j=0}^{n} k_j^2 \mathcal{R}\rceil\Big\{c_j(t)\, X_j(x)\Big\} Z_j(z) + \sum_{j=0}^{\hat{n} k_j^2 \mathcal{R}\rceil\left\{\hat{c}_j(t)\, X_j(x)\right\} \hat{Z}_j(z) = -\phi_{xx}}$$

$$\phi_{xz} = \sum_{j=0}^{n} k_j^2 \mathcal{I}\Updownarrow\Big\{c_j(t)\, X_j(x)\Big\} Z_j(z) - \sum_{j=0}^{\hat{n} k_j^2 \mathcal{I}\Updownarrow\left\{\hat{c}_j(t)\, X_j(x)\right\} \hat{Z}_j(z)}$$

$$\frac{\partial^2 \zeta}{\partial \bar{x}^2}(\bar{x}, \bar{y}, \bar{t}) = \zeta_{xx}\cos^2\beta \qquad \frac{\partial^2 \zeta}{\partial \bar{y}^2}(\bar{x}, \bar{y}, \bar{t}) = \zeta_{xx}\sin^2\beta$$

$$\frac{\partial^2 \zeta}{\partial \bar{x}\partial \bar{y}}(\bar{x}, \bar{y}, \bar{t}) = \zeta_{xx}\sin\beta\cos\beta$$

$$\zeta_{xx} = -\sum_{j=0}^{n} k_j^2 \mathcal{R}\rceil\Big\{h_j(t)\, X_j(x)\Big\}$$

$$p = -\rho\frac{\partial \phi}{\partial \bar{t}} - \frac{1}{2}\rho\bar{\nabla}\phi \cdot \bar{\nabla}\phi - \rho g\bar{z}$$

where $\bar{\nabla}$ denotes gradients with respect to $\bar{x}$, $\bar{y}$ and $\bar{z}$. The particle acceleration is labeled $\frac{d\bar{\nabla}\phi}{dt}$.

The stream function φ is related to the velocity potential ϕ. Hence $\partial\phi/\partial x = \partial\varphi/\partial z$ and $\partial\phi/\partial z = -\partial\varphi/\partial x$.

Implementation notes

Evaluation of costly transcendental functions ($\cos$, $\sin$, $\exp$, ...) are almost eliminated by exploiting the following recursive relations

$$X_j(x) = X_1(x)\, X_{j-1}(x), \qquad Z_j(z) = Z_1(z)\, Z_{j-1}(z), \qquad \hat{Z}_j(z) = \hat{Z}_1(z)\, \hat{Z}_{j-1}(z)$$

In case the *wave generator* applies a perturbation theory of order q we apply the following Taylor expansions above the calm free surface.

$$Z_j(z) = 1 + \sum_{p=1}^{q-1} \frac{(k_j z)^p}{p!}, \qquad \hat{Z}_j(z) = 1 + \sum_{p=1}^{q-1} \frac{(-k_j z)^p}{p!}, \qquad z > 0$$

4.5.4 Shape class 4

This shape class describes general short crested spectral waves propagating in infinite water depth.

$$\phi(x,y,z,t) = \sum_{j_x=0}^{n_x} \sum_{j_y=-n_y}^{n_y} \mathcal{R}\rceil\Big\{c_{j_y,j_x}(t)\, X_{j_x}(x)\, Y_{j_y}(y)\Big\}\, Z_{j_y,j_x}(z)$$

$$\zeta(x,y,t) = \sum_{j_x=0}^{n_x} \sum_{j_y=-n_y}^{n_y} \mathcal{R}\rceil\Big\{h_{j_y,j_x}(t)\, X_{j_x}(x)\, Y_{j_y}(y)\Big\}$$

$$X_{j_x}(x) = e^{-ik_{j_x}x}, \quad Y_{j_y}(y) = e^{-ik_{j_y}y}, \quad Z_{j_y,j_x}(z) = e^{k_{j_y,j_x}z}$$

$$k_{j_x} = j_x \cdot \Delta k_x, \quad k_{j_y} = j_y \cdot \Delta k_y, \quad k_{j_y,j_x} = \sqrt{k_{j_x}^2 + k_{j_y}^2}, \quad i = \sqrt{-1}$$

The set of real constants k_{j_x} and k_{j_y} resemble wave numbers in the x and y directions respectively. It follows that the kinematics is periodic in space

$$\phi(x + L_x, y + L_y, z, t) = \phi(x,y,z,t), \qquad \zeta(x + L_x, y + L_y, t) = \zeta(x,y,t)$$

$$L_x = \frac{2\pi}{\Delta k_x}, \qquad L_y = \frac{2\pi}{\Delta k_y}, \qquad \lambda_{\min} = \frac{2\pi}{\sqrt{(n_x \Delta k_x)^2 + (n_y \Delta k_y)^2}}$$

where $\lambda_{\min}$ is the shortest wave lengths resolved. The actual set of shape functions is uniquely defined by the five input parameters Δk_x, Δk_y, n_x, n_y and d.

Note: The fields related to $j_x = j_y = 0$ are uniform in space (DC bias). Non-zero values of $h_{0,0}(t)$ violates mass conservation. The amplitude $c_{0,0}(t)$ adds a uniform time varying ambient pressure field not influencing the flow field. Consequently, these components will by default be suppressed in the kinematic calculations. However, there is an option in the API for including all DC values provided by the *wave generator*.

The fields related to $j_x = n_x$ and $j_y = \pm n_y$ are expected to correspond to the Nyquist frequencies of the physical resolution applied in the *wave generator*. Hence, typical $n_x = \lfloor n_{x,fft}/2 \rfloor$ and $n_y = \lfloor n_{y,fft}/2 \rfloor$ where $n_{x,fft}$ and $n_{y,fft}$ are the physical spatial resolutions applied in the *wave generator*, in the x and y directions respectively.

Evaluation of kinematics in short-crested seas is in general computational demanding. Consequently, this API provides several alternative implementations in order to exploit eventual symmetric properties or numerical approximations.

- *Shape 4, impl 1:* A general implementation.
- *Shape 4, impl 2:* Optimized for symmetric resolution $\Delta k_x = \Delta k_y$ and $n_x = n_y$.

Shape class 4, impl=1

This implementation of *Shape 4* evaluates the more general non-symmetric spatial resolution where Δk_x may differ from Δk_y or n_x from n_y.

However, since $Y_{-j_y}(y) = \bar{Y}_{j_y}(y)$, where $\bar{Y}_{j_y}(y)$ denotes the complex conjugate of $Y_{j_y}(y)$, and $Z_{-j_y,j_x}(z) = Z_{j_y,j_x}(z)$ we may always apply the following equivalent wave field formulation.

$$\phi(x,y,z,t) = \sum_{j_x=0}^{n_x}\sum_{j_y=0}^{n_y} \mathcal{R}\rceil\Big\{C_{1,j_y,j_x}(y,t)\,X_{j_x}(x)\Big\}\,Z_{j_y,j_x}(z)$$

$$\zeta(x,y,t) = \sum_{j_x=0}^{n_x}\sum_{j_y=0}^{n_y} \mathcal{R}\rceil\Big\{H_{1,j_y,j_x}(y,t)\,X_{j_x}(x)\Big\}$$

$$C_{1,j_y,j_x}(y,t) = c_{1,j_y,j_x}(t)\,Y_{j_y}(y) + c_{2,j_y,j_x}(t)\,\bar{Y}_{j_y}(y)$$

$$H_{1,j_y,j_x}(y,t) = h_{1,j_y,j_x}(t)\,Y_{j_y}(y) + h_{2,j_y,j_x}(t)\,\bar{Y}_{j_y}(y)$$

$$c_{1,j_y,j_x}(t) = c_{j_y,j_x}(t), \qquad c_{2,j_y,j_x}(t) = \begin{cases} c_{-j_y,j_x}(t), & j_y > 0, \\ 0, & j_y = 0 \end{cases}$$

$$h_{1,j_y,j_x}(t) = h_{j_y,j_x}(t), \qquad h_{2,j_y,j_x}(t) = \begin{cases} h_{-j_y,j_x}(t), & j_y > 0, \\ 0, & j_y = 0 \end{cases}$$

Kinematics

Given the definitions above we obtain the following explicit kinematics:

$$\phi(\bar{x},\bar{y},\bar{z},\bar{t}) = \sum_{j_x=0}^{n_x}\sum_{j_y=0}^{n_y} \mathcal{R}\rceil\Big\{C_{1,j_y,j_x}(y,t)\,X_{j_x}(x)\Big\}Z_{j_y,j_x}(z)$$

$$\varphi(\bar{x},\bar{y},\bar{z},\bar{t}) \equiv 0$$

$$\frac{\partial\phi}{\partial\bar{t}}(\bar{x},\bar{y},\bar{z},\bar{t}) = \sum_{j_x=0}^{n_x}\sum_{j_y=0}^{n_y} \mathcal{R}\rceil\Big\{\frac{dC_{1,j_y,j_x}(y,t)}{dt}\,X_{j_x}(x)\Big\}Z_{j_y,j_x}(z)$$

$$\zeta(\bar{x},\bar{y},\bar{t}) = \sum_{j_x=0}^{n_x}\sum_{j_y=0}^{n_y} \mathcal{R}\rceil\Big\{H_{1,j_y,j_x}(y,t)\,X_{j_x}(x)\Big\}$$

$$\frac{\partial\zeta}{\partial\bar{t}}(\bar{x},\bar{y},\bar{t}) = \sum_{j_x=0}^{n_x}\sum_{j_y=0}^{n_y} \mathcal{R}\rceil\Big\{\frac{dH_{1,j_y,j_x}(y,t)}{dt}\,X_{j_x}(x)\Big\}$$

$$\frac{\partial\zeta}{\partial\bar{x}}(\bar{x},\bar{y},\bar{t}) = \zeta_x\cos\beta - \zeta_y\sin\beta, \qquad \frac{\partial\zeta}{\partial\bar{y}}(\bar{x},\bar{y},\bar{t}) = \zeta_x\sin\beta + \zeta_y\cos\beta$$

$$\zeta_x = \sum_{j_x=0}^{n_x}\sum_{j_y=0}^{n_y} k_{j_x}\mathcal{I}\Updownarrow\Big\{H_{1,j_y,j_x}(y,t)\,X_{j_x}(x)\Big\}$$

$$\zeta_y = -\sum_{j_x=0}^{n_x}\sum_{j_y=0}^{n_y} k_{j_y}\mathcal{I}\Updownarrow\Big\{H_{2,j_y,j_x}(y,t)\,X_{j_x}(x)\Big\}$$

$$\bar{\nabla}\phi(\bar{x},\bar{y},\bar{z},\bar{t}) = [\phi_x\cos\beta - \phi_y\sin\beta, \phi_x\sin\beta + \phi_y\cos\beta, \phi_z]^T$$

$$\phi_x = \sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_x} \mathcal{I}\Updownarrow\Big\{ C_{1,j_y,j_x}(y,t)\, X_{j_x}(x) \Big\}\, Z_{j_y,j_x}(z)$$

$$\phi_y = \sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_y} \mathcal{I}\Updownarrow\Big\{ C_{2,j_y,j_x}(y,t)\, X_{j_x}(x) \Big\}\, Z_{j_y,j_x}(z)$$

$$\phi_z = \sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_y,j_x} \mathcal{R}\rceil\Big\{ C_{1,j_y,j_x}(y,t)\, X_{j_x}(x) \Big\}\, Z_{j_y,j_x}(z)$$

$$\frac{\partial \bar{\nabla}\phi}{\partial \bar{t}}(\bar{x},\bar{y},\bar{z},\bar{t}) = [\phi_{xt}\cos\beta - \phi_{yt}\sin\beta, \phi_{xt}\sin\beta + \phi_{yt}\cos\beta, \phi_z]^T$$

$$\phi_{xt} = \sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_x} \mathcal{I}\Updownarrow\Big\{ \frac{dC_{1,j_y,j_x}(y,t)}{dt}\, X_{j_x}(x) \Big\}\, Z_{j_y,j_x}(z)$$

$$\phi_{yt} = \sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_y} \mathcal{I}\Updownarrow\Big\{ \frac{dC_{2,j_y,j_x}(y,t)}{dt}\, X_{j_x}(x) \Big\}\, Z_{j_y,j_x}(z)$$

$$\phi_{zt} = \sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_y,j_x} \mathcal{R}\rceil\Big\{ \frac{dC_{1,j_y,j_x}(y,t)}{dt}\, X_{j_x}(x) \Big\}\, Z_{j_y,j_x}(z)$$

$$\frac{d\bar{\nabla}\phi}{d\bar{t}}(\bar{x},\bar{y},\bar{z},\bar{t}) = \frac{\partial \bar{\nabla}\phi}{\partial \bar{t}} + \bar{\nabla}\phi \cdot \bar{\nabla}\bar{\nabla}\phi$$

$$\bar{\nabla}\bar{\nabla}\phi(\bar{x},\bar{y},\bar{z},\bar{t}) = \begin{bmatrix} \phi_{\bar{x},\bar{x}} & \phi_{\bar{x},\bar{y}} & \phi_{\bar{x},\bar{z}} \\ \phi_{\bar{x},\bar{y}} & \phi_{\bar{y},\bar{y}} & \phi_{\bar{y},\bar{z}} \\ \phi_{\bar{x},\bar{z}} & \phi_{\bar{y},\bar{z}} & \phi_{\bar{z},\bar{z}} \end{bmatrix}$$

$$\phi_{\bar{x},\bar{x}} = \phi_{xx}\cos^2\beta - \phi_{xy}\sin(2\beta) + \phi_{yy}\sin^2\beta$$

$$\phi_{\bar{x},\bar{y}} = \phi_{xy}(\cos^2\beta - \sin^2\beta) + (\phi_{xx} - \phi_{yy})\sin\beta\cos\beta$$

$$\phi_{\bar{x},\bar{z}} = \phi_{xz}\cos\beta - \phi_{yz}\sin\beta$$

$$\phi_{\bar{y},\bar{y}} = \phi_{yy}\cos^2\beta + \phi_{xy}\sin(2\beta) + \phi_{xx}\sin^2\beta$$

$$\phi_{\bar{y},\bar{z}} = \phi_{yz}\cos\beta + \phi_{xz}\sin\beta$$

$$\phi_{\bar{z},\bar{z}} = \phi_{zz} = -\phi_{xx} - \phi_{yy}$$

$$\phi_{xx} = -\sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_x}^2 \mathcal{R}\rceil\Big\{ C_{1,j_y,j_x}(y,t)\, X_{j_x}(x) \Big\} Z_{j_y,j_x}(z)$$

$$\phi_{xy} = -\sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_x} k_{j_y} \mathcal{R}\rceil\Big\{ C_{2,j_y,j_x}(y,t)\, X_{j_x}(x) \Big\} Z_{j_y,j_x}(z)$$

$$\phi_{xz} = \sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_x} k_{j_y,j_x} \mathcal{I}\Updownarrow\Big\{ C_{1,j_y,j_x}(y,t)\, X_{j_x}(x) \Big\} Z_{j_y,j_x}(z)$$

$$\phi_{yy} = -\sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_y}^2 \mathcal{R}\rceil\Big\{ C_{1,j_y,j_x}(y,t)\, X_{j_x}(x) \Big\} Z_{j_y,j_x}(z)$$

$$\phi_{yz} = \sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_y} k_{j_y,j_x} \mathcal{I}\Updownarrow\Big\{ C_{2,j_y,j_x}(y,t)\, X_{j_x}(x) \Big\} Z_{j_y,j_x}(z)$$

$$\phi_{zz} = \sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_y,j_x}^2 \mathcal{R}\rceil\Big\{C_{1,j_y,j_x}(y,t)\, X_{j_x}(x)\Big\} Z_{j_y,j_x}(z) = -\phi_{xx} - \phi_{yy}$$

$$\frac{\partial^2 \zeta}{\partial \bar{x}^2}(\bar{x},\bar{y},\bar{t}) = \zeta_{xx}\cos^2\beta - \zeta_{xy}\sin(2\beta) + \zeta_{yy}\sin^2\beta$$

$$\frac{\partial^2 \zeta}{\partial \bar{x}\partial \bar{y}}(\bar{x},\bar{y},\bar{t}) = \zeta_{xy}(\cos^2\beta - \sin^2\beta) + (\zeta_{xx} - \zeta_{yy})\sin\beta\cos\beta$$

$$\frac{\partial^2 \zeta}{\partial \bar{y}^2}(\bar{x},\bar{y},\bar{t}) = \zeta_{yy}\cos^2\beta + \zeta_{xy}\sin(2\beta) + \zeta_{xx}\sin^2\beta$$

$$\zeta_{xx} = -\sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_x}^2 \mathcal{R}\rceil\Big\{H_{1,j_y,j_x}(y,t)\, X_{j_x}(x)\Big\}$$

$$\zeta_{xy} = -\sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_x} k_{j_y} \mathcal{R}\rceil\Big\{H_{2,j_y,j_x}(y,t)\, X_{j_x}(x)\Big\}$$

$$\zeta_{yy} = -\sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_y}^2 \mathcal{R}\rceil\Big\{H_{1,j_y,j_x}(y,t)\, X_{j_x}(x)\Big\}$$

$$p = -\rho\frac{\partial \phi}{\partial \bar{t}} - \frac{1}{2}\rho\bar{\nabla}\phi\cdot\bar{\nabla}\phi - \rho g\bar{z}$$

where $\bar{\nabla}$ denotes gradients with respect to $\bar{x}$, $\bar{y}$ and $\bar{z}$. We also apply

$$C_{2,j_y,j_x}(y,t) = c_{1,j_y,j_x}(t)\, Y_{j_y}(y) - c_{2,j_y,j_x}(t)\, \bar{Y}_{j_y}(y)$$

$$H_{2,j_y,j_x}(y,t) = h_{1,j_y,j_x}(t)\, Y_{j_y}(y) - h_{2,j_y,j_x}(t)\, \bar{Y}_{j_y}(y)$$

The particle acceleration is labeled $\frac{d\bar{\nabla}\phi}{d\bar{t}}$.

The stream function φ is not relevant for short crested seas. Hence, we apply the dummy definition $\varphi = 0$ for all locations.

Implementation notes

Evaluation of costly transcendental functions ($\cos$, $\sin$, $\exp$, $\dots$) is significantly reduced by exploiting the following recursive relations

$$X_{j_x}(x) = X_1(x)\, X_{j_x-1}(x), \qquad Y_{j_y}(y) = Y_1(y)\, Y_{j_y-1}(y)$$

It should be noted that contrary to long crested seas, there are no trivial recursive relations for the z-dependent term $Z_{j_y,j_x}(z)$. This makes calculations of surface elevations significantly faster than calculations of other kinematics for short crested seas.

In case the *wave generator* applies a perturbation theory of order q we apply the following Taylor expansion above the calm free surface.

$$Z_{j_y,j_x}(z) = 1 + \sum_{p=1}^{q-1} \frac{(k_{j_y,j_x} z)^p}{p!}, \qquad z > 0$$

Shape class 4, impl=2

This implementation of *Shape 4* is designed for exploiting symmetric spatial resolutions.

$$\Delta k_x = \Delta k_y, \qquad n_x = n_y$$

For short, we will apply the notation $\Delta k = \Delta k_x = \Delta k_y$ and $n = n_x = n_y$. It follows that

$$Y_{-j_y}(y) = \bar{Y}_{j_y}(y), \qquad Z_{-j_y,j_x}(z) = Z_{j_y,j_x}(z), \qquad Z_{j_x,j_y}(z) = Z_{j_y,j_x}(z)$$

where $\bar{Y}_{j_y}(y)$ denotes the complex conjugate of $Y_{j_y}(y)$. Consequently, we apply the following equivalent wave field formulation.

$$\phi(x,y,z,t) = \sum_{j_x=0}^{n} \sum_{j_y=0}^{j_x} \mathcal{R}\rceil\Big\{C_{j_y,j_x}(x,y,t)\Big\} Z_{j_y,j_x}(z)$$

$$\zeta(x,y,t) = \sum_{j_x=0}^{n} \sum_{j_y=0}^{j_x} \mathcal{R}\rceil\Big\{H_{j_y,j_x}(x,y,t)\Big\}$$

$$\begin{aligned}
C_{j_y,j_x}(x,y,t) =& \big(c_{1,j_y,j_x}(t)\, Y_{j_y}(y) + c_{2,j_y,j_x}(t)\, \bar{Y}_{j_y}(y)\big) X_{j_x}(x) + \\
& \big(c_{3,j_y,j_x}(t)\, Y_{j_x}(y) + c_{4,j_y,j_x}(t)\, \bar{Y}_{j_x}(y)\big) X_{j_y}(x) \\
H_{j_y,j_x}(x,y,t) =& \big(h_{1,j_y,j_x}(t)\, Y_{j_y}(y) + h_{2,j_y,j_x}(t)\, \bar{Y}_{j_y}(y)\big) X_{j_x}(x) + \\
& \big(h_{3,j_y,j_x}(t)\, Y_{j_x}(y) + h_{4,j_y,j_x}(t)\, \bar{Y}_{j_x}(y)\big) X_{j_y}(x)
\end{aligned}$$

$$c_{1,j_y,j_x}(t) = c_{j_y,j_x}(t), \qquad c_{2,j_y,j_x}(t) = \begin{cases} c_{-j_y,j_x}(t), & j_y > 0, \\ 0, & j_y = 0 \end{cases}$$

$$c_{3,j_y,j_x}(t) = \begin{cases} c_{j_x,j_y}(t), & j_y < j_x, \\ 0, & j_y = j_x \end{cases}, \qquad c_{4,j_y,j_x}(t) = \begin{cases} c_{-j_x,j_y}(t), & j_y < j_x, \\ 0, & j_y = j_x \end{cases}$$

$$h_{1,j_y,j_x}(t) = h_{j_y,j_x}(t), \qquad h_{2,j_y,j_x}(t) = \begin{cases} h_{-j_y,j_x}(t), & j_y > 0, \\ 0, & j_y = 0 \end{cases}$$

$$h_{3,j_y,j_x}(t) = \begin{cases} h_{j_x,j_y}(t), & j_y < j_x, \\ 0, & j_y = j_x \end{cases}, \qquad h_{4,j_y,j_x}(t) = \begin{cases} h_{-j_x,j_y}(t), & j_y < j_x, \\ 0, & j_y = j_x \end{cases}$$

$$k_{j_x} = j_x \cdot \Delta k, \quad k_{j_y} = j_y \cdot \Delta k, \quad k_{j_y,j_x} = \sqrt{j_x^2 + j_y^2}\, \Delta k, \quad i = \sqrt{-1}$$

Kinematics

Given the definitions above we obtain the following explicit kinematics:

$$\phi(\bar{x},\bar{y},\bar{z},\bar{t}) = \sum_{j_x=0}^{n} \sum_{j_y=0}^{j_x} \mathcal{R}\rceil\Big\{C_{j_y,j_x}(x,y,t)\Big\} Z_{j_y,j_x}(z)$$

$$\varphi(\bar{x},\bar{y},\bar{z},\bar{t}) \equiv 0$$

$$\frac{\partial \phi}{\partial \bar{t}}(\bar{x},\bar{y},\bar{z},\bar{t}) = \sum_{j_x=0}^{n} \sum_{j_y=0}^{j_x} \mathcal{R}\rceil\Big\{\frac{dC_{j_y,j_x}(x,y,t)}{dt}\Big\} Z_{j_y,j_x}(z)$$

$$\zeta(\bar{x},\bar{y},\bar{t}) = \sum_{j_x=0}^{n} \sum_{j_y=0}^{j_x} \mathcal{R}\rceil\Big\{H_{j_y,j_x}(x,y,t)\Big\}$$

$$\frac{\partial \zeta}{\partial \bar{t}}(\bar{x},\bar{y},\bar{t}) = \sum_{j_x=0}^{n}\sum_{j_y=0}^{j_x} \mathcal{R}\rceil\Big\{\frac{dH_{j_y,j_x}(x,y,t)}{dt}\Big\}$$

$$\frac{\partial \zeta}{\partial \bar{x}}(\bar{x},\bar{y},\bar{t}) = \zeta_x\cos\beta - \zeta_y\sin\beta, \qquad \frac{\partial \zeta}{\partial \bar{y}}(\bar{x},\bar{y},\bar{t}) = \zeta_x\sin\beta + \zeta_y\cos\beta$$

$$\zeta_x = \sum_{j_x=0}^{n}\sum_{j_y=0}^{j_x} \mathcal{I}\Updownarrow\Big\{H^{1,0}_{j_y,j_x}(x,y,t)\Big\}$$

$$\zeta_y = \sum_{j_x=0}^{n}\sum_{j_y=0}^{j_x} \mathcal{I}\Updownarrow\Big\{H^{0,1}_{j_y,j_x}(x,y,t)\Big\}$$

$$\bar{\nabla}\phi(\bar{x},\bar{y},\bar{z},\bar{t}) = [\phi_x\cos\beta - \phi_y\sin\beta, \phi_x\sin\beta + \phi_y\cos\beta, \phi_z]^T$$

$$\phi_x = \sum_{j_x=0}^{n}\sum_{j_y=0}^{j_x} \mathcal{I}\Updownarrow\Big\{C^{1,0}_{j_y,j_x}(x,y,t)\Big\} Z_{j_y,j_x}(z)$$

$$\phi_y = \sum_{j_x=0}^{n}\sum_{j_y=0}^{j_x} \mathcal{I}\Updownarrow\Big\{C^{0,1}_{j_y,j_x}(x,y,t)\Big\} Z_{j_y,j_x}(z)$$

$$\phi_z = \sum_{j_x=0}^{n}\sum_{j_y=0}^{j_x} k_{j_y,j_x}\mathcal{R}\rceil\Big\{C_{j_y,j_x}(x,y,t)\Big\} Z_{j_y,j_x}(z)$$

$$\frac{\partial \bar{\nabla}\phi}{\partial \bar{t}}(\bar{x},\bar{y},\bar{z},\bar{t}) = [\phi_{xt}\cos\beta - \phi_{yt}\sin\beta, \phi_{xt}\sin\beta + \phi_{yt}\cos\beta, \phi_z]^T$$

$$\phi_{xt} = \sum_{j_x=0}^{n}\sum_{j_y=0}^{j_x} \mathcal{I}\Updownarrow\Big\{\frac{dC^{1,0}_{j_y,j_x}(x,y,t)}{dt}\Big\} Z_{j_y,j_x}(z)$$

$$\phi_{yt} = \sum_{j_x=0}^{n}\sum_{j_y=0}^{j_x} \mathcal{I}\Updownarrow\Big\{\frac{dC^{0,1}_{j_y,j_x}(x,y,t)}{dt}\Big\} Z_{j_y,j_x}(z)$$

$$\phi_{zt} = \sum_{j_x=0}^{n}\sum_{j_y=0}^{j_x} k_{j_y,j_x}\mathcal{R}\rceil\Big\{\frac{dC_{j_y,j_x}(x,y,t)}{dt}\Big\} Z_{j_y,j_x}(z)$$

$$\frac{d\bar{\nabla}\phi}{d\bar{t}}(\bar{x},\bar{y},\bar{z},\bar{t}) = \frac{\partial \bar{\nabla}\phi}{\partial \bar{t}} + \bar{\nabla}\phi\cdot\bar{\nabla}\bar{\nabla}\phi$$

$$\bar{\nabla}\bar{\nabla}\phi(\bar{x},\bar{y},\bar{z},\bar{t}) = \begin{bmatrix} \phi_{\bar{x},\bar{x}} & \phi_{\bar{x},\bar{y}} & \phi_{\bar{x},\bar{z}} \\ \phi_{\bar{x},\bar{y}} & \phi_{\bar{y},\bar{y}} & \phi_{\bar{y},\bar{z}} \\ \phi_{\bar{x},\bar{z}} & \phi_{\bar{y},\bar{z}} & \phi_{\bar{z},\bar{z}} \end{bmatrix}$$

$$\phi_{\bar{x},\bar{x}} = \phi_{xx}\cos^2\beta - \phi_{xy}\sin(2\beta) + \phi_{yy}\sin^2\beta$$

$$\phi_{\bar{x},\bar{y}} = \phi_{xy}(\cos^2\beta - \sin^2\beta) + (\phi_{xx} - \phi_{yy})\sin\beta\cos\beta$$

$$\phi_{\bar{x},\bar{z}} = \phi_{xz}\cos\beta - \phi_{yz}\sin\beta$$

$$\phi_{\bar{y},\bar{y}} = \phi_{yy}\cos^2\beta + \phi_{xy}\sin(2\beta) + \phi_{xx}\sin^2\beta$$

$$\phi_{\bar{y},\bar{z}} = \phi_{yz}\cos\beta + \phi_{xz}\sin\beta$$

$$\phi_{\bar{z},\bar{z}} = \phi_{zz} = -\phi_{xx} - \phi_{yy}$$

$$\phi_{xx} = -\sum_{j_x=0}^{n}\sum_{j_y=0}^{j_x} \mathcal{R}\rceil\Big\{C^{2,0}_{j_y,j_x}(x,y,t)\Big\} Z_{j_y,j_x}(z)$$

$$\phi_{xy} = -\sum_{j_x=0}^{n}\sum_{j_y=0}^{j_x}\mathcal{R}\rceil\Big\{C^{1,1}_{j_y,j_x}(x,y,t)\Big\}Z_{j_y,j_x}(z)$$

$$\phi_{xz} = \sum_{j_x=0}^{n}\sum_{j_y=0}^{j_x}k_{j_y,j_x}\mathcal{I}\Updownarrow\Big\{C^{1,0}_{j_y,j_x}(x,y,t)\Big\}Z_{j_y,j_x}(z)$$

$$\phi_{yy} = -\sum_{j_x=0}^{n}\sum_{j_y=0}^{j_x}\mathcal{R}\rceil\Big\{C^{0,2}_{j_y,j_x}(x,y,t)\Big\}Z_{j_y,j_x}(z)$$

$$\phi_{yz} = \sum_{j_x=0}^{n}\sum_{j_y=0}^{j_x}k_{j_y,j_x}\mathcal{I}\Updownarrow\Big\{C^{0,1}_{j_y,j_x}(x,y,t)\Big\}Z_{j_y,j_x}(z)$$

$$\phi_{zz} = \sum_{j_x=0}^{n}\sum_{j_y=0}^{j_x}k^2_{j_y,j_x}\mathcal{R}\rceil\Big\{C_{j_y,j_x}(x,y,t)\Big\}Z_{j_y,j_x}(z) = -\phi_{xx} - \phi_{yy}$$

$$\frac{\partial^2\zeta}{\partial\bar{x}^2}(\bar{x},\bar{y},\bar{t}) = \zeta_{xx}\cos^2\beta - \zeta_{xy}\sin(2\beta) + \zeta_{yy}\sin^2\beta$$

$$\frac{\partial^2\zeta}{\partial\bar{x}\partial\bar{y}}(\bar{x},\bar{y},\bar{t}) = \zeta_{xy}(\cos^2\beta - \sin^2\beta) + (\zeta_{xx} - \zeta_{yy})\sin\beta\cos\beta$$

$$\frac{\partial^2\zeta}{\partial\bar{y}^2}(\bar{x},\bar{y},\bar{t}) = \zeta_{yy}\cos^2\beta + \zeta_{xy}\sin(2\beta) + \zeta_{xx}\sin^2\beta$$

$$\zeta_{xx} = -\sum_{j_x=0}^{n}\sum_{j_y=0}^{j_x}\mathcal{R}\rceil\Big\{H^{2,0}_{j_y,j_x}(x,y,t)\Big\}$$

$$\zeta_{xy} = -\sum_{j_x=0}^{n}\sum_{j_y=0}^{j_x}\mathcal{R}\rceil\Big\{H^{1,1}_{j_y,j_x}(x,y,t)\Big\}$$

$$\zeta_{yy} = -\sum_{j_x=0}^{n}\sum_{j_y=0}^{j_x}\mathcal{R}\rceil\Big\{H^{0,2}_{j_y,j_x}(x,y,t)\Big\}$$

$$p = -\rho\frac{\partial\phi}{\partial\bar{t}} - \frac{1}{2}\rho\bar{\nabla}\phi\cdot\bar{\nabla}\phi - \rho g\bar{z}$$

where $\bar{\nabla}$ denotes gradients with respect to $\bar{x}$, $\bar{y}$ and $\bar{z}$. We also apply the notation

$$\frac{\partial^{i+j}C_{j_y,j_x}(x,y,t)}{\partial x^i\partial y^j} = (-i)^{i+j}C^{i,j}_{j_y,j_x}(x,y,t)$$
$$\frac{\partial^{i+j}H_{j_y,j_x}(x,y,t)}{\partial x^i\partial y^j} = (-i)^{i+j}H^{i,j}_{j_y,j_x}(x,y,t)$$

$$\begin{aligned}C^{i,j}_{j_y,j_x}(x,y,t) =& k^i_{j_x}k^j_{j_y}\big(c_{1,j_y,j_x}(t)\,Y_{j_y}(y) + (-1)^j c_{2,j_y,j_x}(t)\,\bar{Y}_{j_y}(y)\big)X_{j_x}(x) +\\ & k^i_{j_y}k^j_{j_x}\big(c_{3,j_y,j_x}(t)\,Y_{j_x}(y) + (-1)^j c_{4,j_y,j_x}(t)\,\bar{Y}_{j_x}(y)\big)X_{j_y}(x)\\ H^{i,j}_{j_y,j_x}(x,y,t) =& k^i_{j_x}k^j_{j_y}\big(h_{1,j_y,j_x}(t)\,Y_{j_y}(y) + (-1)^j h_{2,j_y,j_x}(t)\,\bar{Y}_{j_y}(y)\big)X_{j_x}(x) +\\ & k^i_{j_y}k^j_{j_x}\big(h_{3,j_y,j_x}(t)\,Y_{j_x}(y) + (-1)^j h_{4,j_y,j_x}(t)\,\bar{Y}_{j_x}(y)\big)X_{j_y}(x)\end{aligned}$$

The particle acceleration is labeled $\frac{d\bar{\nabla}\phi}{d\bar{t}}$.

The stream function φ is not relevant for short crested seas. Hence, we apply the dummy definition $\varphi = 0$ for all locations.

Implementation notes

Evaluation of costly transcendental functions (cos, sin, exp, ...) is significantly reduced by exploiting the following recursive relations

$$X_{j_x}(x) = X_1(x)\, X_{j_x-1}(x), \qquad Y_{j_y}(y) = Y_1(y)\, Y_{j_y-1}(y)$$

It should be noted that contrary to long crested seas, there are no trivial recursive relations for the z-dependent term $Z_{j_y,j_x}(z)$. This makes calculations of surface elevations significantly faster than calculations of other kinematics for short crested seas.

In case the *wave generator* applies a perturbation theory of order q we apply the following Taylor expansion above the calm free surface.

$$Z_{j_y,j_x}(z) = 1 + \sum_{p=1}^{q-1} \frac{(k_{j_y,j_x} z)^p}{p!}, \qquad z > 0$$

4.5.5 Shape class 5

This shape class describes general short crested spectral waves propagating in constant water depth d.

$$\phi(x,y,z,t) = \sum_{j_x=0}^{n_x} \sum_{j_y=-n_y}^{n_y} \mathcal{R}\rceil\Big\{ c_{j_y,j_x}(t)\, X_{j_x}(x)\, Y_{j_y}(y) \Big\}\, Z_{j_y,j_x}(z)$$

$$\zeta(x,y,t) = \sum_{j_x=0}^{n_x} \sum_{j_y=-n_y}^{n_y} \mathcal{R}\rceil\Big\{ h_{j_y,j_x}(t)\, X_{j_x}(x)\, Y_{j_y}(y) \Big\}$$

$$X_{j_x}(x) = e^{-ik_{j_x}x}, \quad Y_{j_y}(y) = e^{-ik_{j_y}y}, \quad Z_{j_y,j_x}(z) = \frac{\cosh k_{j_y,j_x}(z+d)}{\cosh k_{j_y,j_x} d}$$

$$k_{j_x} = j_x \cdot \Delta k_x, \quad k_{j_y} = j_y \cdot \Delta k_y, \quad k_{j_y,j_x} = \sqrt{k_{j_x}^2 + k_{j_y}^2}, \quad i = \sqrt{-1}$$

The set of real constants k_{j_x} and k_{j_y} resemble wave numbers in the x and y directions respectively. It follows that the kinematics is periodic in space

$$\phi(x+L_x, y+L_y, z, t) = \phi(x,y,z,t), \qquad \zeta(x+L_x, y+L_y, t) = \zeta(x,y,t)$$

$$L_x = \frac{2\pi}{\Delta k_x}, \qquad L_y = \frac{2\pi}{\Delta k_y}, \qquad \lambda_{\min} = \frac{2\pi}{\sqrt{(n_x \Delta k_x)^2 + (n_y \Delta k_y)^2}}$$

where $\lambda_{\min}$ is the shortest wave lengths resolved. The actual set of shape functions is uniquely defined by the five input parameters Δk_x, Δk_y, n_x, n_y and d.

Note: The fields related to $j_x = j_y = 0$ are uniform in space (DC bias). Non-zero values of $h_{0,0}(t)$ violates mass conservation. The amplitude $c_{0,0}(t)$ adds a uniform time varying ambient pressure field not influencing the flow field. Consequently, these components will by default be suppressed in the kinematic calculations. However, there is an option in the API for including all DC values provided by the *wave generator*.

The fields related to $j_x = n_x$ and $j_y = \pm n_y$ are expected to correspond to the Nyquist frequencies of the physical resolution applied in the *wave generator*. Hence, typical $n_x = \lfloor n_{x,fft}/2 \rfloor$ and $n_y = \lfloor n_{y,fft}/2 \rfloor$ where $n_{x,fft}$ and $n_{y,fft}$ are the physical spatial resolutions applied in the *wave generator*, in the x and y directions respectively.

A special implementation optimized for the case $\Delta k_x = \Delta k_y$ and $n_x = n_y$ should be implemented. Such a class will improve the computational speed.

Kinematics

In order to exploit some symmetric properties in the y-direction of this class we apply the following equivalent formulation for more efficient evaluations.

$$\phi(x,y,z,t) = \sum_{j_x=0}^{n_x}\sum_{j_y=0}^{n_y} \mathcal{R}\rceil\Big\{C_{1,j_y,j_x}(y,t)\,X_{j_x}(x)\Big\}\,Z_{j_y,j_x}(z)$$

$$\zeta(x,y,t) = \sum_{j_x=0}^{n_x}\sum_{j_y=0}^{n_y} \mathcal{R}\rceil\Big\{H_{1,j_y,j_x}(y,t)\,X_{j_x}(x)\Big\}$$

$$C_{1,j_y,j_x}(y,t) = c_{1,j_y,j_x}(t)\,Y_{j_y}(y) + c_{2,j_y,j_x}(t)\,\bar{Y}_{j_y}(y)$$

$$H_{1,j_y,j_x}(y,t) = h_{1,j_y,j_x}(t)\,Y_{j_y}(y) + h_{2,j_y,j_x}(t)\,\bar{Y}_{j_y}(y)$$

$$c_{1,j_y,j_x}(t) = c_{j_y,j_x}(t), \qquad c_{2,j_y,j_x}(t) = \begin{cases} c_{-j_y,j_x}(t), & j_y > 0, \\ 0, & j_y = 0 \end{cases}$$

$$h_{1,j_y,j_x}(t) = h_{j_y,j_x}(t), \qquad h_{2,j_y,j_x}(t) = \begin{cases} h_{-j_y,j_x}(t), & j_y > 0, \\ 0, & j_y = 0 \end{cases}$$

where $\bar{Y}_{j_y}(y)$ denotes the complex conjugate of $Y_{j_y}(y)$.

Given the definitions above we obtain the following explicit kinematics:

$$\phi(\bar{x},\bar{y},\bar{z},\bar{t}) = \sum_{j_x=0}^{n_x}\sum_{j_y=0}^{n_y} \mathcal{R}\rceil\Big\{C_{1,j_y,j_x}(y,t)\,X_{j_x}(x)\Big\}Z_{j_y,j_x}(z)$$

$$\varphi(\bar{x},\bar{y},\bar{z},\bar{t}) \equiv 0$$

$$\frac{\partial\phi}{\partial\bar{t}}(\bar{x},\bar{y},\bar{z},\bar{t}) = \sum_{j_x=0}^{n_x}\sum_{j_y=0}^{n_y} \mathcal{R}\rceil\Big\{\frac{dC_{1,j_y,j_x}(y,t)}{dt}\,X_{j_x}(x)\Big\}Z_{j_y,j_x}(z)$$

$$\zeta(\bar{x},\bar{y},\bar{t}) = \sum_{j_x=0}^{n_x}\sum_{j_y=0}^{n_y} \mathcal{R}\rceil\Big\{H_{1,j_y,j_x}(y,t)\,X_{j_x}(x)\Big\}$$

$$\frac{\partial\zeta}{\partial\bar{t}}(\bar{x},\bar{y},\bar{t}) = \sum_{j_x=0}^{n_x}\sum_{j_y=0}^{n_y} \mathcal{R}\rceil\Big\{\frac{dH_{1,j_y,j_x}(y,t)}{dt}\,X_{j_x}(x)\Big\}$$

$$\frac{\partial\zeta}{\partial\bar{x}}(\bar{x},\bar{y},\bar{t}) = \zeta_x\cos\beta - \zeta_y\sin\beta, \qquad \frac{\partial\zeta}{\partial\bar{y}}(\bar{x},\bar{y},\bar{t}) = \zeta_x\sin\beta + \zeta_y\cos\beta$$

$$\zeta_x = \sum_{j_x=0}^{n_x}\sum_{j_y=0}^{n_y} k_{j_x}\mathcal{I}\Updownarrow\Big\{H_{1,j_y,j_x}(y,t)\,X_{j_x}(x)\Big\}$$

$$\zeta_y = -\sum_{j_x=0}^{n_x}\sum_{j_y=0}^{n_y} k_{j_y}\mathcal{I}\Updownarrow\Big\{H_{2,j_y,j_x}(y,t)\,X_{j_x}(x)\Big\}$$

$$\bar{\nabla}\phi(\bar{x},\bar{y},\bar{z},\bar{t}) = [\phi_x\cos\beta - \phi_y\sin\beta, \phi_x\sin\beta + \phi_y\cos\beta, \phi_z]^T$$

$$\phi_x = \sum_{j_x=0}^{n_x}\sum_{j_y=0}^{n_y} k_{j_x}\mathcal{I}\Updownarrow\Big\{C_{1,j_y,j_x}(y,t)\,X_{j_x}(x)\Big\}\,Z_{j_y,j_x}(z)$$

$$\phi_y = \sum_{j_x=0}^{n_x}\sum_{j_y=0}^{n_y} k_{j_y}\mathcal{I}\Updownarrow\Big\{C_{2,j_y,j_x}(y,t)\,X_{j_x}(x)\Big\}\,Z_{j_y,j_x}(z)$$

$$\phi_z = \sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} \mathcal{R}\rceil\Big\{C_{1,j_y,j_x}(y,t)\, X_{j_x}(x)\Big\}\, \frac{dZ_{j_y,j_x}(z)}{dz}$$

$$\frac{\partial \bar{\nabla}\phi}{\partial \bar{t}}(\bar{x},\bar{y},\bar{z},\bar{t}) = [\phi_{xt}\cos\beta - \phi_{yt}\sin\beta, \phi_{xt}\sin\beta + \phi_{yt}\cos\beta, \phi_z]^T$$

$$\phi_{xt} = \sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_x} \mathcal{I}\Updownarrow\Big\{\frac{dC_{1,j_y,j_x}(y,t)}{dt}\, X_{j_x}(x)\Big\}\, Z_{j_y,j_x}(z)$$

$$\phi_{yt} = \sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_y} \mathcal{I}\Updownarrow\Big\{\frac{dC_{2,j_y,j_x}(y,t)}{dt}\, X_{j_x}(x)\Big\}\, Z_{j_y,j_x}(z)$$

$$\phi_{zt} = \sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} \mathcal{R}\rceil\Big\{\frac{dC_{1,j_y,j_x}(y,t)}{dt}\, X_{j_x}(x)\Big\}\, \frac{dZ_{j_y,j_x}(z)}{dz}$$

$$\frac{d\bar{\nabla}\phi}{d\bar{t}}(\bar{x},\bar{y},\bar{z},\bar{t}) = \frac{\partial \bar{\nabla}\phi}{\partial \bar{t}} + \bar{\nabla}\phi \cdot \bar{\nabla}\bar{\nabla}\phi$$

$$\bar{\nabla}\bar{\nabla}\phi(\bar{x},\bar{y},\bar{z},\bar{t}) = \begin{bmatrix} \phi_{\bar{x},\bar{x}} & \phi_{\bar{x},\bar{y}} & \phi_{\bar{x},\bar{z}} \\ \phi_{\bar{x},\bar{y}} & \phi_{\bar{y},\bar{y}} & \phi_{\bar{y},\bar{z}} \\ \phi_{\bar{x},\bar{z}} & \phi_{\bar{y},\bar{z}} & \phi_{\bar{z},\bar{z}} \end{bmatrix}$$

$$\phi_{\bar{x},\bar{x}} = \phi_{xx}\cos^2\beta - \phi_{xy}\sin(2\beta) + \phi_{yy}\sin^2\beta$$

$$\phi_{\bar{x},\bar{y}} = \phi_{xy}(\cos^2\beta - \sin^2\beta) + (\phi_{xx} - \phi_{yy})\sin\beta\cos\beta$$

$$\phi_{\bar{x},\bar{z}} = \phi_{xz}\cos\beta - \phi_{yz}\sin\beta$$

$$\phi_{\bar{y},\bar{y}} = \phi_{yy}\cos^2\beta + \phi_{xy}\sin(2\beta) + \phi_{xx}\sin^2\beta$$

$$\phi_{\bar{y},\bar{z}} = \phi_{yz}\cos\beta + \phi_{xz}\sin\beta$$

$$\phi_{\bar{z},\bar{z}} = \phi_{zz} = -\phi_{xx} - \phi_{yy}$$

$$\phi_{xx} = -\sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_x}^2 \mathcal{R}\rceil\Big\{C_{1,j_y,j_x}(y,t)\, X_{j_x}(x)\Big\} Z_{j_y,j_x}(z)$$

$$\phi_{xy} = -\sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_x} k_{j_y} \mathcal{R}\rceil\Big\{C_{2,j_y,j_x}(y,t)\, X_{j_x}(x)\Big\} Z_{j_y,j_x}(z)$$

$$\phi_{xz} = \sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_x} \mathcal{I}\Updownarrow\Big\{C_{1,j_y,j_x}(y,t)\, X_{j_x}(x)\Big\} \frac{dZ_{j_y,j_x}(z)}{dz}$$

$$\phi_{yy} = -\sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_y}^2 \mathcal{R}\rceil\Big\{C_{1,j_y,j_x}(y,t)\, X_{j_x}(x)\Big\} Z_{j_y,j_x}(z)$$

$$\phi_{yz} = \sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_y} \mathcal{I}\Updownarrow\Big\{C_{2,j_y,j_x}(y,t)\, X_{j_x}(x)\Big\} \frac{dZ_{j_y,j_x}(z)}{dz}$$

$$\phi_{zz} = \sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_y,j_x}^2 \mathcal{R}\rceil\Big\{C_{1,j_y,j_x}(y,t)\, X_{j_x}(x)\Big\} Z_{j_y,j_x}(z) = -\phi_{xx} - \phi_{yy}$$

$$\frac{\partial^2 \zeta}{\partial \bar{x}^2}(\bar{x},\bar{y},\bar{t}) = \zeta_{xx}\cos^2\beta - \zeta_{xy}\sin(2\beta) + \zeta_{yy}\sin^2\beta$$

$$\frac{\partial^2 \zeta}{\partial \bar{x} \partial \bar{y}}(\bar{x}, \bar{y}, \bar{t}) = \zeta_{xy}(\cos^2 \beta - \sin^2 \beta) + (\zeta_{xx} - \zeta_{yy}) \sin \beta \cos \beta$$

$$\frac{\partial^2 \zeta}{\partial \bar{y}^2}(\bar{x}, \bar{y}, \bar{t}) = \zeta_{yy} \cos^2 \beta + \zeta_{xy} \sin(2\beta) + \zeta_{xx} \sin^2 \beta$$

$$\zeta_{xx} = -\sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_x}^2 \mathcal{R}\rceil \Big\{ H_{1,j_y,j_x}(y,t)\, X_{j_x}(x) \Big\}$$

$$\zeta_{xy} = -\sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_x} k_{j_y} \mathcal{R}\rceil \Big\{ H_{2,j_y,j_x}(y,t)\, X_{j_x}(x) \Big\}$$

$$\zeta_{yy} = -\sum_{j_x=0}^{n_x} \sum_{j_y=0}^{n_y} k_{j_y}^2 \mathcal{R}\rceil \Big\{ H_{1,j_y,j_x}(y,t)\, X_{j_x}(x) \Big\}$$

$$p = -\rho \frac{\partial \phi}{\partial \bar{t}} - \frac{1}{2} \rho \bar{\nabla} \phi \cdot \bar{\nabla} \phi - \rho g \bar{z}$$

where $\bar{\nabla}$ denotes gradients with respect to $\bar{x}$, $\bar{y}$ and $\bar{z}$. $\mathcal{I}\Updownarrow\{\alpha\}$ denotes the imaginary part of a complex number α and

$$C_{2,j_y,j_x}(y,t) = c_{1,j_y,j_x}(t)\, Y_{j_y}(y) - c_{2,j_y,j_x}(t)\, \bar{Y}_{j_y}(y)$$

$$H_{2,j_y,j_x}(y,t) = h_{1,j_y,j_x}(t)\, Y_{j_y}(y) - h_{2,j_y,j_x}(t)\, \bar{Y}_{j_y}(y)$$

The particle acceleration is labeled $\frac{d\bar{\nabla}\phi}{d\bar{t}}$.

The stream function φ is not relevant for short crested seas. Hence, we apply the dummy definition $\varphi = 0$ for all locations.

Implementation notes

Evaluation of costly transcendental functions (cos, sin, exp, cosh, ...) is significantly reduced by exploiting the following recursive relations

$$X_{j_x}(x) = X_1(x)\, X_{j_x-1}(x), \qquad Y_{j_y}(y) = Y_1(y)\, Y_{j_y-1}(y)$$

$$Z_{j_y,j_x}(z) = U_{j_y,j_x} S_{j_y,j_x} + V_{j_y,j_x} T_{j_y,j_x}, \qquad \frac{dZ_{j_y,j_x}(z)}{dz} = k_{j_y,j_x}(U_{j_y,j_x} S_{j_y,j_x} - V_{j_y,j_x} T_{j_y,j_x})$$

$$U_{j_y,j_x} = \frac{1 + R_{j_y,j_x}}{2}, \qquad V_{j_y,j_x} = 1 - U_{j_y,j_x}$$

$$S_{j_y,j_x} \equiv e^{k_{j_y,j_x} z}, \quad T_{j_y,j_x} \equiv e^{-k_{j_y,j_x} z} = 1/S_{j_y,j_x}, \quad R_{j_y,j_x} \equiv \tanh k_{j_y,j_x} d$$

It should be noted that contrary to long crested seas, there are no trivial recursive relations for the z-dependent terms S_{j_y,j_x}, T_{j_y,j_x} and R_{j_y,j_x}. This makes calculations of surface elevations significantly faster than calculations of other kinematics for short crested seas.

In case the *wave generator* applies a perturbation theory of order q we apply the following Taylor expansion above the calm free surface.

$$S_{j_y,j_x}(z) = 1 + \sum_{p=1}^{q-1} \frac{(k_{j_y,j_x} z)^p}{p!}, \qquad z > 0$$

4.5.6 Shape class 6

This shape class describes a general set of linear Airy waves propagating in infinite or constant water depth d. Schemes for evaluation of kinematics above $z = 0$ is described below.

Note: Due to the flexibility of allowing varying spacing of wave frequencies the default implementation of this shape class is not ideal for simulation of irregular seas with a large number of frequencies. For such simulations it is highly recommended to apply shape class 1, 2, 4 or 5 for speed considerations.

$$\phi(x,y,z,t) = \sum_{j=1}^{n} \frac{-gA_j}{\omega_j} Z_j(z) \sin(\omega_j t - k_{j_x} x - k_{j_y} y + \delta_j) = \sum_{j=1}^{n} \mathcal{R}\rceil \Big\{ c_j(t)\, E_j(x,y) \Big\}\, Z_j(z)$$

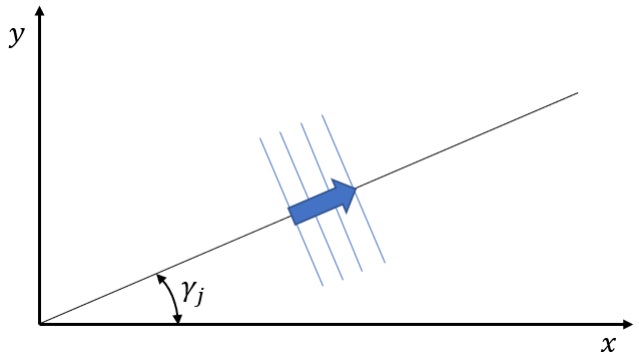
$$\zeta(x,y,t) = \sum_{j=1}^{n} A_j \cos(\omega_j t - k_{j_x} x - k_{j_y} y + \delta_j) = \sum_{j=1}^{n} \mathcal{R}\rceil \Big\{ h_j(t)\, E_j(x,y) \Big\}$$

$$E_j(x,y) = e^{-i(k_{j_x} x + k_{j_y} y)}, \qquad Z_j(z) = \frac{\cosh k_j(z+d)}{\cosh k_j d}$$

$$c_j(t) = i\frac{gA_j}{\omega_j} e^{i(\omega_j t + \delta_j)}, \qquad h_j(t) = A_j e^{i(\omega_j t + \delta_j)}$$

$$\omega_j^2 = k_j g \tanh(k_j d), \quad k_{j_x} = k_j \cos\gamma_j, \quad k_{j_y} = k_j \sin\gamma_j, \quad i = \sqrt{-1}$$

Wave component j propagates with an angle γ_j relative to the SWD x-axis as indicated in this figure.



The single amplitude of the wave component is A_j and the oscillation frequency is ω_j. The phase is δ_j. The acceleration of gravity is denoted g.

For infinite water depth d we obtain the asymptotic relations

$$Z_j(z) = e^{k_j z}, \qquad \omega_j^2 = k_j g$$

The actual set of shape functions is uniquely defined by the input parameters A_j, k_j, γ_j, δ_j, n, g and d. Negative d on input indicates infinite water depth.

Kinematics

Given the definitions above we obtain the following explicit kinematics:

$$\phi(\bar{x},\bar{y},\bar{z},\bar{t}) = \sum_{j=1}^{n} \mathcal{R}\rceil\Big\{c_j(t)\,E_j(x,y)\Big\}\,Z_j(z)$$

$$\varphi(\bar{x},\bar{y},\bar{z},\bar{t}) = \begin{cases} \sum_{j=1}^{n} \mathcal{I}\Updownarrow\Big\{c_j(t)\,E_j(x,y)\Big\}\hat{Z}_j(z), & \text{if all } \gamma_j = \gamma_1, \\ 0, & \text{otherwise} \end{cases}$$

$$\frac{\partial\phi}{\partial\bar{t}}(\bar{x},\bar{y},\bar{z},\bar{t}) = \sum_{j=1}^{n} \mathcal{R}\rceil\Big\{\frac{dc_j(t)}{dt}\,E_j(x,y)\Big\}\,Z_j(z)$$

$$\zeta(\bar{x},\bar{y},\bar{t}) = \sum_{j=1}^{n} \mathcal{R}\rceil\Big\{h_j(t)\,E_j(x,y)\Big\}$$

$$\frac{\partial\zeta}{\partial\bar{t}}(\bar{x},\bar{y},\bar{t}) = \sum_{j=1}^{n} \mathcal{R}\rceil\Big\{\frac{dh_j(t)}{dt}\,E_j(x,y)\Big\}$$

$$\frac{\partial\zeta}{\partial\bar{x}}(\bar{x},\bar{y},\bar{t}) = \zeta_x\cos\beta - \zeta_y\sin\beta, \qquad \frac{\partial\zeta}{\partial\bar{y}}(\bar{x},\bar{y},\bar{t}) = \zeta_x\sin\beta + \zeta_y\cos\beta$$

$$\zeta_x = \sum_{j=1}^{n} k_{j_x}\mathcal{I}\Updownarrow\Big\{h_j(t)\,E_j(x,y)\Big\}$$

$$\zeta_y = \sum_{j=1}^{n} k_{j_y}\mathcal{I}\Updownarrow\Big\{h_j(t)\,E_j(x,y)\Big\}$$

$$\bar{\nabla}\phi(\bar{x},\bar{y},\bar{z},\bar{t}) = [\phi_x\cos\beta - \phi_y\sin\beta, \phi_x\sin\beta + \phi_y\cos\beta, \phi_z]^T$$

$$\phi_x = \sum_{j=1}^{n} k_{j_x}\mathcal{I}\Updownarrow\Big\{c_j(t)\,E_j(x,y)\Big\}\,Z_j(z)$$

$$\phi_y = \sum_{j=1}^{n} k_{j_y}\mathcal{I}\Updownarrow\Big\{c_j(t)\,E_j(x,y)\Big\}\,Z_j(z)$$

$$\phi_z = \sum_{j=1}^{n} \mathcal{R}\rceil\Big\{c_j(t)\,E_j(x,y)\Big\}\,\frac{dZ_j(z)}{dz}$$

$$\frac{\partial\bar{\nabla}\phi}{\partial\bar{t}}(\bar{x},\bar{y},\bar{z},\bar{t}) = [\phi_{xt}\cos\beta - \phi_{yt}\sin\beta, \phi_{xt}\sin\beta + \phi_{yt}\cos\beta, \phi_z]^T$$

$$\phi_{xt} = \sum_{j=1}^{n} k_{j_x}\mathcal{I}\Updownarrow\Big\{\frac{dc_j(t)}{dt}\,E_j(x,y)\Big\}\,Z_j(z)$$

$$\phi_{yt} = \sum_{j=1}^{n} k_{j_y}\mathcal{I}\Updownarrow\Big\{\frac{dc_j(t)}{dt}\,E_j(x,y)\Big\}\,Z_j(z)$$

$$\phi_{zt} = \sum_{j=1}^{n} \mathcal{R}\rceil\Big\{\frac{dc_j(t)}{dt}\,E_j(x,y)\Big\}\,\frac{dZ_j(z)}{dz}$$

$$\frac{d\bar{\nabla}\phi}{d\bar{t}}(\bar{x},\bar{y},\bar{z},\bar{t}) = \frac{\partial\bar{\nabla}\phi}{\partial\bar{t}} + \bar{\nabla}\phi\cdot\bar{\nabla}\bar{\nabla}\phi$$

$$\bar{\nabla}\bar{\nabla}\phi(\bar{x},\bar{y},\bar{z},\bar{t}) = \begin{bmatrix} \phi_{\bar{x},\bar{x}} & \phi_{\bar{x},\bar{y}} & \phi_{\bar{x},\bar{z}} \\ \phi_{\bar{x},\bar{y}} & \phi_{\bar{y},\bar{y}} & \phi_{\bar{y},\bar{z}} \\ \phi_{\bar{x},\bar{z}} & \phi_{\bar{y},\bar{z}} & \phi_{\bar{z},\bar{z}} \end{bmatrix}$$

$$\phi_{\bar{x},\bar{x}} = \phi_{xx}\cos^2\beta - \phi_{xy}\sin(2\beta) + \phi_{yy}\sin^2\beta$$

$$\phi_{\bar{x},\bar{y}} = \phi_{xy}(\cos^2\beta - \sin^2\beta) + (\phi_{xx} - \phi_{yy})\sin\beta\cos\beta$$

$$\phi_{\bar{x},\bar{z}} = \phi_{xz}\cos\beta - \phi_{yz}\sin\beta$$

$$\phi_{\bar{y},\bar{y}} = \phi_{yy}\cos^2\beta + \phi_{xy}\sin(2\beta) + \phi_{xx}\sin^2\beta$$

$$\phi_{\bar{y},\bar{z}} = \phi_{yz}\cos\beta + \phi_{xz}\sin\beta$$

$$\phi_{\bar{z},\bar{z}} = \phi_{zz} = -\phi_{xx} - \phi_{yy}$$

$$\phi_{xx} = -\sum_{j=1}^{n} k_{j_x}^2 \mathcal{R}\rceil\Big\{c_j(t)\,E_j(x,y)\Big\}\,Z_j(z)$$

$$\phi_{xy} = -\sum_{j=1}^{n} k_{j_x}k_{j_y} \mathcal{R}\rceil\Big\{c_j(t)\,E_j(x,y)\Big\}\,Z_j(z)$$

$$\phi_{xz} = \sum_{j=1}^{n} k_{j_x} \mathcal{I}\Updownarrow\Big\{c_j(t)\,E_j(x,y)\Big\}\,\frac{dZ_j(z)}{dz}$$

$$\phi_{yy} = -\sum_{j=1}^{n} k_{j_y}^2 \mathcal{R}\rceil\Big\{c_j(t)\,E_j(x,y)\Big\}\,Z_j(z)$$

$$\phi_{yz} = \sum_{j=1}^{n} k_{j_y} \mathcal{I}\Updownarrow\Big\{c_j(t)\,E_j(x,y)\Big\}\,\frac{dZ_j(z)}{dz}$$

$$\phi_{zz} = \sum_{j=1}^{n} \mathcal{R}\rceil\Big\{c_j(t)\,E_j(x,y)\Big\}\,\frac{d^2Z_j(z)}{dz^2} = -\phi_{xx} - \phi_{yy}$$

$$\frac{\partial^2\zeta}{\partial\bar{x}^2}(\bar{x},\bar{y},\bar{t}) = \zeta_{xx}\cos^2\beta - \zeta_{xy}\sin(2\beta) + \zeta_{yy}\sin^2\beta$$

$$\frac{\partial^2\zeta}{\partial\bar{x}\partial\bar{y}}(\bar{x},\bar{y},\bar{t}) = \zeta_{xy}(\cos^2\beta - \sin^2\beta) + (\zeta_{xx} - \zeta_{yy})\sin\beta\cos\beta$$

$$\frac{\partial^2\zeta}{\partial\bar{y}^2}(\bar{x},\bar{y},\bar{t}) = \zeta_{yy}\cos^2\beta + \zeta_{xy}\sin(2\beta) + \zeta_{xx}\sin^2\beta$$

$$\zeta_{xx} = -\sum_{j=1}^{n} k_{j_x}^2 \mathcal{R}\rceil\Big\{h_j(t)\,E_j(x,y)\Big\}$$

$$\zeta_{xy} = -\sum_{j=1}^{n} k_{j_x}k_{j_y} \mathcal{R}\rceil\Big\{h_j(t)\,E_j(x,y)\Big\}$$

$$\zeta_{yy} = -\sum_{j=1}^{n} k_{j_y}^2 \mathcal{R}\rceil\Big\{h_j(t)\,E_j(x,y)\Big\}$$

$$p = -\rho\frac{\partial\phi}{\partial\bar{t}} - \frac{1}{2}\rho\bar{\nabla}\phi\cdot\bar{\nabla}\phi - \rho g\bar{z}$$

where $\bar{\nabla}$ denotes gradients with respect to $\bar{x}$, $\bar{y}$ and $\bar{z}$. $\mathcal{R}\rceil\{\alpha\}$ and $\mathcal{I}\Updownarrow\{\alpha\}$ denote the real and imaginary part of a complex number α.

The particle acceleration is labeled $\frac{d\bar{\nabla}\phi}{d\bar{t}}$.

The stream function φ is only relevant for long crested waves and is related to the velocity potential ϕ. Hence $\partial\phi/\partial x = \partial\varphi/\partial z$ and $\partial\phi/\partial z = -\partial\varphi/\partial x$. Note that for the stream function evaluation we apply the function

$$\hat{Z}_j(z) = \frac{\sinh k_j(z+d)}{\cosh k_j d}$$

Kinematics above $z = 0$

The kinematics above $z = 0$ is not well defined by the Airy theory. However, the following schemes are defined by the constructor parameter **norder**.

norder	Scheme
< 0	Always evaluate the exponential terms at the actual location. This is not recommended unless the wave slope is very small.
0	Evaluate the exponential terms at $z \leftarrow min(z, 0)$ (default)
1	Apply linear extrapolation above $z = 0$
2	Wheeler stretching: Evaluate exponential terms at $z \leftarrow \frac{z-\zeta(x,y,t)}{1+\zeta(x,y,t)/d}$

It should be noted that the wave field is not harmonic when using e.g. Wheeler stretching. Consequently, all kinematics become more approximative. (E.g. no mass conservation)

At steep crests the choice of **norder** may significantly influence the results.

For the linear extrapolation scheme we obtain:

$$Z_j(z) = 1 + tanh(k_j d)\, k_j z, \qquad \hat{Z}_j(z) = tanh(k_j d) + k_j z, \qquad z > 0$$

Implementation notes

For Airy waves the temporal derivatives of $c_j(t)$ and $h_j(t)$ have the analytical expressions

$$\frac{dc_j(t)}{dt} = -g h_j(t), \quad \frac{dh_j(t)}{dt} = i\omega_j h_j(t)$$

Note also that

$$\frac{dZ_j(z)}{dz} = k_j \hat{Z}_j(z), \qquad \frac{d^2 Z_j(z)}{dz^2} = k_j^2 Z_j(z)$$

$$\frac{d\hat{Z}_j(z)}{dz} = k_j Z_j(z), \qquad \frac{d^2 \hat{Z}_j(z)}{dz^2} = k_j^2 \hat{Z}_j(z)$$

$$Z_j(z) = a\frac{1+c}{1+b}, \qquad \hat{Z}_j(z) = a\frac{1-c}{1+b}$$

$$a = e^{k_j z} \quad b = e^{-2k_j d} < 1, \quad c = b/a^2 \le 1$$

These exact fractional expressions for $Z_j(z)$ and $\hat{Z}_j(z)$ are always numerical stable. For the linear extrapolation scheme above $z = 0$ we get:

$$Z_j(z) = 1 + \frac{1-b}{1+b} k_j z, \qquad \hat{Z}_j(z) = \frac{1-b}{1+b} + k_j z$$

4.6 Temporal amplitudes

Applying a shape class as defined above, the kinematics at time t is well defined given the set of associated temporal amplitudes $c_j(t)$ and $h_j(t)$. The *wave generators* are required by this API to provide discrete values of these functions at equidistant time intervals. In order to obtain consistent smooth kinematics in *application programs*, it is also required that $\frac{dc_j(t)}{dt}$ and $\frac{dh_j(t)}{dt}$ are provided by the *wave generators* at the same time intervals. In order to minimize errors in *application programs* the latter functions should be provided directly from the kinematic and dynamic free surface conditions and not by using temporal finite difference schemes:

$$\frac{\partial \phi}{\partial t} = \sum_{j=1}^{n} \mathcal{R}\rceil\left\{\frac{dc_j(t)}{dt}\,\psi_j(\mathbf{x})\right\} = f_d(\phi, \nabla\phi, \ldots)$$

$$\frac{\partial \zeta}{\partial t} = \sum_{j=1}^{n} \mathcal{R}\rceil\left\{\frac{dh_j(t)}{dt}\,\xi_j(x,y)\right\} = f_k(\phi, \nabla\phi, \ldots)$$

where the functions $f_d()$ and $f_k()$ follows from the applied dynamic and kinematic free surface conditions and are calculated using *spatial* gradients only.

Note:

- For spectral solvers $\frac{dc_j(t)}{dt}$ and $\frac{dh_j(t)}{dt}$ are obtained from the right-hand-side of the temporal ordinary-differential-equation system.
- Only a few time steps of $c_j(t)$, $h_j(t)$, $\frac{dc_j(t)}{dt}$ and $\frac{dh_j(t)}{dt}$ needs to be in memory for evaluating the kinematics at any time instance.
- The presence of explicit values of $\frac{dc_j(t)}{dt}$ and $\frac{dh_j(t)}{dt}$ ensures that the kinematics produced by the API comply with the physical boundary conditions at the specified instances.
- When applying interpolation schemes including constraints of derivatives, the issue of over-shooting is reduced.

4.6.1 Consistent interpolation

Given the value and slope of $c_j(t)$ and $h_j(t)$ at equidistant time instances $t_i, t_{i+1}, \ldots$ it is important that the predicted kinematics between these intervals are consistent and sufficiently smooth for proper application in other solvers. E.g. if the potential ϕ is assumed linear between two time steps the pressure distribution will be discontinuous at every new time interval due to the pressure component $\frac{\partial \phi}{\partial t}$. This is not acceptable.

In order to deal with simulations over a long time range it is important that the constant time intervals $\Delta t = t_{i+1} - t_i$ can be relatively large. Even slightly larger than in the program providing the kinematics. The program that applies the kinematics may have order of magnitude smaller time steps like in slamming CFD simulations. The kinematics provided by this API should not introduce numerical noise in such calculations.

Let f_i and $\frac{df_i}{dt}$ denote $c_j(t_i)$ and $\frac{dc_j(t_i)}{dt}$ or $h_j(t_i)$ and $\frac{dh_j(t_i)}{dt}$ respectively. When the application program updates the actual time t to an arbitrary value, the API will once approximate the values of $f(t)$ and $\frac{df(t)}{dt}$ using one of the interpolation schemes described below. Then any kinematics at time t can be evaluated using these current spectral amplitudes.

Two alternative interpolation schemes are provided.

C^1 continuous spline

Given the spectral data at two subsequent time steps t_i and t_{i+1}

$$f_i, f_{i+1}, \frac{df_i}{dt}, \frac{df_{i+1}}{dt}$$

we obtain a cubic spline which is C^1 continuous at the boundary of each time interval.

$$f(t) = (1-\delta)f_i + \delta f_{i+1} + \delta(1-\delta)(a(1-\delta)+b\delta), \qquad t \in [t_i, t_{i+1}]$$

$$\frac{df}{dt}(t) = \frac{f_{i+1}-f_i}{\Delta t} + (1-2\delta)\frac{a(1-\delta)+b\delta}{\Delta t} + \delta(1-\delta)\frac{b-a}{\Delta t}$$

$$\delta = \frac{t-t_i}{\Delta t}, \qquad a = \Delta t \frac{df_i}{dt} - (f_{i+1}-f_i)$$

$$b = -\Delta t \frac{df_{i+1}}{dt} + (f_{i+1}-f_i)$$

This spline will produce a pressure field which is C^0 continuous at the transition of each time step.

C^2 continuous spline

Note: This is the default scheme in the API.

Some boundary value formulations require continuity of quantities like $\frac{\partial^2 \phi}{\partial t^2}$ and $\frac{\partial^2 \zeta}{\partial t^2}$. Since we know the first order derivatives from the dynamic and kinematic free surface conditions we construct a spline which is second order continuous at the transitions of time steps by first approximating the second order gradients at these points.

The following approximations

$$\frac{d^2 f_i}{dt^2} \approx 2\frac{f_{i-1} - 2f_i + f_{i+1}}{\Delta t^2} - \frac{\frac{df_{i+1}}{dt} - \frac{df_{i-1}}{dt}}{2\Delta t}$$

$$\frac{d^2 f_{i+1}}{dt^2} \approx 2\frac{f_i - 2f_{i+1} + f_{i+2}}{\Delta t^2} - \frac{\frac{df_{i+2}}{dt} - \frac{df_i}{dt}}{2\Delta t}$$

are obtained applying the central finite difference scheme as explained in this figure.

Knowing the value, slope and approximate curvature at t_i and t_{i+1} the lowest order unique spline $f(t)$, and its derivative $df/dt(t)$, on this interval follows.

$$f(t) = f_i + \sum_{i=1}^{5} q_i \delta^i, \qquad \frac{df}{dt}(t) = \frac{df_i}{dt} + \frac{1}{\Delta t}\sum_{i=2}^{5} i q_i \delta^{i-1} \qquad t \in [t_i, t_{i+1}]$$

$$\delta = \frac{t-t_i}{\Delta t} \in [0,1), \quad q_1 = \frac{df_i}{dt}\Delta t, \quad q_2 = f_{i-1} - 2f_i + f_{i+1} + \left(\frac{df_{i-1}}{dt} - \frac{df_{i+1}}{dt}\right)\frac{\Delta t}{4}$$

$$q_3 = -3f_{i-1} - 3f_i + 5f_{i+1} + f_{i+2} - \left(3\frac{df_{i-1}}{dt} + 23\frac{df_i}{dt} + 13\frac{df_{i+1}}{dt} + \frac{df_{i+2}}{dt}\right)\frac{\Delta t}{4}$$

$$q_4 = 3f_{i-1} + 7f_i - 8f_{i+1} - 2f_{i+2} + \left(3\frac{df_{i-1}}{dt} + 30\frac{df_i}{dt} + 25\frac{df_{i+1}}{dt} + 2\frac{df_{i+2}}{dt}\right)\frac{\Delta t}{4}$$

$$q_5 = -f_{i-1} - 3f_i + 3f_{i+1} + f_{i+2} - \left(\frac{df_{i-1}}{dt} + 11\frac{df_i}{dt} + 11\frac{df_{i+1}}{dt} + \frac{df_{i+2}}{dt}\right)\frac{\Delta t}{4}$$

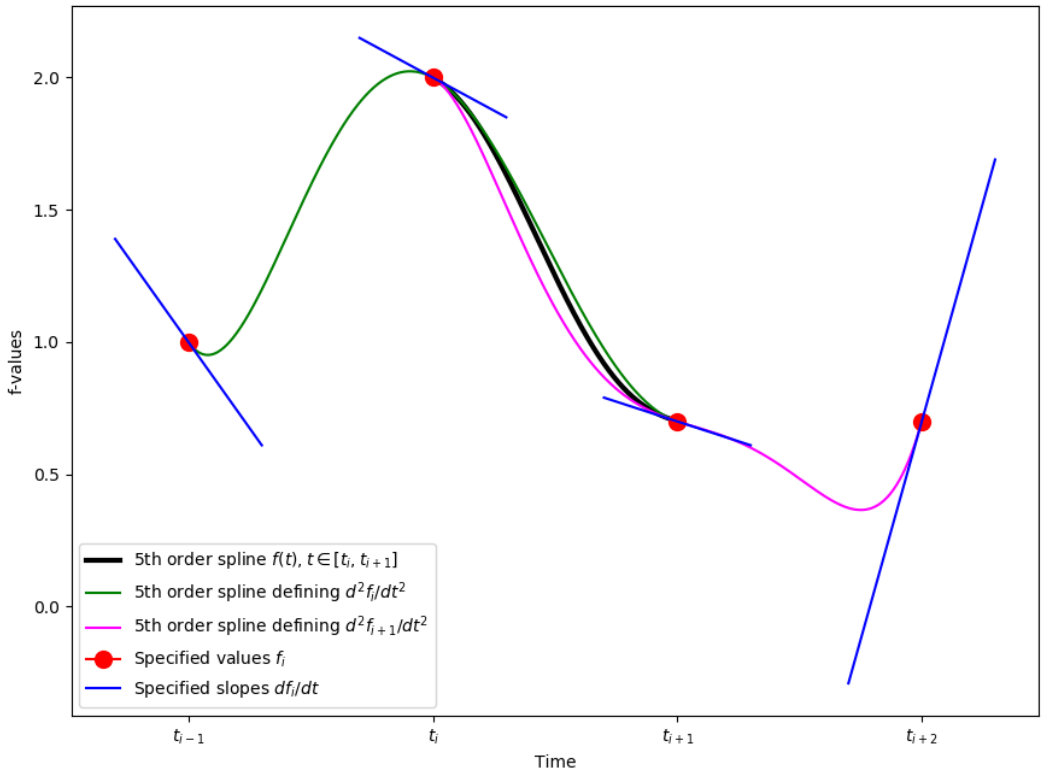


Fig. 2: The green curve is the lowest order polynomial matching the values and slopes at t_{i-1}, t_i, and t_{i+1}. The magenta curve is the lowest order polynomial matching the values and slopes at t_i, t_{i+1}, and t_{i+2}. The 5th order polynomial $f(t)$ matches the values and slopes at t_i, and t_{i+1}. In addition $f(t)$ matches the curvature of the green curve at t_i and the curvature of the magenta curve at t_{i+1}.

At the first time step ($i = 1$) of the simulation we apply the following "constant acceleration" padding of data when applying the formulas above.

$$f_0 = f_1 + \Big(\frac{df_2}{dt} - 3\frac{df_1}{dt}\Big)\frac{\Delta t}{2}$$

$$\frac{df_0}{dt} = 2\frac{df_1}{dt} - \frac{df_2}{dt}$$

Similar we apply the following padding at the very last time step ($i = n$).

$$f_{n+1} = f_n - \Big(\frac{df_{n-1}}{dt} - 3\frac{df_n}{dt}\Big)\frac{\Delta t}{2}$$

$$\frac{df_{n+1}}{dt} = 2\frac{df_n}{dt} - \frac{df_{n-1}}{dt}$$

It should be noted that the spline $f(t)$ is constructed using only 4 time steps in memory. However, the curve is still C^2 continuous at the transition of each time step.

CHAPTER

FIVE

SWD FORMAT

In this section we describe the format of the SWD-file. Symbols follow closely the definitions in the *theory* section.

Note: This section is not relevant for programmers making *application programs*. The API automatically takes care of all file handling. However, when making *wave generators*, the details described below needs to be understood for your relevant shape class as defined in the *theory* section.

Note: In order to ensure portability between various computer systems SWD files are required to be stored using the 'little endian' byte order convention. This is the default convention for all Intel/AMD CPUs running Windows, Linux or Apple. PowerPC CPUs (older Macintosh machines and IBM clusters and some supercomputers) use big endian. Wave generators making SWD files on big endian systems are required to specify a flag to signal that the SWD file should be written in little endian. This is usually specifed in the file open statement, but can also be defined using environment variables.

The units of all quantities are required to be consistent with the SI-system.

The SWD-file is a C-stream binary file (see the *rationals*).

The binary stream in the SWD file is outlined in the following pseudo code:

```
magic
fmt
shp, amp
prog
date
nid
cid
grav, lscale
nstrip, nsteps, dt
order
select case(shp)
case(1)
   n, dk
   do i = 1, nsteps
        h(0), h(1), ..., h(n)
        ht(0), ht(1), ..., ht(n)
        if amp < 3 then
           c(0), c(1), ..., c(n)
           ct(0), ct(1), ..., ct(n)
        end if
   end do
case(2)
```

(continues on next page)

(continued from previous page)

```
   n, dk, d
   do i = 1, nsteps
       h(0), h(1), ..., h(n)
       ht(0), ht(1), ..., ht(n)
       if amp < 3 then
          c(0), c(1), ..., c(n)
          ct(0), ct(1), ..., ct(n)
       end if
   end do
case(3)
   n, nh, dk, isf
   nsf
   xsf(1), xsf(2), ..., xsf(nsf)
   zsf(1), zsf(2), ..., zsf(nsf)
   do i = 1, nsteps
       h(0), h(1), ..., h(n)
       ht(0), ht(1), ..., ht(n)
       if amp < 3 then
           c(0), c(1), ..., c(n)
           ct(0), ct(1), ..., ct(n)
           if nsf > 1 then
              ch(0), ch(1), ..., ch(nh)
              cht(0), cht(1), ..., cht(nh)
           end if
       end if
   end do
case(4)
   nx, ny, dkx, dky
   do i = 1, nsteps
       h(-ny,0), h(-ny+1,0), ..., h(ny-1,nx), h(ny,nx)         ! Fortran element order
       ht(-ny,0), ht(-ny+1,0), ..., ht(ny-1,nx), ht(ny,nx)
       if amp < 3 then
           c(-ny,0), c(-ny+1,0), ..., c(ny-1,nx), c(ny,nx)
           ct(-ny,0), ct(-ny+1,0), ..., ct(ny-1,nx), ct(ny,nx)
       end if
   end do
case(5)
   nx, ny, dkx, dky, d
   do i = 1, nsteps
       h(-ny,0), h(-ny+1,0), ..., h(ny-1,nx), h(ny,nx)         ! Fortran element order
       ht(-ny,0), ht(-ny+1,0), ..., ht(ny-1,nx), ht(ny,nx)
       if amp < 3 then
           c(-ny,0), c(-ny+1,0), ..., c(ny-1,nx), c(ny,nx)
           ct(-ny,0), ct(-ny+1,0), ..., ct(ny-1,nx), ct(ny,nx)
       end if
   end do
case(6)
   n, d
   do i = 1, n
       amp(i), kw(i), gam(i), phs(n)
   end do
end select
```

where

name	C type	bytes	description
magic	float	4	magic = 37.0221 Constant decimal number in all SWD files. (future proof)
fmt	int	4	Integer to identify version of the file format. In this version: fmt = 100
shp	int	4	Actual shape functions as defined in the *theory* section. (e.g. shp = 3 implies *Shape 3* is applied)
amp	int	4	Flag to indicate which temporal amplitudes are stored in the swd-file 1: All temporal amplitudes related to the shape class is specified as defined in the *theory* section. 2: All amplitudes related to the shape class is specified. However, the velocity potential is only interpreted on the free surface. The z-dependencies are removed from all formulas. This option is introduced for reasearch on how to map the potential to other vertical locations. More accurate calculations are possible but at significantly higher computational cost. The current implementation of the API does not support this feature. 3: Functions related to the velocity potential are not specified. It is only possible to evaluate surface elevation quantities with this option. For other calculations $\phi \equiv 0$ is applied.
prog	30 char	30	The name of the program building this file. (Including version number)
date	20 char	20	Text providing the date and time this file was build.
nid	int	4	Number of characters describing the next field. (nid > 0)
cid	nid char	nid	Text to describe the wave field. It is expected to be the complete content of the input file applied in the wave generator.
grav	float	4	Acceleration of gravity applied in the wave generator (Not applied in current version)
lscale	float	4	Number of length units per meter applied in the wave generator. (Not applied in current version)
nstrip	int	4	Number of initial time steps stripped off from the original simulation. Default nstrip=0. The strip() method will remove some initial and trailing time steps. nstrip can be used to deduce the original time reference.
nsteps	int	4	Number of time steps stored in this swd file.

continues on next page

Table 1 – continued from previous page

name	C type	bytes	description
dt	float	4	Constant time step for spectral amplitudes stored in this file
order	int	4	Perturbation order applied in the *wave generator*. (<0 if fully non-linear)
n, nx	int	4	Number of spectral components, n or n_x, in the x-direction.
ny	int	4	Number of spectral components n_y in the y-direction.
nh	int	4	Number of auxiliary spectral components $\hat{n}$ in the x-direction in case of bathymetry.
dk, dkx	float	4	Spacing of wave numbers, Δk or Δk_x, in the x-direction.
dky	float	4	Spacing of wave numbers Δk_y in the y-direction.
d	float	4	Constant or average water depth d. (<0 if infinite)
isf	int	4	Flag to indicate geometric description of the sea floor 0: Piecewise linear sea floor
nsf	int	4	Number of offset points defining the sea floor in x-direction. 0: Infinite water depth 1: Constant water depth >1: Varying water depth (bathymetry)
xsf()	float	4	x-positions of offset points defining the sea floor. xsf() should cover the range $x \in [0, 2\pi/\Delta k]$.
zsf()	float	4	z-positions of offset points defining the sea floor.
h()	complex	4+4	Spectral amplitudes $h()$ (real and imaginary part)
ht()	complex	4+4	Spectral amplitudes $\frac{dh}{dt}()$ (real and imaginary part)
c()	complex	4+4	Spectral amplitudes $c()$ (real and imaginary part)
ct()	complex	4+4	Spectral amplitudes $\frac{dc}{dt}()$ (real and imaginary part)
ch()	complex	4+4	Spectral amplitudes $\hat{c}()$ (real and imaginary part)
cht()	complex	4+4	Spectral amplitudes $\frac{d\hat{c}}{dt}()$ (real and imaginary part)
amp()	float	4	Single amplitudes A_j for Airy model (shp=6).
kw()	float	4	Wave number k_j for Airy model (shp=6).
gam()	float	4	Wave direction γ_j (rad) for Airy model (shp=6).
phs()	float	4	Wave phase δ_j (rad) for Airy model (shp=6).

CHAPTER

SIX

THE API

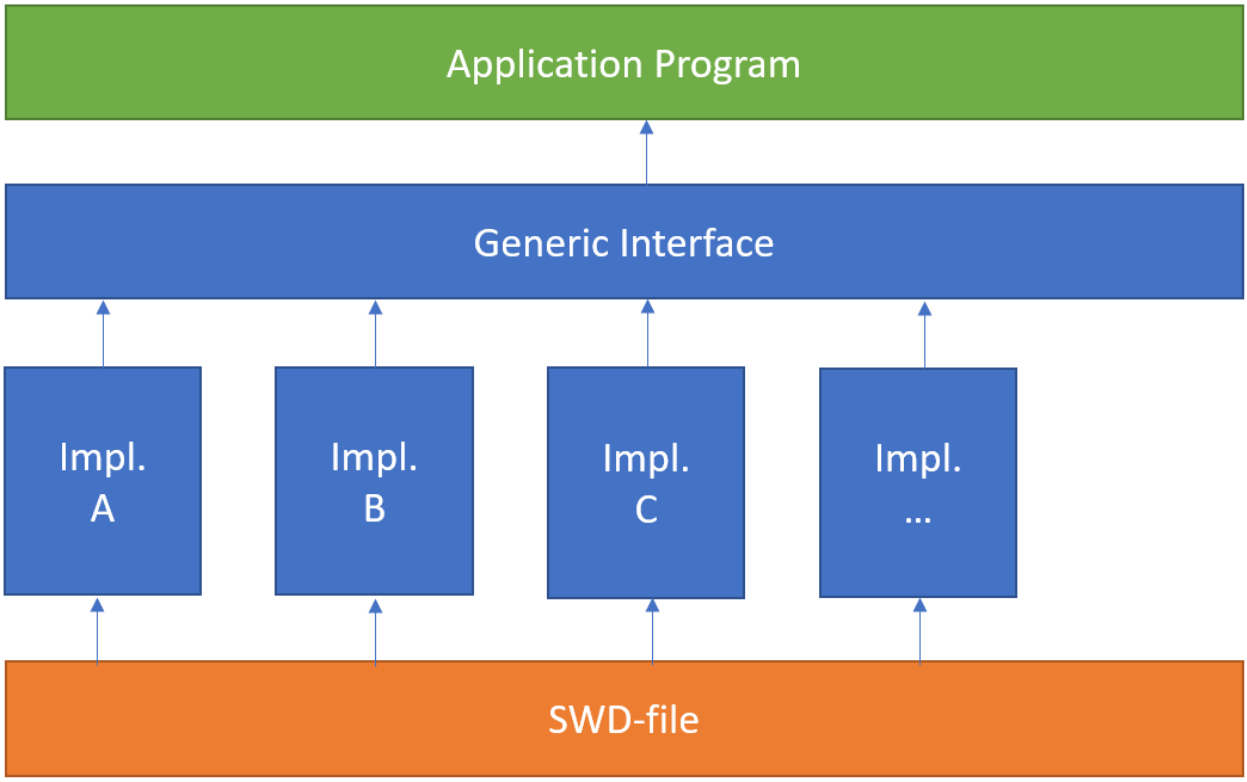


Fig. 1: A generic interface handling all applied spectral wave formulations

In this section we define the official API to be utilized in *application programs*.

In general the API provides an abstract base class defining a generic syntax for how to obtain kinematics and to extract metadata.

Specialized classes are derived from the base class in order to implement actual algorithms for a specific wave formulation. For an actual shape type, as defined in the *theory* section, there might be several specialized classes based on different implementations and algorithms.

The API provides a special constructor which allocate and initialize an instance of a proper specialized class based on the content of the actual SWD and an optional request for specific implementation.

When the specific instance has been initialized, the generic syntax from the base class applies in the *application program* is independent on the actual wave formulation.

Examples of using the API are located in the *programming application* section. It is recommended to study those examples before investigating the API.

The language specific API's are very similar:

- *Python*
- *C++*
- *C*
- *Fortran*

6.1 Python 2 and 3

The Python package **spectral_wave_data** contains a module *spectral_wave_data.py*. It defines the generic class `SpectralWaveData` to be applied in applications based on Python-2 or Python-3.

6.1.1 Constructor and methods

Detailed documentation of class members is provided *here*.

SWD constructor and methods

The documentation for each method in this table is autogenerated from the Python source code.

`SpectralWaveData.__init__`(file_swd, x0, y0, ...)	Constructor
`SpectralWaveData.__getitem__`(key)	self[key] is an alias for calling the method `get()`
`SpectralWaveData.update_time`(time)	Set the current time for all kinematic calculations.
`SpectralWaveData.acc_euler`(x, y, z)	Calculates the Euler acceleration (grad(phi_t)) at (x,y,z).
`SpectralWaveData.acc_particle`(x, y, z)	Calculates the particle acceleration at (x,y,z).
`SpectralWaveData.bathymetry`(x, y)	Calculates the vertical distance from z=0 to the sea floor at (x, y).
`SpectralWaveData.bathymetry_nvec`(x, y)	Return the unit normal vector of the sea floor at (x,y).
`SpectralWaveData.close`()	Manual destructor closing the object and the SWD-file.
`SpectralWaveData.convergence`(x, y, z, csv)	For a specific location create a CSV-file on how velocity, elevation and pressure converge as a function of number of spectral components.
`SpectralWaveData.elev`(x, y)	Calculates the wave elevation at (x,y).
`SpectralWaveData.elev_t`(x, y)	Calculates the time derivative of the wave elevation at earth fixed location.
`SpectralWaveData.get`(key)	Extract metadata from the SWD object
`SpectralWaveData.grad_elev`(x, y)	Calculates the gradients of the wave elevation.
`SpectralWaveData.grad_elev_2nd`(x, y)	Calculates the 2nd order gradients of the wave elevations.
`SpectralWaveData.grad_phi`(x, y, z)	Calculates the particle velocity.
`SpectralWaveData.grad_phi_2nd`(x, y, z)	Calculates the 2nd order derivatives of the wave potential.
`SpectralWaveData.phi`(x, y, z)	Calculates the velocity potential at the actual location.

continues on next page

Table 1 – continued from previous page

`SpectralWaveData.phi_t`(x, y, z)	Calculates the time derivative of the wave potential at the actual location (Euler derivative).
`SpectralWaveData.pressure`(x, y, z)	Calculates the complete nonlinear Bernoulli-pressure (Const=0).
`SpectralWaveData.stream`(x, y, z)	Calculates the stream function at the actual location.
`SpectralWaveData.strip`(tmin, tmax, file_swd)	Create a new SWD file containing only the time steps within the time window [tmin, tmax].

orphan SpectralWaveData.__init__ ==========

spectral_wave_data

method

orphan SpectralWaveData.__getitem__ ==========

spectral_wave_data

method

orphan SpectralWaveData.update_time ==========

spectral_wave_data

method

orphan SpectralWaveData.acc_euler ==========

spectral_wave_data

method

orphan SpectralWaveData.acc_particle ==========

spectral_wave_data

method

orphan SpectralWaveData.bathymetry ==========

spectral_wave_data

method

orphan SpectralWaveData.bathymetry_nvec ==========

spectral_wave_data

method

orphan SpectralWaveData.close ==========

spectral_wave_data

method

orphan SpectralWaveData.convergence ==========

spectral_wave_data

method

orphan SpectralWaveData.elev ==========

spectral_wave_data

method

orphan SpectralWaveData.elev_t ========================

spectral_wave_data

method

orphan SpectralWaveData.get ====================

spectral_wave_data

method

orphan SpectralWaveData.grad_elev ==========================

spectral_wave_data

method

orphan SpectralWaveData.grad_elev_2nd ================================

spectral_wave_data

method

orphan SpectralWaveData.grad_phi ==========================

spectral_wave_data

method

orphan SpectralWaveData.grad_phi_2nd ================================

spectral_wave_data

method

orphan SpectralWaveData.phi ====================

spectral_wave_data

method

orphan SpectralWaveData.phi_t ========================

spectral_wave_data

method

orphan SpectralWaveData.pressure ========================

spectral_wave_data

method

orphan SpectralWaveData.stream ========================

spectral_wave_data

method

orphan SpectralWaveData.strip ========================

spectral_wave_data

method

6.1.2 Exception handling

Related to the class `SpectralWaveData` is a set of class specific exceptions defined in the same module. These exceptions as described in the table below, can be imported and applied in application programs and scripts.

```
>>> from spectral_wave_data import SpectralWaveData, SwdError, SwdInputValueError, ...
>>> ...
>>> try:
>>>     swd = SpectralWaveData('my_waves.swd', x0=0.0, y0=0.0, t0=0.0, beta=180.0)
>>> except SwdError as err:
>>>     print(err)  # Print actual error message
>>>     # You may do some application specific error recovery before the show must go␣
↪on...
>>> ...
```

If no try/except block is applied, the application program will abort and print the exception and backtrace as a normal Python crash.

The only other associated class methods that may throw exceptions are `update_time()`, `convergence()`, `strip()` and `get()`.

Python Exceptions	Usage
`SwdError`	Base class for all SWD specific exceptions
`SwdFileCantOpenError`	Typical if SWD file is not an existing file.
`SwdFileBinaryError`	SWD file does not apply float/little-endian
`SwdFileDataError`	Error during reading and checking SWD data
`SwdInputValueError`	Input arguments for class methods are not sound
`SwdAllocateError`	Not able to allocate internal SWD storage

6.1.3 Metadata

To extract metadata from a SWD file you may apply the method `swd.get(key)`, or using the more Pythonesque syntax `swd[key]`, where *key* is a string to identify the requested metadata. A key is either:

- A relevant parameter from the *SWD-file format description*.
- A constructor parameter.
- A key from the table below.

Only scalar meta data is supported in this version.

Key	Returned value
version	The repository version number of **spectral_wave_data** applied in this Python distribution.
class	Name of the specialized SWD-class handling this object. (Fortran class)
tmax	Maximum allowed application time consistent with the header of the SWD file and the constructor parameter *t0=0.0*. [s]
lmin	Shortest wave length component. [m]
lmax	Longest wave length component. [m]
sizex	Periodic length of wave domain in x-direction (swd-system). [m]
sizey	Periodic length of wave domain in y-direction (swd-system). [m]

A `SwdInputValueError` exception is raised if the actual key is not relevant for the actual SWD class.

The script swd_meta

For convenience this Python wheel package includes the script `swd_meta` listing the relevant metadata for a given SWD-file. It runs on Windows and Linux.

In a terminal window with access to your installed **spectral_wave_data** package you can invoke it like in this example where `my.swd` is the name of the actual SWD-file:

```
>>> swd_meta my.swd
version: 1.0.0-beta.9
prog:    raschii-1.0.3.dev0
date:    2020:01:22 19:57:55
fmt:     100
shp:     2
amp:     1
tmax:    6.3000000938773155
dt:      0.10000000149011612
nsteps:  64
nstrip:  0
order:   -1
depth:   32.0
n:       50
sizex:   220.00000561733003
lmax:    220.00000561733003
lmin:    4.400000112346601
dk:      0.028559932485222816
cid:     {'model': 'Fenton', 'T': 12.792885811907514, 'height': 18.5, 'depth': 32.0,
↪'N': 50, 'air': 'NoneType', 'g': 9.81, 'c': 17.19705805512825, 'relax': 0.5}
```

6.1.4 Implementation

In the current version the implementation of the class `SpectralWaveData` is obtained by wrapping the C-implementation using the standard Python module `ctypes` in the package module *swd_c_interface.py*.

spectral_wave_data.py

```
# -*- coding: utf-8 -*-

"""
:platform: Linux, Windows, python 2.7 and 3.x
:synopsis: Python wrapper for the spectral_wave_data ocean wave model.

Author  - Jens Bloch Helmers, DNVGL
Created - 2019-08-11
"""

from __future__ import division
from __future__ import absolute_import
from __future__ import print_function
from __future__ import unicode_literals

import os

__all__ = ['SpectralWaveData', 'SwdError', 'SwdFileCantOpenError',
           'SwdFileBinaryError', 'SwdFileDataError', 'SwdInputValueError',
           'SwdAllocateError']

from .swd_c_interface import swdlib, vecswd, vecphi2ndswd, vecelev2ndswd


class SwdError(Exception):
    pass


class SwdFileCantOpenError(SwdError):
    pass


class SwdFileBinaryError(SwdError):
    pass


class SwdFileDataError(SwdError):
    pass


class SwdInputValueError(SwdError):
    pass


class SwdAllocateError(SwdError):
    pass


class SpectralWaveData(object):
```

(continues on next page)

(continued from previous page)

```
    """An object of SpectraWaveData is an instance of the SWD ocean wave model.

    Raises
    ------
    SwdError
         Base class for the following exceptions
    SwdFileCantOpenError
         Typical if SWD file is not an existing file
    SwdFileBinaryError
         SWD file does not apply float/little-endian
    SwdFileDataError
         Error during reading and checking data
    SwdInputValueError
         Input arguments for class methods are not sound
    SwdAllocateError
         Not able to allocate internal SWD storage

    Attributes
    ----------
    No public attributes

    """

    def __init__(self, file_swd, x0, y0, t0, beta, rho=1025.0, nsumx=-1,
                 nsumy=-1, impl=0, ipol=0, norder=0, dc_bias=False):
        """Constructor

        Parameters
        ----------
        file_swd : str
            The name of the swd file defining the ocean waves.
        x0, y0 : float
            The origin of the application wave coordinate system relative to the
            SWD coordinate system. [m]
        t0 : float
            The SWD time corresponding to t=0 in the application simulation. [s]
        beta : float
            Rotation of the SWD x-axis relative to the application x-axis. [deg]
        rho : float, optional
            Density of water. [kg/m^3] (Only relevant for pressure calculations)
        nsumx, nsumy : int, optional
            Number of applied spectral components in x and y-directions. Higher␣
↪frequency
            components in the swd file will not be applied. nsumx < 0 and nsumy < 0
            indicates that all components from the swd file will be applied.
        impl : int, optional
            Index to request actual derived SWD class
                *  0 = Apply a default class based on the content of this SWD-file.
                * <0 = Apply an in-house and experimental implementation
                * >0 = Apply a specific implementations from the open software
        ipol : int, optional
            Index to request actual temporal interpolation scheme
                *  0 = Default (C^2 continous scheme)
                *  1 = C^1 continous scheme
                *  2 = C^3 continous scheme
        norder : int, optional
             Expansion order to apply in kinematics for z>0
```

(continues on next page)

(continued from previous page)

```
                *  0 = Apply expansion order specified on swd file (default)
                * <0 = Apply exp(kj z)
                * >0 = Apply expansion order = norder
        dc_bias : bool, optional
             Control application of zero-frequency bias present in SWD file
                * False = Suppress contribution from zero frequency amplitudes␣
↪(default)
                * True  = Apply zero frequency amplitudes from SWD file.

        Raises
        ------
        SwdError
             Base class for the following exceptions
        SwdFileCantOpenError
             Typical if SWD file is not an existing file
        SwdFileBinaryError
             SWD file does not apply float/little-endian
        SwdFileDataError
             Error during reading and checking data
        SwdInputValueError
             Input arguments for class methods are not sound
        SwdAllocateError
             Not able to allocate internal SWD storage

        Examples
        --------
        >>> from spectral_wave_data import SpectralWaveData
        >>> swd = SpectralWaveData('my_waves.swd', x0=0.0, y0=0.0, t0=0.0, beta=180.0)

        """
        self.obj = swdlib.swd_api_allocate(file_swd.encode('ascii'), x0, y0, t0, beta,
                                           rho, nsumx, nsumy, impl, ipol, norder, dc_
↪bias)
        if swdlib.swd_api_error_raised(self.obj):
            id = swdlib.swd_api_error_get_id(self.obj)
            msg = swdlib.swd_api_error_get_msg(self.obj).decode()
            if id == 1001:
                raise SwdFileCantOpenError(msg)
            elif id == 1002:
                raise SwdFileBinaryError(msg)
            elif id == 1003:
                raise SwdFileDataError(msg)
            elif id == 1004:
                raise SwdInputValueError(msg)
            elif id == 1005:
                raise SwdAllocateError(msg)
            else:
                raise SwdError(msg)
        self._alive = True

    def update_time(self, time):
        """Set the current time for all kinematic calculations.

        .. note::

           This method must be called at least once before any kinematic
           calculations can be done.
```

(continues on next page)

(continued from previous page)

```
        Parameters
        ----------
        time : float
            Time as defined in the application program [s]

        Returns
        -------
        None

        Raises
        ------
        SwdError
             Base class for the following exceptions
        SwdFileDataError
             Error during reading the SWD file
        SwdInputValueError
             Input arguments are not sound

        Examples
        --------
        >>> swd.update_time(time=73.241)    # Time as defined in the application

        """
        swdlib.swd_api_update_time(self.obj, time)
        if swdlib.swd_api_error_raised(self.obj):
            id = swdlib.swd_api_error_get_id(self.obj)
            msg = swdlib.swd_api_error_get_msg(self.obj).decode()
            swdlib.swd_api_error_clear(self.obj) # To simplify safe recovery...
            if id == 1003:
                raise SwdFileDataError(msg)
            elif id == 1004:
                raise SwdInputValueError(msg)
            else:
                raise SwdError(msg)

    def phi(self, x, y, z):
        """Calculates the velocity potential at the actual location. It is
        assumed that the current time has been set using the method
        :meth:`update_time`.

        Parameters
        ----------
        x, y, z : float
            Position as defined in the application program [m]

        Returns
        -------
        float
            Velocity potential at (x,y,z) [m^2/s]

        Raises
        ------
        None

        Examples
        --------
```

(continues on next page)

(continued from previous page)

```
        >>> print("potential at (x,y,z) = ", swd.phi(x,y,z))

        """
        res = swdlib.swd_api_phi(self.obj, x, y, z)
        return res

    def stream(self, x, y, z):
        """Calculates the stream function at the actual location. It is
        assumed that the current time has been set using the method
        :meth:`update_time`.

        Parameters
        ----------
        x, y, z : float
            Position as defined in the application program [m]

        Returns
        -------
        float
            Stream function at (x,y,z)

        Raises
        ------
        None

        Examples
        --------
        >>> print("Stream function at (x,y,z) = ", swd.stream(x,y,z))

        """
        res = swdlib.swd_api_stream(self.obj, x, y, z)
        return res

    def phi_t(self, x, y, z):
        """Calculates the time derivative of the wave potential at the actual
        location (Euler derivative). It is assumed that the current time has
        been set using the method :meth:`update_time`.

        Parameters
        ----------
        x, y, z : float
            Position as defined in the application program [m]

        Returns
        -------
        float
            Time derivative (Euler) of wave potential at (x,y,z) [m^2/s^2]

        Raises
        ------
        None

        Examples
        --------
        >>> print("delta(phi)/delta(t) at (x,y,z) = ", swd.phi_t(x,y,z))

        """
```

(continues on next page)

(continued from previous page)

```
        res = swdlib.swd_api_phi_t(self.obj, x, y, z)
        return res

    def grad_phi(self, x, y, z):
        """Calculates the particle velocity. The velocity components are
        with respect to the application coordinate system. It is
        assumed that the current time has been set using the method
        :meth:`update_time`.

        Parameters
        ----------
        x, y, z : float
            Position as defined in the application program [m]

        Returns
        -------
        vecswd (struct with x, y and z float attributes)
            Particle velocity at (x,y,z) [m/s]

        Raises
        ------
        None

        Examples
        --------
        >>> vel = swd.grad_phi(x, y, z)
        >>> print("velocity in x-dir = ", vel.x)
        >>> print("velocity in y-dir = ", vel.y)
        >>> print("velocity in z-dir = ", vel.z)

        """
        res = swdlib.swd_api_grad_phi(self.obj, x, y, z)
        return res      # res.x, res.y, res.z

    def grad_phi_2nd(self, x, y, z):
        """Calculates the 2nd order derivatives of the wave potential.
        Gradients are with with respect to the application coordinate system.
        It is assumed that the current time has been set using the method
        :meth:`update_time`.

        Parameters
        ----------
        x, y, z : float
            Position as defined in the application program [m]

        Returns
        -------
        vecphi2ndswd (struct with xx, xy, xz, yy, yz, zz float attributes)
            2nd order derivatives of velocity potential [1/s]

        Raises
        ------
        None

        Examples
        --------
        >>> res = swd.grad_phi_2nd(x, y, z)
```

(continues on next page)

(continued from previous page)

```
        >>> print("phi_xx = ", res.xx)
        >>> print("phi_xy = ", res.xy)
        >>> print("phi_xz = ", res.xz)
        >>> print("phi_yy = ", res.yy)
        >>> print("phi_yz = ", res.yz)
        >>> print("phi_zz = ", res.zz)

        """
        res = swdlib.swd_api_grad_phi_2nd(self.obj, x, y, z)
        return res       # res.xx, res.xy, res.xz, res.yy, res.yz, res.zz

    def acc_euler(self, x, y, z):
        """Calculates the Euler acceleration (grad(phi_t)) at (x,y,z).
        Components are with respect to the application coordinate system.
        It is assumed that the current time has been set using the method
        :meth:`update_time`.

        Parameters
        ----------
        x, y, z : float
            Position as defined in the application program [m]

        Returns
        -------
        vecswd (struct with x, y and z float attributes)
            Euler acceleration at (x,y,z) [m/s^2]

        Raises
        ------
        None

        Examples
        --------
        >>> acc = swd.acc_euler(x, y, z)
        >>> print("Euler acceleration in x-dir = ", acc.x)
        >>> print("Euler acceleration in y-dir = ", acc.y)
        >>> print("Euler acceleration in z-dir = ", acc.z)

        """
        res = swdlib.swd_api_acc_euler(self.obj, x, y, z)
        return res      # res.x, res.y, res.z

    def acc_particle(self, x, y, z):
        """Calculates the particle acceleration at (x,y,z).
        Components are with respect to the application coordinate system.
        It is assumed that the current time has been set using the method
        :meth:`update_time`.

        Parameters
        ----------
        x, y, z : float
            Position as defined in the application program [m]

        Returns
        -------
        vecswd (struct with x, y and z float attributes)
            Particle acceleration at (x,y,z) [m/s^2]
```

(continues on next page)

(continued from previous page)

```
        Raises
        ------
        None

        Examples
        --------
        >>> acc = swd.acc_particle(x, y, z)
        >>> print("Particle acceleration in x-dir = ", acc.x)
        >>> print("Particle acceleration in y-dir = ", acc.y)
        >>> print("Particle acceleration in z-dir = ", acc.z)

        """
        res = swdlib.swd_api_acc_particle(self.obj, x, y, z)
        return res      # res.x, res.y, res.z

    def elev(self, x, y):
        """Calculates the wave elevation at (x,y). It is
        assumed that the current time has been set using the method
        :meth:`update_time`.

        Parameters
        ----------
        x, y : float
            Horizontal position as defined in the application program [m]

        Returns
        -------
        float
            Wave elevation at (x,y) [m]

        Raises
        ------
        None

        Examples
        --------
        >>> print("Wave elevation at (x,y) = ", swd.elev(x,y))

        """
        res = swdlib.swd_api_elev(self.obj, x, y)
        return res

    def elev_t(self, x, y):
        """Calculates the time derivative of the wave elevation
        at earth fixed location. It is assumed that the current
        time has been set using the method :meth:`update_time`.

        Parameters
        ----------
        x, y : float
            Horizontal position as defined in the application program [m]

        Returns
        -------
        float
            Time derivative of wave elevation at (x,y)  [m/s]
```

(continues on next page)

(continued from previous page)

```
        Raises
        ------
        None

        Examples
        --------
        >>> print("Time derivative of elevation at (x,y) = ", swd.elev_t(x,y))

        """
        res = swdlib.swd_api_elev_t(self.obj, x, y)
        return res

    def grad_elev(self, x, y):
        """Calculates the gradients of the wave elevation. The gradients
        are with respect to the application coordinate system. It is
        assumed that the current time has been set using the method
        :meth:`update_time`.

        Parameters
        ----------
        x, y : float
            Horizontal location as defined in the application program [m]

        Returns
        -------
        vecswd (struct with x, y and z float attributes)
            Spatial gradients of surface elevation at (x,y) [-]

        Raises
        ------
        None

        Examples
        --------
        >>> res = swd.grad_elev(x, y)
        >>> print("d/dx(elevation) = ", res.x)
        >>> print("d/dy(elevation) = ", res.y)
        >>> res.z
        0.0

        """
        res = swdlib.swd_api_grad_elev(self.obj, x, y)
        return res      # res.x, res.y, res.z=0

    def grad_elev_2nd(self, x, y):
        """Calculates the 2nd order gradients of the wave elevations. The gradients
        are with respect to the application coordinate system. It is
        assumed that the current time has been set using the method
        :meth:`update_time`.

        Parameters
        ----------
        x, y : float
            Horizontal location as defined in the application program [m]

        Returns
```

(continues on next page)

(continued from previous page)

```
        -------
        vecelev2ndswd (struct with xx, xy and yy float attributes)
            2nd order derivatives of wave elevation at (x,y) [1/m]

        Raises
        ------
        None

        Examples
        --------
        >>> res = swd.grad_elev_2nd(x, y)
        >>> print("elvation_xx = ", res.xx)
        >>> print("elvation_xy = ", res.xy)
        >>> print("elvation_yy = ", res.yy)

        """
        res = swdlib.swd_api_grad_elev_2nd(self.obj, x, y)
        return res       # res.xx, res.xy, res.yy

    def bathymetry(self, x, y):
        """Calculates the vertical distance from z=0 to the sea floor at (x, y).

        Parameters
        ----------
        x, y : float
            Horizontal position as defined in the application program [m]

        Returns
        -------
        float
            Local water depth at (x,y)  (<0 indicates infinite depth) [m]

        Raises
        ------
        None

        Examples
        --------
        >>> print("Local water depth at (x,y) = ", swd.bathymetry(x,y))

        """
        res = swdlib.swd_api_bathymetry(self.obj, x, y)
        return res

    def bathymetry_nvec(self, x, y):
        """Return the unit normal vector of the sea floor at (x,y).
        The orientation of the vector is into the ocean and with
        respect to the application coordinate system.

        Parameters
        ----------
        x, y : float
            Horizontal location as defined in the application program [m]

        Returns
        -------
        vecswd (struct with x, y and z float attributes)
```

(continues on next page)

(continued from previous page)

```
            Unit normal vector on sea floor at (x,y)

        Raises
        ------
        None

        Examples
        --------
        >>> nvec = swd.bathymetry_nvec(x, y)
        >>> print("n_x = ", nvec.x)
        >>> print("n_y = ", nvec.y)
        >>> print("n_z = ", nvec.z)  # Typical close to 1

        """
        res = swdlib.swd_api_bathymetry_nvec(self.obj, x, y)
        return res    # res.x, res.y, res.z

    def pressure(self, x, y, z):
        """Calculates the complete nonlinear Bernoulli-pressure (Const=0).
        It is assumed that the current time has been set using the method
        :meth:`update_time`.

        Parameters
        ----------
        x, y, z : float
            Position as defined in the application program [m]

        Returns
        -------
        float
            Pressure at (x,y,z)  [Pa]

        Raises
        ------
        None

        Examples
        --------
        >>> print("Total pressure at (x,y,z) = ", swd.pressure(x,y,z))

        """
        res = swdlib.swd_api_pressure(self.obj, x, y, z)
        return res

    def convergence(self, x, y, z, csv):
        """For a specific location create a CSV-file on how velocity, elevation
        and pressure converge as a function of number of spectral components.
        It is assumed that the current time has been set using the method
        :meth:`update_time`.

        Parameters
        ----------
        x, y, z : float
            Position as defined in the application program [m]
        csv : str
            Name of requested CSV-file to be generated.
```

(continues on next page)

(continued from previous page)

```
        Returns
        -------
        None

        Raises
        ------
        SwdError
            Base class for the following exceptions
        SwdFileCantOpenError
            If an existing CSV-file with the same name is locked.

        Examples
        --------
        >>> swd.convergence(x, y, z, 'convergence_data_at_xyz.csv')

        """
        swdlib.swd_api_convergence(self.obj, x, y, z, csv.encode('ascii'))
        if swdlib.swd_api_error_raised(self.obj):
            id = swdlib.swd_api_error_get_id(self.obj)
            msg = swdlib.swd_api_error_get_msg(self.obj).decode()
            swdlib.swd_api_error_clear(self.obj) # To simplify safe recovery...
            if id == 1001:
                raise SwdFileCantOpenError(msg)
            else:
                raise SwdError(msg)

    def strip(self, tmin, tmax, file_swd):
        """Create a new SWD file containing only the time steps within the
        time window [tmin, tmax].

        Parameters
        ----------
        tmin, tmax : float
            Time window as defined in the application program [s]
        file_swd : str
            Name of new SWD file containing only the actual time window

        Returns
        -------
        None

        Raises
        ------
        SwdError
            Base class for the following exceptions
        SwdFileCantOpenError
            If not able to open new SWD file
        SwdFileDataError
            Error during reading data from existing SWD file
        SwdInputValueError
            The time window is outside the content of existing SWD file.

        Examples
        --------
        >>> swd.strip(tmin=850.0, tmax=950.0, file_swd='freak_wave_at_850_950.swd')

        """
```

(continues on next page)

(continued from previous page)

```
        swdlib.swd_api_strip(self.obj, tmin, tmax, file_swd.encode('ascii'))
        if swdlib.swd_api_error_raised(self.obj):
            id = swdlib.swd_api_error_get_id(self.obj)
            msg = swdlib.swd_api_error_get_msg(self.obj).decode()
            swdlib.swd_api_error_clear(self.obj) # To simplify safe recovery...
            if id == 1001:
                raise SwdFileCantOpenError(msg)
            elif id == 1003:
                raise SwdFileDataError(msg)
            elif id == 1004:
                raise SwdInputValueError(msg)
            else:
                raise SwdError(msg)

    def __getitem__(self, key):
        """self[key] is an alias for calling the method :meth:`get`

        Examples
        --------
        >>> swd = SpectralWaveData('my_waves.swd', ...)
        >>> swd['tmax']     # Return max allowed application time

        """
        return self.get(key)

    def get(self, key):
        """Extract metadata from the SWD object

        Parameters
        ----------
        key : str
            A key to identify requested metadata. A key is either:
                 * A relevant parameter from the SWD-file format description.
                 * A constructor parameter.
                 * A key from the table below.

            ::

               Key         Returned Value
               ------------------------------------------------------------------------
               'version'   The repository version number of spectral-wave-data
                           applied in this Python distribution.
               'class'     Name of the specialized SWD-class handling this object.
               'tmax'      Maximum allowed application time consistent with the
                           header of the SWD file.
               'lmin'      Shortest wave length component. [m]
               'lmax'      Longest wave length component. [m]
               'sizex'     Periodic length of wave domain in x-direction (swd-system).
               'sizey'     Periodic length of wave domain in y-direction (swd-system).

        Returns
        -------
        float, int, bool, or str
            Value of actual metadata

        Raises
        ------
```

(continues on next page)

(continued from previous page)

```
        SwdInputValueError
            Key does not correspond to relevant metadata for this object.

        Examples
        --------
        >>> print("swd version number = ", swd.get('version'))
        >>> print("Max allowed user time = ", swd.get('tmax'))
        >>> print("Size of periodic domain in x-dir (SWD) = ", swd.get('sizex'))
        >>> print("Actual spectral shape model = ", swd.get('shp'))
        >>> print("Applied SWD-class = ", swd.get('class'))

        """
        key_c = key.encode('ascii')

        if key in ['file', 'file_swd', 'version', 'class', 'cid', 'prog', 'date']:
            res = swdlib.swd_api_get_chr(self.obj, key_c).decode()
        elif key in ['magic', 'grav', 'lscale', 'dt', 'dk', 'dkx', 'dky',
                     'd', 'depth', 'tmax', 'lmin', 'lmax',
                     'sizex', 'sizey', 't0', 'x0', 'y0', 'beta', 'rho']:
            res = swdlib.swd_api_get_real(self.obj, key_c)
        elif key in ['dc_bias']:
            res = swdlib.swd_api_get_bool(self.obj, key_c)
        else:
            res = swdlib.swd_api_get_int(self.obj, key_c)

        if swdlib.swd_api_error_raised(self.obj):
            id = swdlib.swd_api_error_get_id(self.obj)
            msg = swdlib.swd_api_error_get_msg(self.obj).decode()
            swdlib.swd_api_error_clear(self.obj) # To simplify safe recovery...
            if id == 1004:
                raise SwdInputValueError(msg)
            else:
                raise SwdError(msg)  # Just in case....

        return res

    def close(self):
        """Manual destructor closing the object and the SWD-file.

        Parameters
        ----------
        None

        Returns
        -------
        None

        Raises
        ------
        None

        """
        if self._alive is True:
            swdlib.swd_api_close(self.obj)
            self._alive = False
```

swd_c_interface.py

```
# -*- coding: utf-8 -*-

"""
:platform: Linux, Windows, python 2.7 and 3.x
:synopsis: Defines the Python-C interface

Author  - Jens Bloch Helmers, DNVGL
Created - 2019-08-11
"""

from __future__ import division
from __future__ import absolute_import
from __future__ import print_function
from __future__ import unicode_literals

import platform, sys, os
from ctypes import c_bool, c_double, c_int, c_char_p, c_void_p, Structure, CDLL

assert sys.version_info >= (2, 7, 11)

HERE = os.path.dirname(os.path.abspath(__file__))
if platform.system()=='Windows':
    intel_redist_path = os.getenv('INTEL_DEV_REDIST')
    if intel_redist_path is None:
        msg = """
              To apply "spectral_wave_data" you need to install the latest version of:
              1) Redistributable Libraries for Intel® C++ and Fortran Compilers for␣
↪Windows
                 This package can be downloaded for free from intel.com
              2) You also need Microsoft Visual C++ redistributables or
                 Visual Studio with C++ tools and libraries.
                 These tools can be downloaded from microsoft.com
              """
        raise AssertionError(msg)
    if sys.version_info >= (3, 8):
        intel_redist_path = os.path.join(intel_redist_path, 'redist', 'intel64',
↪'compiler')
        os.add_dll_directory(HERE)
        os.add_dll_directory(intel_redist_path)
    swdlib = CDLL(str(os.path.join(HERE, 'SpectralWaveData.dll')))
elif platform.system()=='Linux':
    swdlib = CDLL(str(os.path.join(HERE, 'libSpectralWaveData.so')))
else:
    raise AssertionError('Not supported platform: ' + platform.system())

"""
================================================================================================
BEGIN interface definition to the C-implementation
================================================================================================
NOTE: STRANGE ERRORS may occur if this interface does not comply with the original C␣
↪source code.
"""

class vecswd(Structure):
     _fields_ = [("x", c_double), ("y", c_double), ("z", c_double)]
```

(continues on next page)

(continued from previous page)

```
class vecphi2ndswd(Structure):
    _fields_ = [("xx", c_double), ("xy", c_double), ("xz", c_double),
                ("yy", c_double), ("yz", c_double), ("zz", c_double)]

class vecelev2ndswd(Structure):
    _fields_ = [("xx", c_double), ("xy", c_double), ("yy", c_double)]

swdlib.swd_api_allocate.argtypes = [c_char_p, c_double, c_double,
                                    c_double, c_double, c_double,
                                    c_int, c_int, c_int, c_int,
                                    c_int, c_bool]
swdlib.swd_api_allocate.restype = c_void_p

swdlib.swd_api_update_time.argtypes = [c_void_p, c_double]
swdlib.swd_api_update_time.restype = c_void_p

swdlib.swd_api_phi.argtypes = [c_void_p, c_double, c_double, c_double]
swdlib.swd_api_phi.restype = c_double

swdlib.swd_api_stream.argtypes = [c_void_p, c_double, c_double, c_double]
swdlib.swd_api_stream.restype = c_double

swdlib.swd_api_phi_t.argtypes = [c_void_p, c_double, c_double, c_double]
swdlib.swd_api_phi_t.restype = c_double

swdlib.swd_api_grad_phi.argtypes = [c_void_p, c_double, c_double, c_double]
swdlib.swd_api_grad_phi.restype = vecswd

swdlib.swd_api_grad_phi_2nd.argtypes = [c_void_p, c_double, c_double, c_double]
swdlib.swd_api_grad_phi_2nd.restype = vecphi2ndswd

swdlib.swd_api_acc_euler.argtypes = [c_void_p, c_double, c_double, c_double]
swdlib.swd_api_acc_euler.restype = vecswd

swdlib.swd_api_acc_particle.argtypes = [c_void_p, c_double, c_double, c_double]
swdlib.swd_api_acc_particle.restype = vecswd

swdlib.swd_api_elev.argtypes = [c_void_p, c_double, c_double]
swdlib.swd_api_elev.restype = c_double

swdlib.swd_api_elev_t.argtypes = [c_void_p, c_double, c_double]
swdlib.swd_api_elev_t.restype = c_double

swdlib.swd_api_grad_elev.argtypes = [c_void_p, c_double, c_double]
swdlib.swd_api_grad_elev.restype = vecswd

swdlib.swd_api_grad_elev_2nd.argtypes = [c_void_p, c_double, c_double]
swdlib.swd_api_grad_elev_2nd.restype = vecelev2ndswd

swdlib.swd_api_pressure.argtypes = [c_void_p, c_double, c_double, c_double]
swdlib.swd_api_pressure.restype = c_double

swdlib.swd_api_bathymetry.argtypes = [c_void_p, c_double, c_double]
swdlib.swd_api_bathymetry.restype = c_double
```

(continues on next page)

(continued from previous page)

```
swdlib.swd_api_bathymetry_nvec.argtypes = [c_void_p, c_double, c_double]
swdlib.swd_api_bathymetry_nvec.restype = vecswd

swdlib.swd_api_convergence.argtypes = [c_void_p, c_double, c_double, c_double, c_char_
↪p]
swdlib.swd_api_convergence.restype = c_void_p

swdlib.swd_api_strip.argtypes = [c_void_p, c_double, c_double, c_char_p]
swdlib.swd_api_strip.restype = c_void_p

swdlib.swd_api_get_chr.argtypes = [c_void_p, c_char_p]
swdlib.swd_api_get_chr.restype = c_char_p

swdlib.swd_api_get_int.argtypes = [c_void_p, c_char_p]
swdlib.swd_api_get_int.restype = c_int

swdlib.swd_api_get_bool.argtypes = [c_void_p, c_char_p]
swdlib.swd_api_get_bool.restype = c_bool

swdlib.swd_api_get_real.argtypes = [c_void_p, c_char_p]
swdlib.swd_api_get_real.restype = c_double

swdlib.swd_api_error_raised.argtypes = [c_void_p]
swdlib.swd_api_error_raised.restype = c_bool

swdlib.swd_api_error_get_id.argtypes = [c_void_p]
swdlib.swd_api_error_get_id.restype = c_int

swdlib.swd_api_error_get_msg.argtypes = [c_void_p]
swdlib.swd_api_error_get_msg.restype = c_char_p

swdlib.swd_api_error_clear.argtypes = [c_void_p]
swdlib.swd_api_error_clear.restype = c_void_p

swdlib.swd_api_close.argtypes = [c_void_p]
swdlib.swd_api_close.restype = c_void_p

"""
================================================================================
END interface to C implementation
================================================================================
"""
```

6.2 C++

The official C++ API is defined in the header file *SpectralWaveData.h*. It defines the generic class `SpectralWaveData` to be applied in applications. All member functions and input arguments are explained in that header file.

The data types `real_swd`, `vector_swd`, `vector_2nd_phi_swd` and `vector_2nd_elev_swd` are defined in the C-header file *spectral_wave_data.h*.

If your C++ application apply float (not double) for the SWD API interface you need to:

1. Set the macro `SWD_API_FLOAT` before the header file is included.

2. The fortran library should be compiled with *kind_swd_c = c_float* in the file `kind_values.f90`.

By default it is assumed that the C/C++ interface apply *double*. Hence *kind_swd_c = c_double* in the file `kind_values.f90` and the macro `SWD_API_FLOAT` should be unset.

C++ Exceptions	Usage
`SwdException`	Base class for all SWD specific exceptions
`SwdFileCantOpenException`	Typical if SWD file is not an existing file.
`SwdFileBinaryException`	SWD file does not apply float/little-endian
`SwdFileDataException`	Error during reading and checking SWD data
`SwdInputValueException`	Input arguments for class methods are not sound
`SwdAllocateException`	Not able to allocate internal SWD storage

6.2.1 SpectralWaveData.h

```
#ifndef SWD_CPP_H_INCLUDED
#define SWD_CPP_H_INCLUDED

/*
The C++ interface for the spectral-wave-data ocean wave model.

Ver-3.0:
Coded by: Jens B. Helmers DNVGL,  2019.08.11

*/
#include <string>
#include <stdexcept>

#include "spectral_wave_data.h"  // The core C interface

// Specific Exception Classes...

class SwdException : public std::runtime_error
{
public:
    SwdException(const char* msg) : std::runtime_error(msg) { }
};

class SwdFileCantOpenException : public SwdException
{
public:
    SwdFileCantOpenException(const char* msg) : SwdException(msg) { }
};

class SwdFileBinaryException : public SwdException
{
public:
    SwdFileBinaryException(const char* msg) : SwdException(msg) { }
};

class SwdFileDataException : public SwdException
{
public:
    SwdFileDataException(const char* msg) : SwdException(msg) { }
};
```

(continues on next page)

(continued from previous page)

```
class SwdInputValueException : public SwdException
{
public:
    SwdInputValueException(const char* msg) : SwdException(msg) { }
};

class SwdAllocateException : public SwdException
{
public:
    SwdAllocateException(const char* msg) : SwdException(msg) { }
};


// The main class...

class SpectralWaveData
{
    void* obj; // Wrapper to the C-object of the ocean wave model

public:
    SpectralWaveData(std::string file_swd, real_swd x0, real_swd y0,
                     real_swd t0, real_swd beta, real_swd rho=1025.0,
                     int nsumx=-1, int nsumy=-1, int impl=0, int ipol=0,
                     int norder=0, bool dc_bias=false);
    /*
    file_swd:           Name of actual swd file

    x0, y0, t0, beta:   Relation between SWD and application coordinates.
                        beta in degree.

    rho:                Density of water(applied for pressure calculations)
    nsumx, nsumy        Number of spectral components to apply (<0: apply all)

    impl                Index to determine actual derived class
                        0 = Default
                       <0 = In-house and experimental implementations
                       >0 = Validated implementations available open software

    ipol                Index to request actual temporal interpolation scheme
                        0 = Default (C^2 continous scheme)
                        1 = C^1 continous
                        2 = C^3 continous

    norder              Expansion order to apply in kinematics for z>0
                        0 = Apply expansion order specified in swd file (default)
                        <0 = Apply exp(kj z)
                        >0 = Apply expansion order = norder

    dc_bias             Control application of zero-frequency bias present in SWD file
                        false = Suppress contribution from zero frequency amplitudes␣
↪(default)
                        true  = Apply zero frequency amplitudes from SWD file.

    C++ exceptions the constructor may throw:  (should be catched in application)

    SwdException:               Base class for the following exceptions
```

(continues on next page)

(continued from previous page)

```
    SwdFileCantOpenException:   Typical if SWD file is not an existing file
    SwdFileBinaryException:     SWD file does not apply float/little-endian
    SwdFileDataException:       Error during reading and checking data
    SwdInputValueException:     Input arguments for class methods are not sound
    SwdAllocateException:       Not able to allocate internal SWD storage

    /**/

    ~SpectralWaveData();

    // ================================================================
    // Methods for evaluating kinematics...
    // ================================================================

    // Apply current application time
    // Possible exceptions thrown: SwdFileDataException, SwdInputValueException
    void UpdateTime(real_swd time);

    // Calculate velocity potential
    // Possible exceptions thrown: None
    real_swd Phi(real_swd x, real_swd y, real_swd z);

    // Calculate stream function
    // Possible exceptions thrown: None
    real_swd Stream(real_swd x, real_swd y, real_swd z);

    // Calculate time derivative of wave potential (earth fixed observer)
    // Possible exceptions thrown: None
    real_swd DdtPhi(real_swd x, real_swd y, real_swd z);

    // Calculate particle velocity
    // Possible exceptions thrown: None
    vector_swd GradPhi(real_swd x, real_swd y, real_swd z);

    // 2nd order gradients of potential
    // Possible exceptions thrown: None
    vector_2nd_phi_swd GradPhi2nd(real_swd x, real_swd y, real_swd z);

    // Calculate Euler acceleration: grad(phi_t)
    // Possible exceptions thrown: None
    vector_swd AccEuler(real_swd x, real_swd y, real_swd z);

    // Calculate particle acceleration
    // Possible exceptions thrown: None
    vector_swd AccParticle(real_swd x, real_swd y, real_swd z);

    // Calculate wave elevation
    // Possible exceptions thrown: None
    real_swd Elev(real_swd x, real_swd y);

    // Calculate time derivative of wave elevation (Euler derivative, earth fixed)
    // Possible exceptions thrown: None
    real_swd DdtElev(real_swd x, real_swd y);

    // Calculate gradient of wave elevation (slopes)
    // Possible exceptions thrown: None
    vector_swd GradElev(real_swd x, real_swd y);
```

(continues on next page)

(continued from previous page)

```
    //  2nd order gradients of elevation
    // Possible exceptions thrown: None
    vector_2nd_elev_swd GradElev2nd(real_swd x, real_swd y);

    // Complete Bernoulli pressure
    // Possible exceptions thrown: None
    real_swd Pressure(real_swd x, real_swd y, real_swd z);

    // Vertical distance from z=0 to sea floor (<0 if infinite)
    // Possible exceptions thrown: None
    real_swd Bathymetry(real_swd x, real_swd y);

    // Unit normal vector of sea floor into the ocean at(x, y)
    // Possible exceptions thrown: None
    vector_swd BathymetryNvec(real_swd x, real_swd y);

    // For a specific location return a csv-file on how velocity, elevation
    // and pressure converge as a function of number of spectral components
    // Possible exceptions thrown: SwdFileCantOpenException
    void Convergence(real_swd x, real_swd y, real_swd z, std::string csv);

    // To save storage for an interesting event you may create a new SWD file
    // containing only the time steps within the time window [tmin, tmax].
    // The name of the new file is defined by file_swd_new
    // Possible exceptions thrown: SwdFileCantOpenException
    //                             SwdInputValueException
    void Strip(real_swd tmin, real_swd tmax, std::string file_swd_new);

    // ================================================================
    //  Provide parameters from the swd-file:

    // Extract the character parameter 'name' from object
    // Possible exceptions thrown: SwdInputValueException
    std::string GetChr(std::string const& name);

    // Extract the int parameter 'name' from object
    // Possible exceptions thrown: SwdInputValueException
    int GetInt(std::string const& name);

        // Extract the int parameter 'name' from object
        // Possible exceptions thrown: SwdInputValueException
       bool GetBool(std::string const& name);

        // Extract the real parameter 'name' from object
    // Possible exceptions thrown: SwdInputValueException
    real_swd GetReal(std::string const& name);
    // ================================================================

    // Clear error signal in C/Fortran code in case of recovering from
    // advanced exception handling
    // Possible exceptions thrown: None
    void ExceptionClear();

};
#endif SWD_CPP_H_INCLUDED
```

6.2.2 SpectralWaveData.cpp

```
#include <string>
#include "SpectralWaveData.h"

/*
SpectralWaveData is a C++ class for the spectral-wave-data ocean wave model.

This implementation is based on establishing wrappers to the C-implementation.

Ver-3.0:
Coded by: Jens B. Helmers DNVGL,  2019.08.11

/**/

SpectralWaveData::SpectralWaveData(
    std::string file_swd,      // Name of actual SWD file
    real_swd x0, real_swd y0, // Parameters relating the SWD and application
    real_swd t0,               // coordinate system. beta is in degree.
    real_swd beta,
    real_swd rho,              // Density of water (only relevant for pressures)
    int nsumx, int nsumy,      // Number of applied spectral components. (negative=>␣
↪apply all)
    int impl,                  // Index to determine actual derived class
                               //    0 = Default
                               //   <0 = In-house and experimental implementations
                               //   >0 = Validated implementations available open␣
↪software
    int ipol,                  // Index to request actual temporal interpolation scheme
                               //    0 = Default (C^2 continous scheme)
                               //    1 = C^1 continous
                               //    2 = C^3 continous
    int norder,                // Expansion order to apply in kinematics for z>0
                               //    0 = Apply expansion order specified in swd␣
↪file(default)
                               //   <0 = Apply exp(kj z)
                               //   >0 = Apply expansion order = norder
    bool dc_bias               // Control application of zero - frequency bias present␣
↪in SWD file
                               //  false = Suppress contribution from zero frequency␣
↪amplitudes(default)
                               //  true = Apply zero frequency amplitudes from SWD␣
↪file.
) {
    obj = swd_api_allocate(file_swd.c_str(), x0, y0, t0, beta, rho, nsumx, nsumy,
                           impl, ipol, norder, dc_bias);
        if (swd_api_error_raised(obj)) {
                char *msg = swd_api_error_get_msg(obj);
                switch (swd_api_error_get_id(obj)) {
                case 1001:
                        throw SwdFileCantOpenException(msg);
                        break;
                case 1002:
                        throw SwdFileBinaryException(msg);
                        break;
                case 1003:
                        throw SwdFileDataException(msg);
```

(continues on next page)

(continued from previous page)

```
                        break;
                case 1004:
                        throw SwdInputValueException(msg);
                        break;
                case 1005:
                        throw SwdAllocateException(msg);
                        break;
                default:
                        throw SwdException(msg);
                        break;
                }
        }
}

SpectralWaveData::~SpectralWaveData() {
    swd_api_close(obj);
    //delete obj;
}

// ====================================================================
// Methods for evaluating kinematics...
// ====================================================================

// update current user time
void SpectralWaveData::UpdateTime(real_swd time)
{
    swd_api_update_time(obj, time);
        if (swd_api_error_raised(obj)) {
                throw SwdException(swd_api_error_get_msg(obj));
        }
}

// Calculate wave potential
real_swd SpectralWaveData::Phi(real_swd x, real_swd y, real_swd z)
{
    return swd_api_phi(obj, x, y, z);
}

// Calculate stream function
real_swd SpectralWaveData::Stream(real_swd x, real_swd y, real_swd z)
{
    return swd_api_stream(obj, x, y, z);
}

// Calculate time derivative of wave potential (earth fixed observer)
real_swd SpectralWaveData::DdtPhi(real_swd x, real_swd y, real_swd z)
{
    return swd_api_phi_t(obj, x, y, z);
}

// Calculate particle velocity
vector_swd SpectralWaveData::GradPhi(real_swd x, real_swd y, real_swd z)
{
    return swd_api_grad_phi(obj, x, y, z);
}

// 2nd order gradients of potential
```

(continues on next page)

(continued from previous page)

```
vector_2nd_phi_swd SpectralWaveData::GradPhi2nd(real_swd x, real_swd y, real_swd z)
{
    return swd_api_grad_phi_2nd(obj, x, y, z);
}

// Calculate Euler acceleration: grad(phi_t)
vector_swd SpectralWaveData::AccEuler(real_swd x, real_swd y, real_swd z)
{
    return swd_api_acc_euler(obj, x, y, z);
}

// Calculate particle acceleration
vector_swd SpectralWaveData::AccParticle(real_swd x, real_swd y, real_swd z)
{
    return swd_api_acc_particle(obj, x, y, z);
}

// Calculate wave elevation
real_swd SpectralWaveData::Elev(real_swd x, real_swd y)
{
    return swd_api_elev(obj, x, y);
}

// Calculate time derivative of wave elevation (Euler derivative, earth fixed)
real_swd SpectralWaveData::DdtElev(real_swd x, real_swd y)
{
    return swd_api_elev_t(obj, x, y);
}

// Calculate gradient of wave elevation (slopes)
vector_swd SpectralWaveData::GradElev(real_swd x, real_swd y)
{
    return swd_api_grad_elev(obj, x, y);
}

// 2nd order gradients of elevation
vector_2nd_elev_swd SpectralWaveData::GradElev2nd(real_swd x, real_swd y)
{
    return swd_api_grad_elev_2nd(obj, x, y);
}

// Calculate complete Bernoulli pressure
real_swd SpectralWaveData::Pressure(real_swd x, real_swd y, real_swd z)
{
    return swd_api_pressure(obj, x, y, z);
}

// Vertical distance from z=0 to sea floor (<0 if infinite)
real_swd SpectralWaveData::Bathymetry(real_swd x, real_swd y)
{
    return swd_api_bathymetry(obj, x, y);
}

// Unit normal vector of sea floor into the ocean at(x, y)
vector_swd SpectralWaveData::BathymetryNvec(real_swd x, real_swd y)
{
    return swd_api_bathymetry_nvec(obj, x, y);
```

(continues on next page)

(continued from previous page)

```
}

// For a specific location return a csv-file on how velocity, elevation
// and pressure converge as a function of number of spectral components
void SpectralWaveData::Convergence(real_swd x, real_swd y, real_swd z, std::string␣
↪csv)
{
    swd_api_convergence(obj, x, y, z, csv.c_str());
        if (swd_api_error_raised(obj)) {
                throw SwdException(swd_api_error_get_msg(obj));
        }
}

// To save storage for an interesting event you may create a new SWD file
// containing only the time steps within the time window [tmin, tmax].
// The name of the new file is defined by file_swd_new
void SpectralWaveData::Strip(real_swd tmin, real_swd tmax, std::string file_swd_new)
{
    swd_api_strip(obj, tmin, tmax, file_swd_new.c_str());
        if (swd_api_error_raised(obj)) {
                throw SwdException(swd_api_error_get_msg(obj));
        }
}

// ====================================================================
//  Provide parameters from the swd-file:

// Extract the character parameter 'name' from object
std::string SpectralWaveData::GetChr(std::string const& name)
{
    return swd_api_get_chr(obj, name.c_str());
}

// Extract the int parameter 'name' from object
int SpectralWaveData::GetInt(std::string const& name)
{
    return swd_api_get_int(obj, name.c_str());
}

// Extract the int parameter 'name' from object
bool SpectralWaveData::GetBool(std::string const& name)
{
        return swd_api_get_bool(obj, name.c_str());
}

// Extract the real parameter 'name' from object
real_swd SpectralWaveData::GetReal(std::string const& name)
{
    return swd_api_get_real(obj, name.c_str());
}
// ====================================================================

// Clear error flag in C/Fortran implementation.
// Only applied in case of advanced exception handling. (recovery)
void SpectralWaveData::ExceptionClear()
{
        swd_api_error_clear(obj);
```

(continues on next page)

(continued from previous page)

```
}
```

6.2.3 Implementation

In the current version the implementation of the class `SpectralWaveData` is obtained by wrapping the C-implementation as done in the file *SpectralWaveData.cpp*.

6.3 C

C does not support object-oriented-programming. However, an object based approach is applied. All functions apply the prefix naming convention `swd_api_*`. The API is defined in the header file *spectral_wave_data.h*.

The data types `real_swd`, `vector_swd`, `vector_2nd_phi_swd` and `vector_2nd_elev_swd` are also defined in the C-header file.

If your application apply float (not double) for the SWD-API you need to:

1. Set the macro `SWD_API_FLOAT` before the header file is included.
2. The fortran library should be compiled with *kind_swd_c = c_float* in the file `kind_values.f90`.

By default it is assumed that the C-interface apply *double*. Hence *kind_swd_c = c_double* in the file `kind_values.f90` and the macro `SWD_API_FLOAT` should be unset.

6.3.1 spectral_wave_data.h

```
#ifndef SWD_API_H_INCLUDED
#define SWD_API_H_INCLUDED
/*
The C interface for the spectral-wave-data ocean wave model.

For all functions we apply the naming convention: swd_api_*

Ver-3.0:
Coded by: Jens B. Helmers DNVGL,  2019.07.31

*/

#ifdef __cplusplus
extern "C" {
#endif

#ifndef __cplusplus
#include <stdbool.h>
#endif

// Floating point model for interfacing with spectral_wave_data
#ifdef SWD_API_FLOAT
   typedef float real_swd;
#else
   typedef double real_swd;
#endif
```

(continues on next page)

(continued from previous page)

```
typedef struct {
    real_swd x;
    real_swd y;
    real_swd z;
} vector_swd;

typedef struct {
    real_swd xx;   // Direction of derivatives
    real_swd xy;
    real_swd xz;
    real_swd yy;
    real_swd yz;
    real_swd zz;
} vector_2nd_phi_swd;

typedef struct {
    real_swd xx;   // Direction of derivatives
    real_swd xy;
    real_swd yy;
} vector_2nd_elev_swd;


// constructor
void *swd_api_allocate(const char *file_swd, real_swd x0, real_swd y0,
                       real_swd t0, real_swd beta, real_swd rho,
                       int nsumx, int nsumy, int impl, int ipol,
                       int norder, bool dc_bias);
    /*
    file_swd:          Name of actual swd file

    x0, y0, t0, beta:  Relation between SWD and application coordinates.
                       beta in degree.

    rho:               Density of water(applied for pressure calculations)

    nsumx, nsumy       Number of spectral components to apply (<0: apply all)

    impl               Index to determine actual derived class to apply
                       0 = Automatic selection of best class (default)
                      <0 = In-house and experimental implementations
                      >0 = Validated implementations available open software

    ipol               Index to request actual temporal interpolation scheme
                       0 = C^2 continous scheme (default)
                       1 = C^1 continous
                       2 = C^3 continous

    norder             Expansion order to apply in kinematics for z>0
                       0 = Apply expansion order specified in swd file (default)
                      <0 = Apply exp(kj z)
                      >0 = Apply expansion order = norder

    dc_bias            Control application of zero-frequency bias present in SWD file␣
↪
                       false = Suppress contribution from zero frequency amplitudes␣
↪(default)
```

(continues on next page)

(continued from previous page)

```
                        true  = Apply zero frequency amplitudes from SWD file.

    Error signals has been raised if the function 'swd_api_error_raised()' return␣
↪true.
    That function should be called after this constructor to check if successful␣
↪creation.
    The actual error codes is returned using the function 'swd_api_error_get_id()':

    1001: SWD file not able to open
    1002: SWD file has wrong binary convention (not float/little endian)
    1003: SWD file data read error
    1004: invalid input parameters
    1005: Not able to allocate memory

   /**/

// destructor
void swd_api_close(void *swd);


// ======================================================================
//  Field calculations at current application time:
// ======================================================================

// apply current application time
// Eventual error signals (ref constructor): 1003, 1004
void swd_api_update_time(void *swd, real_swd time);

// wave potential
// Eventual error signals (ref constructor): None
real_swd swd_api_phi(void *swd, real_swd x, real_swd y, real_swd z);

// stream function
// Eventual error signals (ref constructor): None
real_swd swd_api_stream(void *swd, real_swd x, real_swd y, real_swd z);

// time derivative of wave potential (Euler derivative, earth fixed)
// Eventual error signals (ref constructor): None
real_swd swd_api_phi_t(void *swd, real_swd x, real_swd y, real_swd z);

// particle velocity
// Eventual error signals (ref constructor): None
vector_swd swd_api_grad_phi(void *swd, real_swd x, real_swd y, real_swd z);

// 2nd order gradients of potential
// Eventual error signals (ref constructor): None
vector_2nd_phi_swd swd_api_grad_phi_2nd(void *swd, real_swd x, real_swd y, real_swd␣
↪z);

// Local (Euler) acceleration
// Eventual error signals (ref constructor): None
vector_swd swd_api_acc_euler(void *swd, real_swd x, real_swd y, real_swd z);

// Particle acceleration
// Eventual error signals (ref constructor): None
vector_swd swd_api_acc_particle(void *swd, real_swd x, real_swd y, real_swd z);
```

(continues on next page)

(continued from previous page)

```
// wave elevation
// Eventual error signals (ref constructor): None
real_swd swd_api_elev(void *swd, real_swd x, real_swd y);

// Local time derivative of wave elevation
// Eventual error signals (ref constructor): None
real_swd swd_api_elev_t(void *swd, real_swd x, real_swd y);

// Gradient of wave elevation (slopes)
// Eventual error signals (ref constructor): None
vector_swd swd_api_grad_elev(void *swd, real_swd x, real_swd y);

//  2nd order gradients of elevation
// Eventual error signals (ref constructor): None
vector_2nd_elev_swd swd_api_grad_elev_2nd(void *swd, real_swd x, real_swd y);

// Complete Bernoulli pressure
// Eventual error signals (ref constructor): None
real_swd swd_api_pressure(void *swd, real_swd x, real_swd y, real_swd z);

// Vertical distance from z=0 to sea floor (<0 if infinite)
// Eventual error signals (ref constructor): None
real_swd swd_api_bathymetry(void *swd, real_swd x, real_swd y);

// Unit normal vector of sea floor into the ocean at(x, y)
// Eventual error signals (ref constructor): None
vector_swd swd_api_bathymetry_nvec(void *swd, real_swd x, real_swd y);

// For a specific location return a csv-file on how velocity, elevation
// and pressure converge as a function of number of spectral components.
// Eventual error signals (ref constructor): 1001
void swd_api_convergence(void *swd, real_swd x, real_swd y, real_swd z, const char␣
↪*csv);

// To save storage for an interesting event you may create a new SWD file
// containing only the time steps within the time window [tmin, tmax].
// The name of the new SWD file is defined by file_swd_new
// Eventual error signals (ref constructor): 1001, 1003
void swd_api_strip(void *swd, real_swd tmin, real_swd tmax, const char *file_swd_new);

// ===================================================================
//  Provide parameters from the swd-file:

// Extract the character parameter 'name'
// Eventual error signals (ref constructor): 1004
// **NOTE**: This function is not thread-safe. Make a copy of the
// result if you need to store it before calling this function again.
char *swd_api_get_chr(void *swd, const char *name);

// Extract the int parameter 'name'
// Eventual error signals (ref constructor): 1004
int swd_api_get_int(void *swd, const char *name);

// Extract the bool parameter 'name'
// Eventual error signals (ref constructor): 1004
bool swd_api_get_bool(void *swd, const char *name);
```

(continues on next page)

(continued from previous page)

```
// Extract the real parameter 'name'
// Eventual error signals (ref constructor): 1004
real_swd swd_api_get_real(void *swd, const char *name);

// ================================================================
//  Provide error handling:

// Check if SWD object has signaled an error
bool swd_api_error_raised(void *swd);

// Extract error id from SWD obhect
int swd_api_error_get_id(void *swd);

// Extract error message from SWD object
char *swd_api_error_get_msg(void *swd);

// Clear the error flag from SWD object
void *swd_api_error_clear(void *swd);

// ================================================================

#ifdef __cplusplus
}
#endif

#endif
```

6.3.2 Implementation

In the current version the implementation of these functions are obtained from the fortran implementation using the standard ISO_C_BINDINGS defined in *spectral_wave_data_c.f90*.

spectral_wave_data_c.f90

```
module spectral_wave_data_c_def

use, intrinsic :: iso_fortran_env, only: output_unit
use, intrinsic :: iso_c_binding,   only: c_char, c_int, c_double, c_ptr, c_loc, &
                                         c_f_pointer, c_null_char, c_bool

use kind_values, only: wp => kind_swd_interface, c_wp => kind_swd_c

use spectral_wave_data_def,           only: spectral_wave_data
use spectral_wave_data_allocate_def,  only: spectral_wave_data_allocate

implicit none
private

! This module provides a standard C-compatible interface to the
! fortran spectral_wave_data class.
!
! Written by Jens Bloch Helmers, July, 20. 2019
!
```

(continues on next page)

(continued from previous page)

```
!------------------------------------------------------------------------------

!##############################################################################
!
!              B E G I N    P U B L I C    Q U A N T I T I E S
!
!------------------------------------------------------------------------------
!
public :: spectral_wave_data_c  ! Target for C-pointer. Fortran SWD object is in␣
↪target.
!
! C-compatible structs
!
public :: vector        ! Vector [x,y,z]
public :: vector_phi_2nd ! Vector [xx,xy,xz,yy,yz,zz]
public :: vector_elev_2nd ! Vector [xx,xy,yy]
!
! Public fortran functions with pure C-interface
! All C-compatible names are defined by bind(c, name=...) in the function definitions
!
public :: constructor   ! Construct a C-pointer to a new specific SWD object
public :: update_time   ! Set current user time
public :: phi           ! Wave potential
public :: stream        ! Stream function
public :: phi_t         ! d(potential)/dt (Euler derivative)
public :: grad_phi      ! Particle velocity
public :: grad_phi_2nd  ! 2nd order gradients of potential
public :: acc_euler     ! Euler acceleration (grad(phi_t))
public :: acc_particle  ! Particle acceleration
public :: elev          ! Surface elevation
public :: elev_t        ! d(surface elevation)/dt (Euler derivative)
public :: grad_elev     ! Gradient of surface elevation
public :: grad_elev_2nd ! 2nd order gradients of elevation
public :: pressure      ! Fully nonlinear Bernoulli pressure
public :: bathymetry    ! Vertical distance from z=0 to sea floor (<0 if infinite)
public :: bathymetry_nvec ! Unit normal vector of sea floor into the ocean
public :: convergence   ! For a specific (t,x,y,z) return a csv-file on how velocity,␣
↪elevation
                        ! and pressure converge as a function of number of spectral␣
↪components
public :: strip         ! Create an new SWD file based on a time window of current␣
↪SWD file.
public :: get_int       ! Extract a specified int parameter from the swd file.
public :: get_bool      ! Extract a specified bool parameter from the swd file.
public :: get_real      ! Extract a specified real parameter from the swd file.
public :: get_chr       ! Extract a specified char parameter from the swd file.
public :: error_raised  ! Return true if an error has been raised, otherwise false
public :: error_get_id  ! Return error id
public :: error_get_msg ! Return error message
public :: error_clear   ! Clear the error flag
public :: close         ! Destructor
!------------------------------------------------------------------------------
!
!               E N D    P U B L I C    Q U A N T I T I E S
!
!##############################################################################
```

(continues on next page)

(continued from previous page)

```
type :: spectral_wave_data_c
    class(spectral_wave_data), allocatable :: obj
end type spectral_wave_data_c

type, bind(c) :: vector
    real(c_wp) :: x, y, z
end type vector

type, bind(c) :: vector_phi_2nd
    real(c_wp) :: xx, xy, xz, yy, yz, zz
end type vector_phi_2nd

type, bind(c) :: vector_elev_2nd
    real(c_wp) :: xx, xy, yy
end type vector_elev_2nd

contains

!=============================================================================

function constructor(file_swd, x0, y0, t0, beta, rho, nsumx, nsumy, impl, &
                     ipol, norder, dc_bias) bind(c, name='swd_api_allocate')
!-----------------------------------------------------------------------------
! Provides a C pointer to an object containing the actual Fortran SWD object.
!-----------------------------------------------------------------------------
type(c_ptr) :: constructor  ! Pointer to construct

! Name of swd file
character(c_char), intent(in) :: file_swd(*)

! Relation between SWD and application coordinates (beta is in degree)
real(c_wp), value, intent(in) :: x0, y0, t0, beta

! Density of water (applied for pressure calculations)
real(c_wp), value, intent(in) :: rho

! Number of applied spectral comp. in x and y-dir.
! If negative: apply all spectral components from the SWD file
integer(c_int), value, intent(in) :: nsumx
integer(c_int), value, intent(in) :: nsumy

! Index to determine actual derived class
!    0 = Default based on content of SWD file
!   <0 = In-house and experimental implementations
!   >0 = Validated implementations available open software
integer(c_int), value, intent(in) :: impl

! Index to request actual temporal interpolation scheme
!   0 = C^2 continous scheme (default)
!   1 = C^1 continous
!   2 = C^3 continous
integer(c_int), value, intent(in) :: ipol

! Expansion order to apply in kinematics for z>0
!   0  = Apply expansion order specified on swd file (default)
!   <0 = Apply exp(kj z)
!   >0 = Apply expansion order = norder
```

(continues on next page)

(continued from previous page)

```
integer(c_int), value, intent(in) :: norder

! Control application of zero-frequency bias present in SWD file
!  False = Suppress contribution from zero frequency amplitudes (default)
!  True  = Apply zero frequency amplitudes from SWD file.
logical(c_bool), value, intent(in):: dc_bias
!------------------------------------------------------------------------------

integer :: impl_f, nsumx_f, nsumy_f, ipol_f, norder_f
logical :: dc_bias_f
type(spectral_wave_data_c), pointer :: swdc
real(wp) :: x0_f, y0_f, t0_f, beta_f, rho_f
character(len=:), allocatable :: file_swd_f

! Convert to fortran types
file_swd_f = str_c2f(file_swd)
x0_f = x0
y0_f = y0
t0_f = t0
beta_f = beta
rho_f = rho
nsumx_f = nsumx
nsumy_f = nsumy
impl_f = impl
ipol_f = ipol
norder_f = norder
dc_bias_f = dc_bias
!
allocate(swdc)
call spectral_wave_data_allocate(swdc % obj, file_swd_f, x0_f, y0_f,    &
                                 t0_f, beta_f, rho_f, nsumx_f, nsumy_f, &
                                 impl_f, ipol_f, norder_f, dc_bias_f)

constructor = c_loc(swdc)
!
end function constructor

!==============================================================================

subroutine update_time(this, time) bind(c, name='swd_api_update_time')
type(c_ptr), value             :: this  ! Actual spectral_wave_data_c object
real(c_wp),  value, intent(in) :: time  ! Current application time
!
type(spectral_wave_data_c), pointer :: swd
real(wp) :: time_f
time_f = time
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
call swd % obj % update_time(time_f)
!
end subroutine update_time

!==============================================================================

function phi(this, x, y, z) bind(c, name='swd_api_phi')
type(c_ptr), value             :: this  ! Actual spectral_wave_data_c object
real(c_wp),  value, intent(in) :: x,y,z ! Position application program
```

(continues on next page)

(continued from previous page)

```
real(c_wp)                      :: phi    ! Potential at (x,y,z)
!
type(spectral_wave_data_c), pointer :: swd
real(wp) :: x_f, y_f, z_f
x_f = x
y_f = y
z_f = z
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
phi = swd % obj % phi(x_f, y_f, z_f)
!
end function phi

!================================================================================

function stream(this, x, y, z) bind(c, name='swd_api_stream')
type(c_ptr), value             :: this   ! Actual spectral_wave_data_c object
real(c_wp),  value, intent(in) :: x,y,z  ! Position application program
real(c_wp)                     :: stream ! Stream function at at (x,y,z)
!
type(spectral_wave_data_c), pointer :: swd
real(wp) :: x_f, y_f, z_f
x_f = x
y_f = y
z_f = z
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
stream = swd % obj % stream(x_f, y_f, z_f)
!
end function stream

!================================================================================

function phi_t(this, x, y, z) bind(c, name='swd_api_phi_t')
type(c_ptr), value             :: this  ! Actual spectral_wave_data_c object
real(c_wp),  value, intent(in) :: x,y,z ! Position application program
real(c_wp)                     :: phi_t ! d(potential)/dt (Euler) at (x,y,z)
!
type(spectral_wave_data_c), pointer :: swd
real(wp) :: x_f, y_f, z_f
x_f = x
y_f = y
z_f = z
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
phi_t = swd % obj % phi_t(x_f, y_f, z_f)
!
end function phi_t

!================================================================================

function grad_phi(this, x, y, z) bind(c, name='swd_api_grad_phi')
type(c_ptr), value             :: this     ! Actual spectral_wave_data_c object
real(c_wp),  value, intent(in) :: x,y,z    ! Position application program
type(vector)                   :: grad_phi ! Particle velocity at (x,y,z)
!
type(spectral_wave_data_c), pointer :: swd
```

(continues on next page)

(continued from previous page)

```
real(wp) :: x_f, y_f, z_f, vec(3)
x_f = x
y_f = y
z_f = z
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
vec = swd % obj % grad_phi(x_f, y_f, z_f)
grad_phi = vector(vec(1), vec(2), vec(3))
!
end function grad_phi

!=============================================================================

function grad_phi_2nd(this, x, y, z) bind(c, name='swd_api_grad_phi_2nd')
type(c_ptr), value              :: this     ! Actual spectral_wave_data_c object
real(c_wp),  value, intent(in) :: x,y,z    ! Position application program
type(vector_phi_2nd)            :: grad_phi_2nd ! [xx,xy,xz,yy,yz,zz] at (x,y,z)
!
type(spectral_wave_data_c), pointer :: swd
real(wp) :: x_f, y_f, z_f, vec(6)
x_f = x
y_f = y
z_f = z
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
vec = swd % obj % grad_phi_2nd(x_f, y_f, z_f)
grad_phi_2nd = vector_phi_2nd(vec(1), vec(2), vec(3), vec(4), vec(5), vec(6))
!
end function grad_phi_2nd

!=============================================================================

function acc_euler(this, x, y, z) bind(c, name='swd_api_acc_euler')
!-----------------------------------------------------------------------------
! Actual spectral_wave_data_c object
type(c_ptr), value :: this

! Position application program
real(c_wp), value, intent(in) :: x,y,z

! Euler acceleration (grad(phi_t)) at (x,y,z) for current time)
type(vector) :: acc_euler
!-----------------------------------------------------------------------------
!
type(spectral_wave_data_c), pointer :: swd
real(wp) :: x_f, y_f, z_f, acc(3)
x_f = x
y_f = y
z_f = z
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
acc = swd % obj % acc_euler(x_f, y_f, z_f)
acc_euler = vector(acc(1), acc(2), acc(3))
!
end function acc_euler

!=============================================================================
```

(continues on next page)

(continued from previous page)

```
function acc_particle(this, x, y, z) bind(c, name='swd_api_acc_particle')
!-------------------------------------------------------------------------
! Actual spectral_wave_data_c object
type(c_ptr), value :: this

! Position application program
real(c_wp), value, intent(in) :: x,y,z

! Euler acceleration (grad(phi_t)) at (x,y,z) for current time)
type(vector) :: acc_particle
!-------------------------------------------------------------------------
!
type(spectral_wave_data_c), pointer :: swd
real(wp) :: x_f, y_f, z_f, acc(3)
x_f = x
y_f = y
z_f = z
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
acc = swd % obj % acc_particle(x_f, y_f, z_f)
acc_particle = vector(acc(1), acc(2), acc(3))
!
end function acc_particle

!=========================================================================

function elev(this, x, y) bind(c, name='swd_api_elev')
type(c_ptr), value             :: this  ! Actual spectral_wave_data_c object
real(c_wp),  value, intent(in) :: x,y   ! Position application program
real(c_wp)                     :: elev  ! Wave elevation at (x,y)
!
type(spectral_wave_data_c), pointer :: swd
real(wp) :: x_f, y_f
x_f = x
y_f = y
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
elev = swd % obj % elev(x_f, y_f)
!
end function elev

!=========================================================================

function elev_t(this, x, y) bind(c, name='swd_api_elev_t')
type(c_ptr), value             :: this   ! Actual spectral_wave_data_c object
real(c_wp),  value, intent(in) :: x,y    ! Position application program
real(c_wp)                     :: elev_t ! d(surface elevation)/dt (Euler derivative)
                                         ! at (x,y) for current time
!
type(spectral_wave_data_c), pointer :: swd
real(wp) :: x_f, y_f
x_f = x
y_f = y
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
elev_t = swd % obj % elev_t(x_f, y_f)
```

(continues on next page)

(continued from previous page)

```
!
end function elev_t

!================================================================================

function grad_elev(this, x, y) bind(c, name='swd_api_grad_elev')
type(c_ptr), value             :: this      ! Actual spectral_wave_data_c object
real(c_wp),  value, intent(in) :: x,y       ! Position application program
type(vector)                   :: grad_elev ! Gradient of surface elevation at (x,y)␣
↪for current time
!
type(spectral_wave_data_c), pointer :: swd
real(wp) :: x_f, y_f, grad(3)
x_f = x
y_f = y
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
grad = swd % obj % grad_elev(x_f, y_f)
grad_elev = vector(grad(1), grad(2), grad(3))
!
end function grad_elev

!================================================================================

function grad_elev_2nd(this, x, y) bind(c, name='swd_api_grad_elev_2nd')
type(c_ptr), value             :: this      ! Actual spectral_wave_data_c object
real(c_wp),  value, intent(in) :: x,y       ! Position application program
type(vector_elev_2nd)          :: grad_elev_2nd ! Gradient of surface elevation at (x,
↪y) for current time
!
type(spectral_wave_data_c), pointer :: swd
real(wp) :: x_f, y_f, grad(3)
x_f = x
y_f = y
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
grad = swd % obj % grad_elev_2nd(x_f, y_f)
grad_elev_2nd = vector_elev_2nd(grad(1), grad(2), grad(3))
!
end function grad_elev_2nd

!================================================================================

function pressure(this, x, y, z) bind(c, name='swd_api_pressure')
type(c_ptr), value             :: this     ! Actual spectral_wave_data_c object
real(c_wp),  value, intent(in) :: x,y,z    ! Position application program
real(c_wp)                     :: pressure ! Fully nonlinear Bernoulli pressure
                                           ! at (x,y,z) for current time
!
type(spectral_wave_data_c), pointer :: swd
real(wp) :: x_f, y_f, z_f
x_f = x
y_f = y
z_f = z
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
pressure = swd % obj % pressure(x_f, y_f, z_f)
```

(continues on next page)

(continued from previous page)

```
!
end function pressure

!=============================================================================

function bathymetry(this, x, y) bind(c, name='swd_api_bathymetry')
type(c_ptr), value             :: this       ! Actual spectral_wave_data_c object
real(c_wp),  value, intent(in) :: x,y        ! Position application program
real(c_wp)                     :: bathymetry ! Local water depth at
                                               ! (x,y) for current time
!
type(spectral_wave_data_c), pointer :: swd
real(wp) :: x_f, y_f
x_f = x
y_f = y
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
bathymetry = swd % obj % bathymetry(x_f, y_f)
!
end function bathymetry

!=============================================================================

function bathymetry_nvec(this, x, y) bind(c, name='swd_api_bathymetry_nvec')
type(c_ptr), value :: this  ! Actual spectral_wave_data_c object
real(c_wp), value, intent(in) :: x,y   ! Position application program
type(vector) :: bathymetry_nvec ! Unit normal vector of sea floor into the ocean at(x,
↪ y)
!
type(spectral_wave_data_c), pointer :: swd
real(wp) :: x_f, y_f, res(3)
x_f = x
y_f = y
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
res = swd % obj % bathymetry_nvec(x_f, y_f)
bathymetry_nvec = vector(res(1), res(2), res(3))
!
end function bathymetry_nvec

!=============================================================================

subroutine convergence(this, x, y, z, csv) bind(c, name='swd_api_convergence')
!-----------------------------------------------------------------------------
! For a specific(x, y, z) at this t, return a CSV file on how particle velocity,
! elevation and pressure converge as a function of number of spectral components
!-----------------------------------------------------------------------------
type(c_ptr), value  :: this ! Actual spectral_wave_data_c object
real(c_wp),  value, intent(in) :: x,y,z    ! Position application program
character(c_char),  intent(in) :: csv(*)   ! Name of output CSV file
!-----------------------------------------------------------------------------
!
type(spectral_wave_data_c), pointer :: swd
character(len=:), allocatable :: csv_f
real(wp) :: x_f, y_f, z_f
x_f = x
y_f = y
```

(continues on next page)

(continued from previous page)

```
z_f = z
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
csv_f = str_c2f(csv)  ! Convert to fortran string
!
call swd % obj % convergence(x_f, y_f, z_f, csv_f)
!
end subroutine convergence

!================================================================================

subroutine strip(this, tmin, tmax, swd_file) bind(c, name='swd_api_strip')
!--------------------------------------------------------------------------------
! Create a new SWD file containing only the time steps within
! the range [tmin, tmax]
!--------------------------------------------------------------------------------
type(c_ptr), value :: this ! Actual spectral_wave_data_c object
real(c_wp),  value, intent(in) :: tmin, tmax ! Time window
character(c_char),  intent(in) :: swd_file(*)   ! Name of new SWD file
!--------------------------------------------------------------------------------
!
type(spectral_wave_data_c), pointer :: swd
real(wp) :: tmin_f, tmax_f
character(len=:), allocatable :: swd_file_f
!
swd_file_f = str_c2f(swd_file)  ! Convert to fortran string
tmin_f = tmin
tmax_f = tmax
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
call swd % obj % strip(tmin_f, tmax_f, swd_file_f)
!
end subroutine strip

!================================================================================

function get_int(this, name) bind(c, name='swd_api_get_int')
!--------------------------------------------------------------------------------
! Extract 'name' from the SWD file assuming it is an int
! Error signal raised if 'name' is not sound (error_get_id()==1004)
!--------------------------------------------------------------------------------
type(c_ptr), value :: this ! Actual spectral_wave_data_c object
integer(c_int) :: get_int  ! The value of the requested parameter
character(c_char), intent(in) :: name(*)  ! Name of character variable
!--------------------------------------------------------------------------------
!
type(spectral_wave_data_c), pointer :: swd
character(len=:), allocatable :: name_f
!
name_f = str_c2f(name)  ! Convert to fortran string
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
get_int = swd % obj % get_int(name_f)
!
end function get_int

!================================================================================
```

(continues on next page)

(continued from previous page)

```
function get_bool(this, name) bind(c, name='swd_api_get_bool')
!------------------------------------------------------------------------
! Extract 'name' from the SWD file assuming it is a bool
! Error signal raised if 'name' is not sound (error_get_id()==1004)
!------------------------------------------------------------------------
type(c_ptr), value :: this ! Actual spectral_wave_data_c object
logical(c_bool) :: get_bool ! The value of the requested parameter
character(c_char), intent(in) :: name(*)  ! Name of character variable
!------------------------------------------------------------------------
!
type(spectral_wave_data_c), pointer :: swd
character(len=:), allocatable :: name_f
!
name_f = str_c2f(name)  ! Convert to fortran string
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
get_bool = swd % obj % get_logical(name_f)
!
end function get_bool

!============================================================================

function get_real(this, name) bind(c, name='swd_api_get_real')
!------------------------------------------------------------------------
! Extract 'name' from the SWD file assuming it is a float
! Error signal raised if 'name' is not sound (error_get_id()==1004)
!------------------------------------------------------------------------
type(c_ptr), value :: this ! Actual spectral_wave_data_c object
real(c_wp) :: get_real  ! The value of the requested parameter
character(c_char), intent(in) :: name(*)  ! Name of character variable
!------------------------------------------------------------------------
!
type(spectral_wave_data_c), pointer :: swd
character(len=:), allocatable :: name_f
!
name_f = str_c2f(name)  ! Convert to fortran string
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
get_real = swd % obj % get_real(name_f)
!
end function get_real

!============================================================================

function get_chr(this, name) bind(c, name='swd_api_get_chr')
!------------------------------------------------------------------------
! Extract 'name' from the SWD file assuming it is a character array
! Error signal raised if 'name' is not sound (error_get_id()==1004)
!------------------------------------------------------------------------
type(c_ptr), value  :: this     ! Actual spectral_wave_data_c object
type(c_ptr)         :: get_chr  ! The value of the requested parameter
character(c_char),  intent(in) :: name(*)  ! Name of character variable
!------------------------------------------------------------------------
!
type(spectral_wave_data_c), pointer :: swd
character(kind=c_char, len=:), allocatable, save, target :: value_c
character(len=:), allocatable :: name_f, value_f
```

(continues on next page)

(continued from previous page)

```
!
name_f = str_c2f(name)  ! Convert to fortran string
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
value_f = swd % obj % get_chr(name_f)
value_c = trim(value_f) // c_null_char
get_chr = c_loc(value_c)
!
end function get_chr

!================================================================================

function error_raised(this) bind(c, name='swd_api_error_raised')
!--------------------------------------------------------------------------------
! Check if an error has been raised in the SWD object
!--------------------------------------------------------------------------------
type(c_ptr), value :: this ! Actual spectral_wave_data_c object
logical(c_bool) :: error_raised  ! True if an error is raised (false if not)
!--------------------------------------------------------------------------------
!
type(spectral_wave_data_c), pointer :: swd
!
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
error_raised = swd % obj % error % raised()
!
end function error_raised

!================================================================================

function error_get_id(this) bind(c, name='swd_api_error_get_id')
!--------------------------------------------------------------------------------
! Extract error id from SWD object
!--------------------------------------------------------------------------------
type(c_ptr), value :: this ! Actual spectral_wave_data_c object
integer(c_int) :: error_get_id  ! The error id
!--------------------------------------------------------------------------------
!
type(spectral_wave_data_c), pointer :: swd
!
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
error_get_id = swd % obj % error % get_id()
!
end function error_get_id

!================================================================================

function error_get_msg(this) bind(c, name='swd_api_error_get_msg')
!--------------------------------------------------------------------------------
! Extract error message from SWD object
!--------------------------------------------------------------------------------
type(c_ptr), value  :: this     ! Actual spectral_wave_data_c object
type(c_ptr)         :: error_get_msg  ! The error message
!--------------------------------------------------------------------------------
!
type(spectral_wave_data_c), pointer :: swd
```

(continues on next page)

(continued from previous page)

```
character(kind=c_char, len=:), allocatable, save, target :: value_c
character(len=:), allocatable :: value_f
!
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
value_f = swd % obj % error % get_msg()
value_c = trim(value_f) // c_null_char
error_get_msg = c_loc(value_c)
!
end function error_get_msg

!=============================================================================

subroutine error_clear(this) bind(c, name='swd_api_error_clear')
type(c_ptr), value :: this  ! Actual spectral_wave_data_c object
!
type(spectral_wave_data_c), pointer :: swd
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
call swd % obj % error % clear()
!
end subroutine error_clear

!=============================================================================

subroutine close(this) bind(c, name='swd_api_close')
type(c_ptr), value :: this  ! Actual spectral_wave_data_c object to destroy
!
type(spectral_wave_data_c), pointer :: swd
!
call c_f_pointer(this, swd) ! Find corresponding Fortran object (swd)
!
call swd % obj % close()
deallocate(swd % obj)
!
end subroutine close

!=============================================================================

function str_c2f(c_string) result (f_string)     ! Local helper function
character(c_char), intent(in) :: c_string(*)     ! Classical array of C-characters
character(len=:), allocatable :: f_string        ! Corresponding Fortran string
!
integer :: i, n
i = 0
do
    i = i + 1
    if (c_string(i)==c_null_char) exit
    if (i>1000000) then
        ! Seems that c_string is not null terminated!"...
        i = 20 ! Make it short and sweet to indicate bug...
    end if
end do
n = i - 1
allocate(character(len=n) :: f_string)
do i = 1, n
    f_string(i:i) = c_string(i)
```

(continues on next page)

(continued from previous page)

```
end do
!
end function str_c2f

!=============================================================================

end module spectral_wave_data_c_def
```

6.4 Fortran-2008

Using standard ISO Fortran-2008 the name of the abstract base class is `spectral_wave_data` which is available from the module `spectral_wave_data_def` defined in *spectral_wave_data.f90*. The abstract interface block in this module defines the signature for all applied methods.

The constructor for allocating and initializing a proper specialized class is defined in *spectral_wave_data_allocate.f90*. All constructor arguments are explained in the header of this constructor.

```
program my_application

use spectral_wave_data_def,   only: spectral_wave_data
use spectral_wave_data_allocate_def,   only: spectral_wave_data_allocate
...
class(spectral_wave_data), allocatable :: swd   ! Type is not known at this stage
...
file_swd = 'my_wave.swd'    ! File based on any supported spectral formulation
...
! Allocate and initialize the swd using the default implementation for the
! actual spectral formulation as defined in the swd file...
call spectral_wave_data_allocate(swd, file_swd, x0, y0, t0, beta)
if (swd % error % raised()) then
    print*, swd % error % get_msg()
    stop
end if

t = 0.0_wp; dt = 0.1_wp; tmax = swd % get_real('tmax')
do
   if (t > tmax) exit
   call swd % update_time(t)

  ! apply generic swd % methods() according to the interface defined in spectral_wave_
↪data.f90

  t = t + dt
end do
call swd % close()

end program my_application
```

Application developers may modify the `SELECT CASE` block in *spectral_wave_data_allocate.f90* to include support for new implementations derived from the base class.

6.4.1 spectral_wave_data.f90

```
module spectral_wave_data_def

use, intrinsic :: iso_fortran_env, only: int64

use kind_values, only: knd => kind_swd_interface, wp => kind_swd_internal

use spectral_wave_data_error, only: swd_error

implicit none
private

! This module provides the abstract base class for spectral_wave_data_X objects.
!
! Written by Jens Bloch Helmers, August, 2. 2019
!
!------------------------------------------------------------------------------

!##############################################################################
!
!             B E G I N    P U B L I C    Q U A N T I T I E S
!
!------------------------------------------------------------------------------
!
public :: spectral_wave_data
!
!------------------------------------------------------------------------------
!
!               E N D    P U B L I C    Q U A N T I T I E S
!
!##############################################################################

type, abstract :: spectral_wave_data
    ! Common attributes for all shape classes
    character(len=30)  :: prog ! Name of the program who created this swd file␣
↪including version.
    character(len=20)  :: date ! Date and time this swd file was created
    character(len=200) :: file ! Name of swd file
    integer            :: unit ! Unit number associated with swd file
    integer            :: fmt  ! Code to identify format of swd file.
    integer            :: shp  ! Index of actual spectral shape class
    integer            :: amp  ! Index of which spectral amplitudes are available
    character(len=:), allocatable :: cid  ! Identification text in swd file
    integer            :: nstrip ! Number of initial time steps removed from original␣
↪simulation
    integer            :: nsteps ! Total number of time steps in swd file.
    integer            :: order  ! Order of perturbation (<0 if fully nonlinear)␣
↪applied in wave generator
    integer            :: norder  ! Expansion order to apply in kinematics for z>0
                                  ! <0: apply exp(kj z)
                                  ! 0:  apply expansion order specified on swd file
                                  ! >0: apply expansion order = norder
    integer            :: ipol ! Index defining the temporal interpolation scheme
    real(wp)           :: dt   ! Constant time step in swd file
    real(wp)           :: t0   ! Input seed for time (t0>=0)
    real(wp)           :: x0,y0 ! Input seed for spatial location
```

(continues on next page)

(continued from previous page)

```
    real(wp)            :: tswd ! Current swd time
    real(wp)            :: grav ! Acceleration of gravity
    real(wp)            :: lscale ! Number of length units in wave generator per meter.
    real(wp)            :: rho  ! Density of water
    real(wp)            :: cbeta ! cos(beta), beta=angle between swd and application x-
↪axis
    real(wp)            :: sbeta ! sin(beta), beta=angle between swd and application x-
↪axis
    real(wp)            :: tmax  ! Maximum allowed simulation time (user system)
    integer             :: size_complex ! On most systems size_complex=8 for c_float␣
↪based numbers
    integer             :: size_step ! Fortran storage size per time step
    integer(int64)      :: ipos0 ! File postion where temporal functions starts
    logical             :: eof  ! End-of-file detected for SWD file
    logical             :: dc_bias ! True: apply zero frequency amplitudes from SWD␣
↪file.
                                   ! False: Suppress contribution from zero frequency␣
↪amplitudes (Default)
    type(swd_error)     :: error   ! Abort free error handler
contains
    procedure(update_time), deferred :: update_time       ! Obtain spectral data for␣
↪current time
    procedure(phi),         deferred :: phi               ! Calculate potential at␣
↪location for current time
    procedure(stream),      deferred :: stream            ! Calculate stream function
    procedure(phi_t),       deferred :: phi_t             ! Calculate d(potential)/dt␣
↪(Euler) at location for current time
    procedure(grad_phi),    deferred :: grad_phi          ! Calculate particle␣
↪velocity at location for current time
    procedure(grad_phi_2nd),deferred :: grad_phi_2nd      ! Calculate second order␣
↪spatial gradients of potential
    procedure(acc_euler),   deferred :: acc_euler         ! Calculate Euler␣
↪acceleration (grad(phi_t)) at location for current time
    procedure(acc_particle),deferred :: acc_particle      ! Calculate particle␣
↪acceleration at location for current time
    procedure(elev),        deferred :: elev              ! Calculate surface␣
↪elevation at location for current time
    procedure(elev_t),      deferred :: elev_t            ! Calculate d(surface␣
↪elevation)/dt (Euler) at location for current time
    procedure(grad_elev),   deferred :: grad_elev         ! Calculate gradient of␣
↪surface elevation at location for current time
    procedure(grad_elev_2nd),deferred:: grad_elev_2nd     ! Calculate second order␣
↪spatial gradients of elevation
    procedure(pressure),    deferred :: pressure          ! Fully nonlinear Bernoulli␣
↪pressure
    procedure(bathymetry),  deferred :: bathymetry        ! Return local depth at␣
↪application position (x, y)
    procedure(bathymetry_nvec),deferred :: bathymetry_nvec ! Unit normal vector of␣
↪sea floor into the ocean at (x,y)
    procedure(convergence), deferred :: convergence       ! For a specific (t,x,y,z)␣
↪return a csv-file on how particle velocity, elevation
                                                          ! and pressure converge as␣
↪a function of number of spectral components
    procedure(strip),       deferred :: strip             ! Create a new SWD file␣
↪based on a time window of current SWD file.
    procedure(get_int),     deferred :: get_int           ! Extract a specified int␣
↪parameter
```

(continues on next page)

(continued from previous page)

```
    procedure(get_logical), deferred :: get_logical       ! Extract a specified␣
↪logical parameter
    procedure(get_real),    deferred :: get_real          ! Extract a specified float␣
↪parameter
    procedure(get_chr),     deferred :: get_chr           ! Extract a specified char␣
↪parameter
    procedure(close),       deferred :: close             ! Manual destructor
    procedure :: error_raised                             ! Return .true. if error␣
↪has been signaled
    procedure :: error_id                                 ! Return error id
    procedure :: error_msg                                ! Return error message
    procedure :: error_clear                              ! Clear error signal (id=0)
end type spectral_wave_data

abstract interface
    !
    subroutine update_time(self, time)
        import
        class(spectral_wave_data), intent(inout) :: self  ! Update data in memory (if␣
↪needed)
        real(knd),                 intent(in)    :: time  ! Current time in␣
↪simulation program
    end subroutine update_time

    function phi(self, x, y, z) result(res)
        import
        class(spectral_wave_data), intent(in) :: self  ! Actual class
        real(knd),                 intent(in) :: x,y,z ! Position application program
        real(knd)                             :: res   ! Potential at (x,y,z)
    end function phi

    function stream(self, x, y, z) result(res)
        import
        class(spectral_wave_data), intent(in) :: self  ! Actual class
        real(knd),                 intent(in) :: x,y,z ! Position application program
        real(knd)                             :: res   ! Stream function value at (x,
↪y,z)
    end function stream

    function phi_t(self, x, y, z) result(res)
        import
        class(spectral_wave_data), intent(in) :: self  ! Actual class
        real(knd),                 intent(in) :: x,y,z ! Position application program
        real(knd)                             :: res   ! Euler time derivative of␣
↪potential at (x,y,z)
    end function phi_t

    function grad_phi(self, x, y, z) result(res)
        import
        class(spectral_wave_data), intent(in) :: self    ! Actual class
        real(knd),                 intent(in) :: x,y,z   ! Position application program
        real(knd)                             :: res(3) ! Particle velocity at (x,y,z)
    end function grad_phi

    function grad_phi_2nd(self, x, y, z) result(res)
        import
        class(spectral_wave_data), intent(in) :: self    ! Actual class
```

(continues on next page)

(continued from previous page)

```
        real(knd),                  intent(in) :: x,y,z  ! Position application program
        real(knd)                              :: res(6) ! Second order gradients of␣
↪potential at (x,y,z)
                                                         ! res(1) = d^2(potential) /␣
↪dx^2
                                                         ! res(2) = d^2(potential) /␣
↪dx dy
                                                         ! res(3) = d^2(potential) /␣
↪dx dz
                                                         ! res(4) = d^2(potential) /␣
↪dy^2
                                                         ! res(5) = d^2(potential) /␣
↪dy dz
                                                         ! res(6) = d^2(potential) /␣
↪dz^2
    end function grad_phi_2nd

    function acc_euler(self, x, y, z) result(res)
        import
        class(spectral_wave_data), intent(in) :: self   ! Actual class
        real(knd),                 intent(in) :: x,y,z  ! Position application program
        real(knd)                             :: res(3) ! Euler acceleration at (x,y,
↪z)
    end function acc_euler

    function acc_particle(self, x, y, z) result(res)
        import
        class(spectral_wave_data), intent(in) :: self   ! Actual class
        real(knd),                 intent(in) :: x,y,z  ! Position application program
        real(knd)                             :: res(3) ! Particle acceleration at (x,
↪y,z)
    end function acc_particle

    function elev(self, x, y) result(res)
        import
        class(spectral_wave_data), intent(in) :: self ! Actual class
        real(knd),                 intent(in) :: x,y  ! Position application program
        real(knd)                             :: res  ! Surface elevation at (x,y)
    end function elev

    function elev_t(self, x, y) result(res)
        import
        class(spectral_wave_data), intent(in) :: self ! Actual class
        real(knd),                 intent(in) :: x,y  ! Position application program
        real(knd)                             :: res  ! d/dt of surface elevation at␣
↪(x,y)
    end function elev_t

    function grad_elev(self, x, y) result(res)
        import
        class(spectral_wave_data), intent(in) :: self   ! Actual class
        real(knd),                 intent(in) :: x,y    ! Position application program
        real(knd)                             :: res(3) ! x, y and z gradients of␣
↪surface elevation at (x,y)
    end function grad_elev

    function grad_elev_2nd(self, x, y) result(res)
```

(continues on next page)

(continued from previous page)

```
        import
        class(spectral_wave_data), intent(in) :: self   ! Actual class
        real(knd),                 intent(in) :: x,y    ! Position application program
        real(knd)                             :: res(3) ! Second order gradients of␣
↪surface elevation
                                                        ! res(1) = d^2(elevation) /␣
↪dx^2
                                                        ! res(2) = d^2(elevation) /␣
↪dx dy
                                                        ! res(3) = d^2(elevation) /␣
↪dy^2
    end function grad_elev_2nd

    function pressure(self, x, y, z) result(res)
        import
        class(spectral_wave_data), intent(in) :: self   ! Actual class
        real(knd),                 intent(in) :: x,y,z  ! Position application program
        real(knd)                             :: res    ! Nonlinear pressure
    end function pressure

    function bathymetry(self, x, y) result(res)
        import
        class(spectral_wave_data), intent(in) :: self ! Actual class
        real(knd),                 intent(in) :: x,y  ! Position application program
        real(knd)                             :: res  ! Local depth at (x,y)
    end function bathymetry

    function bathymetry_nvec(self, x, y) result(res)
        import
        class(spectral_wave_data), intent(in) :: self ! Actual class
        real(knd),                 intent(in) :: x,y  ! Position application program
        real(knd)                             :: res(3) ! Unit normal vector of sea␣
↪floor into the ocean at (x,y)
    end function bathymetry_nvec

    subroutine convergence(self, x, y, z, csv)
        import
        class(spectral_wave_data), intent(inout) :: self   ! Actual class
        real(knd),                 intent(in) :: x,y,z  ! Position application program
        character(len=*),          intent(in) :: csv    ! Name of output csv-file
    end subroutine convergence

    subroutine strip(self, tmin, tmax, file_swd)
        ! Create a new swd file containing the spectral information limited
        ! to the application time window:  tmin <= t <= tmax.
        import
        class(spectral_wave_data), intent(inout) :: self       ! Actual class
        real(knd),                 intent(in) :: tmin, tmax ! Time window application␣
↪program
        character(len=*),          intent(in) :: file_swd   ! Name of new swd file
    end subroutine strip

    function get_int(self, name) result(res)
        import
        class(spectral_wave_data), intent(inout) :: self ! Actual class
        character(len=*),          intent(in) :: name ! Name of int parameter
        integer                               :: res  ! Value of name parameter
```

(continues on next page)

(continued from previous page)

```
    end function get_int

    function get_logical(self, name) result(res)
        import
        class(spectral_wave_data), intent(inout) :: self ! Actual class
        character(len=*),          intent(in) :: name ! Name of logical parameter
        logical                               :: res  ! Value of name parameter
    end function get_logical

    function get_real(self, name) result(res)
        import
        class(spectral_wave_data), intent(inout) :: self ! Actual class
        character(len=*), intent(in) :: name ! Name of real parameter
        real(knd)                    :: res  ! Value of name parameter
    end function get_real

    function get_chr(self, name) result(res)
        import
        class(spectral_wave_data), intent(inout) :: self ! Actual class
        character(len=*),          intent(in) :: name ! Name of char parameter
        character(len=:), allocatable         :: res  ! Value of name parameter
    end function get_chr

    subroutine close(self)
        import
        class(spectral_wave_data) :: self  ! Object to destruct
    end subroutine close

end interface

contains

!=============================================================================

function error_raised(self) result(res)
class(spectral_wave_data), intent(in) :: self ! Error handler
logical                               :: res  ! .true. if error has been signaled
!
res = self % error % raised()
!
end function error_raised

!=============================================================================

function error_id(self) result(res)
class(spectral_wave_data), intent(in) :: self ! Error handler
integer                               :: res  ! Return error code
!
res = self % error % id
!
end function error_id

!=============================================================================

function error_msg(self) result(res)
class(spectral_wave_data), intent(in)   :: self ! Error handler
character(len=len_trim(self%error%msg)) :: res  ! Return error code
```

(continues on next page)

(continued from previous page)

```
!
res = self % error % msg
!
end function error_msg

!===============================================================================

subroutine error_clear(self)
class(spectral_wave_data), intent(inout) :: self ! Error handler
!
call self % error % clear()
!
end subroutine error_clear

!===============================================================================

end module spectral_wave_data_def
```

6.4.2 spectral_wave_data_allocate.f90

```
module spectral_wave_data_allocate_def

use, intrinsic :: iso_c_binding,   only: c_int, c_float, c_char

! In kind_values.f90 adjust kind_swd_interface to match applied
! precision in the application program. (real, double precision etc.)
use kind_values, only: knd => kind_swd_interface

! Open file with correct endianess
use open_swd_file_def, only: open_swd_file, swd_validate_binary_convention

! The abstract base class
use spectral_wave_data_def, only: spectral_wave_data

! Include support for all spectral_wave_data implementations
use spectral_wave_data_shape_1_impl_1_def, only: spectral_wave_data_shape_1_impl_1
use spectral_wave_data_shape_2_impl_1_def, only: spectral_wave_data_shape_2_impl_1
use spectral_wave_data_shape_3_impl_1_def, only: spectral_wave_data_shape_3_impl_1
use spectral_wave_data_shape_4_impl_1_def, only: spectral_wave_data_shape_4_impl_1
use spectral_wave_data_shape_4_impl_2_def, only: spectral_wave_data_shape_4_impl_2
use spectral_wave_data_shape_5_impl_1_def, only: spectral_wave_data_shape_5_impl_1
use spectral_wave_data_shape_6_impl_1_def, only: spectral_wave_data_shape_6_impl_1

implicit none
private

! This module provides a procedure for selecting and allocating
! the proper SWD class based on the header of the actual SWD-file
! and optional requested implementation.
!
! Developers may add their own implementations which can
! be selected using this procedure.
!
! Written by Jens Bloch Helmers, August, 2. 2019
!
```

(continues on next page)

(continued from previous page)

```
!------------------------------------------------------------------------------

!##############################################################################
!
!              B E G I N    P U B L I C    Q U A N T I T I E S
!
!------------------------------------------------------------------------------
!
public :: spectral_wave_data_allocate
!
!------------------------------------------------------------------------------
!
!                E N D    P U B L I C    Q U A N T I T I E S
!
!##############################################################################

contains

!==============================================================================

subroutine spectral_wave_data_allocate(swd, file_swd, x0, y0, t0, beta, rho, &
                                       nsumx, nsumy, impl, ipol, norder,     &
                                       dc_bias)
!------------------------------------------------------------------------------
! This procedure selects and allocates the proper SWD class based on
! the header of the actual SWD-file and an optional requested implementation.
!------------------------------------------------------------------------------

! The actual class to be allocated
class(spectral_wave_data), allocatable, intent(out) :: swd

! Name of actual swd file
character(len=*), intent(in) :: file_swd

! Relation between SWD and application coordinates
real(knd), intent(in) :: x0, y0, t0, beta

! The remaining parameters are optional...

! Density of water (applied for pressure calculations)
real(knd), optional, intent(in) :: rho

! If present and nsumx>-1: apply nsumx number of spectral components
!  in x-direction, else apply all spectral components from the SWD file.
integer, optional, intent(in) :: nsumx


! If present and nsumy>-1: apply nsumy number of spectral components
! in y-direction, else apply all spectral components from the SWD file.
integer, optional, intent(in) :: nsumy

! Index to determine actual derived class:
!   0  = Default
!   <0 = In-house and experimental implementations
!   >0 = Validated implementations available open-software
integer, optional, intent(in) :: impl
```

(continues on next page)

(continued from previous page)

```
! Index to request actual temporal interpolation scheme
!   0 = Default (C^2 continous scheme)
!   1 = C^1 continous
!   2 = C^3 continous
integer, optional, intent(in) :: ipol

! Expansion order to apply in kinematics for z>0
!   0  = Apply expansion order specified on swd file (default)
!   <0 = Apply exp(kj z)
!   >0 = Apply expansion order = norder
integer, optional, intent(in) :: norder

! Control application of zero-frequency bias present in SWD file
!  True  = Apply zero frequency amplitudes from SWD file.
!  False = Suppress contribution from zero frequency amplitudes (Default)
logical, optional, intent(in):: dc_bias
!------------------------------------------------------------------------------

integer :: impl_swd, nsumx_swd, nsumy_swd, ios, shp, amp, ipol_swd
integer :: norder_swd, err_id, lu
real(knd) :: rho_swd
logical :: select_ok, dc_bias_swd, equal_nx_ny
character(len=*), parameter :: err_proc = 'spectral_wave_data_allocate_def::spectral_
↪wave_data_allocate'
character(len=250) :: err_msg(5)

! Assign default values if not present...
if (present(rho)) then
    rho_swd = rho
else
    rho_swd = 1025.0
end if
if (present(nsumx)) then
    nsumx_swd = nsumx
else
    nsumx_swd = -1
end if
if (present(nsumy)) then
    nsumy_swd = nsumy
else
    nsumy_swd = -1
end if
if (present(impl)) then
    impl_swd = impl
else
    impl_swd = 0
end if

if (present(ipol)) then
    ipol_swd = ipol
else
    ipol_swd = 0
end if

if (present(norder)) then
    norder_swd = norder
else
```

(continues on next page)

(continued from previous page)

```
    norder_swd = 0
end if

if (present(dc_bias)) then
    dc_bias_swd = dc_bias
else
    dc_bias_swd = .false.
end if

! Check if it is possible to open input SWD file...
call open_swd_file(newunit=lu, file=file_swd, status='old', &
                   as_little_endian=.true., iostat=ios)
if (ios /= 0) then
    write(err_msg(1),'(a,a)') 'SWD file: ', trim(file_swd)
    write(err_msg(2),'(a)') "Not able to open SWD file."
    allocate(spectral_wave_data_shape_1_impl_1 :: swd)  ! Empty object to store errors
    call swd % error % set_id_msg(err_proc, 1001, err_msg(1:2))
    return
end if
close(lu)

! Before we decode the SWD file we check if
! it applies the correct binary convention...
call swd_validate_binary_convention(file_swd, err_msg(2))
if (err_msg(2) /= '') then
    write(err_msg(1),'(a,a)') 'SWD file: ', trim(file_swd)
    allocate(spectral_wave_data_shape_1_impl_1 :: swd)  ! Empty object to store errors
    call swd % error % set_id_msg(err_proc, 1002, err_msg(1:2))
    return
end if

! Obtain shp and amp from actual SWD file
call swd_sniff_shp_amp()
if (err_msg(2) /= '') then
    write(err_msg(1),'(a,a)') 'SWD file: ', trim(file_swd)
    allocate(spectral_wave_data_shape_1_impl_1 :: swd)  ! Empty object to store errors
    call swd % error % set_id_msg(err_proc, err_id, err_msg(1:2))
    return
end if

select_ok = .false.
select case(shp)
case(1)
    if (impl_swd == 0 .or. impl_swd == 1) then
        if (amp/=2) then
            select_ok = .true.
            allocate(swd,                                          &
                source=spectral_wave_data_shape_1_impl_1(      &
                file_swd, x0, y0, t0, beta, rho=rho_swd,       &
                nsumx=nsumx_swd, ipol=ipol_swd, norder=norder_swd, &
                dc_bias=dc_bias_swd), stat=ios)
        end if
    end if
case(2)
    if (impl_swd == 0 .or. impl_swd == 1) then
        if (amp/=2) then
            select_ok = .true.
```

(continues on next page)

(continued from previous page)

```
            allocate(swd,                                              &
                source=spectral_wave_data_shape_2_impl_1(      &
                file_swd, x0, y0, t0, beta, rho=rho_swd,       &
                nsumx=nsumx_swd, ipol=ipol_swd, norder=norder_swd, &
                dc_bias=dc_bias_swd), stat=ios)
        end if
    end if
case(3)
    if (impl_swd == 0 .or. impl_swd == 1) then
        if (amp/=2) then
            select_ok = .true.
            allocate(swd,                                              &
                source=spectral_wave_data_shape_3_impl_1(      &
                file_swd, x0, y0, t0, beta, rho=rho_swd,       &
                nsumx=nsumx_swd, ipol=ipol_swd, norder=norder_swd, &
                dc_bias=dc_bias_swd), stat=ios)
        end if
    end if
case(4)
    if (impl_swd == 0) then
        equal_nx_ny = swd_sniff_equal_nx_ny()
        if (err_msg(2) /= '') then
            write(err_msg(1),'(a,a)') 'SWD file: ', trim(file_swd)
            allocate(spectral_wave_data_shape_4_impl_1 :: swd)  ! Empty object to␣
↪store errors
            call swd % error % set_id_msg(err_proc, err_id, err_msg(1:2))
            return
        end if
        if (equal_nx_ny) then
            impl_swd = 2
        else
            impl_swd = 1
        end if
    end if
    if (impl_swd == 1) then
        if (amp/=2) then
            select_ok = .true.
            allocate(swd,                                                  &
                source=spectral_wave_data_shape_4_impl_1(            &
                file_swd, x0, y0, t0, beta, rho=rho_swd,             &
                nsumx=nsumx_swd, nsumy=nsumy_swd, norder=norder_swd, &
                dc_bias=dc_bias_swd, ipol=ipol_swd), stat=ios)
        end if
    else if (impl_swd == 2) then
        if (amp/=2) then
            select_ok = .true.
            allocate(swd,                                                  &
                source=spectral_wave_data_shape_4_impl_2(            &
                file_swd, x0, y0, t0, beta, rho=rho_swd,             &
                nsumx=nsumx_swd, nsumy=nsumy_swd, norder=norder_swd, &
                dc_bias=dc_bias_swd, ipol=ipol_swd), stat=ios)
        end if
    end if
case(5)
    if (impl_swd == 0 .or. impl_swd == 1) then
        if (amp/=2) then
            select_ok = .true.
```

(continues on next page)

(continued from previous page)

```
            allocate(swd,                                                  &
                source=spectral_wave_data_shape_5_impl_1(                  &
                file_swd, x0, y0, t0, beta, rho=rho_swd,                   &
                nsumx=nsumx_swd, nsumy=nsumy_swd, norder=norder_swd, &
                dc_bias=dc_bias_swd, ipol=ipol_swd), stat=ios)
        end if
    end if
case(6)
    if (impl_swd == 0 .or. impl_swd == 1) then
        select_ok = .true.
        allocate(swd,                                          &
            source=spectral_wave_data_shape_6_impl_1(      &
            file_swd, x0, y0, t0, beta, rho=rho_swd,       &
            nsumx=nsumx_swd, ipol=ipol_swd, norder=norder_swd, &
            dc_bias=dc_bias_swd), stat=ios)
    end if
end select

if (select_ok) then
    if (swd % error % raised()) then
        return
    end if
else
    write(err_msg(1),'(a,a)') 'SWD file: ', trim(file_swd)
    write(err_msg(2),'(a)')  "Unsupported combination of shp/amp/impl"
    write(err_msg(3),'(a,i0)') "shp = ", shp
    write(err_msg(4),'(a,i0)') "amp = ", amp
    write(err_msg(5),'(a,i0)') "impl = ", impl
    call swd % error % set_id_msg(err_proc, 1003, err_msg(1:5))
    return
end if

if (ios /= 0) then
    write(err_msg(1),'(a,a)') 'SWD file: ', trim(file_swd)
    write(err_msg(2),'(a)')  'Not possible to allocate actual class!'
    write(err_msg(3),'(a,i0)') "shp = ", shp
    write(err_msg(4),'(a,i0)') "amp = ", amp
    write(err_msg(5),'(a,i0)') "impl = ", impl
    allocate(spectral_wave_data_shape_1_impl_1 :: swd)  ! Empty object to store errors
    call swd % error % set_id_msg(err_proc, 1005, err_msg(1:5))
    return
end if

! SWD now has a data type and content...

contains

    subroutine swd_sniff_shp_amp()
    ! Extract shp and amp from swd file
    integer :: luswd
    integer(c_int) :: c_fmt, c_shp, c_amp
    real(c_float) :: c_magic
    err_msg(2) = ''
    call open_swd_file(newunit=luswd, file=file_swd, status='old', &
                       as_little_endian=.true., iostat=ios)
    if ( ios/=0 ) then
        err_id = 1001
```

(continues on next page)

(continued from previous page)

```
        err_msg(2) = 'Not able to open SWD file.'
        return
    end if
    read(luswd) c_magic
    read(luswd) c_fmt
    if (c_fmt /= int(100, c_int)) then
        err_id = 1003
        write(err_msg(2),'(a,i0)') 'Unexpected version number of SWD-file. fmt = ', c_
↪fmt
        return
    end if
    read(luswd) c_shp
    read(luswd) c_amp
    shp = c_shp
    amp = c_amp
    close(luswd)
    end subroutine swd_sniff_shp_amp

    logical function swd_sniff_equal_nx_ny() result(res)
    ! Return true if dkx==dky and nx==ny
    integer :: luswd
    integer(c_int) :: c_fmt, c_shp, c_amp
    integer(c_int) :: c_nx, c_ny, c_order, c_nid, c_nsteps, c_nstrip
    real(c_float) :: c_dkx, c_dky, c_dt, c_grav, c_lscale, c_magic
    character(kind=c_char, len=:), allocatable :: cid
    character(kind=c_char, len=30) :: c_prog
    character(kind=c_char, len=20) :: c_date
    err_msg(2) = ''
    res = .false.
    call open_swd_file(newunit=luswd, file=file_swd, status='old', &
                       as_little_endian=.true., iostat=ios)
    if ( ios/=0 ) then
        err_id = 1001
        err_msg(2) = 'Not able to open SWD file.'
        return
    end if
    read(luswd) c_magic
    read(luswd) c_fmt
    if (c_fmt /= int(100, c_int)) then
        err_id = 1003
        write(err_msg(2),'(a,i0)') 'Unexpected version number of SWD-file. fmt = ', c_
↪fmt
        return
    end if
    read(luswd) c_shp
    if (c_shp == 4 .or. c_shp == 5) then
        read(luswd) c_amp
        read(luswd) c_prog
        read(luswd) c_date
        read(luswd) c_nid
        allocate(character(len=int(c_nid)) :: cid)
        read(luswd) cid
        read(luswd) c_grav
        read(luswd) c_lscale
        read(luswd) c_nstrip
        read(luswd) c_nsteps
        read(luswd) c_dt
```

(continues on next page)

(continued from previous page)

```
        read(luswd) c_order
        read(luswd) c_nx
        read(luswd) c_ny
        read(luswd) c_dkx
        read(luswd) c_dky
        res = (c_nx == c_ny) .and. (abs(c_dkx - c_dky) < epsilon(c_dkx))
    end if
    close(luswd)
    end function swd_sniff_equal_nx_ny

end subroutine spectral_wave_data_allocate

!=============================================================================

end module spectral_wave_data_allocate_def
```

CHAPTER

SEVEN

PROGRAMMING

In this section we describe how to include support for the **spectral_wave_data** API in wave generators and application programs.

- *wave generators*
- *application programs*

7.1 Wave generators

There are no restrictions on programming styles applied in *wave generators* as long as it can produce a sound SWD-file.

Wave generators are typical limited to provide wave fields described by a small set of spectral formulations. Often only one or two formulations. Consequently, only a small subset of the *SWD-specification* needs to be considered when including support for writing SWD files.

7.1.1 Case Studies

Before considering any programming, you first need to establish the relationship between the mathematical formulation you apply in your *wave generator*, and the relevant SWD shape formulation.

To assist you in this process we provide two case studies from real world wave generators.

- *Case 1* explains how the formulation applied in HOS-Ocean, for long-crested finite depth waves, relates to the SWD formulation *shape 2*.
- *Case 2* relates a variable depth flow formulation by Gouin et al. to the SWD formulation *shape 3*.

Case 1: The HOS-Ocean formulation in relation to SWD

The formulation applied in HOS-Ocean, for long-crested waves propagating over a constant water depth d, is explained in [DBTouzeF16]. From equations (6) and (8) in that publication, the velocity potential ϕ and surface elevation ζ are defined as

$$\phi(\mathbf{x},t) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{A_j(t)\,\frac{\cosh k_j(z+d)}{\cosh k_j d}\,e^{ik_j x}\Big\}, \qquad k_j = j\Delta k_x$$

$$\zeta(x,y,t) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{B_j^{\zeta}(t)\,e^{ik_j x}\Big\}$$

We have adjusted the symbols for wave elevation, water depth and loop index to simplify the comparison with the SWD documentation. When *HOS-Ocean* has calculated the temporal spectral amplitudes $A_j(t)$ and $B_j^\zeta(t)$ the kinematics can be calculated everywhere using the above equations.

We compare the formulation above to *shape 2* as defined in the theory section of our documentation:

$$\phi(\mathbf{x},t) = \sum_{j=0}^{n} \mathcal{R}\rceil\left\{c_j(t)\, e^{-ik_j x}\right\} \frac{\cosh k_j(z+d)}{\cosh k_j d}$$

$$\zeta(x,y,t) = \sum_{j=0}^{n} \mathcal{R}\rceil\left\{h_j(t)\, e^{-ik_j x}\right\}$$

It is noted that the sign in the exponential terms are different in theses formulations. We can apply the SWD formulation if and only if the kinematics in the physical space is identical to the *HOS-Ocean* formulation. To satisfy this requirement it follows from pure algebra that:

$$c_j(t) = \bar{A}_j(t), \qquad h_j(t) = \bar{B}_j^\zeta(t)$$

$\bar{A}_j(t)$ and $\bar{B}_j^\zeta(t)$ denote the complex conjugate of $A_j(t)$ and $B_j^\zeta(t)$ respectively.

Hence, when *HOS-Ocean* has established $A_j(t)$ and $B_j^\zeta(t)$ at the end of each time step, the program may in an SWD output routine calculate $c_j(t)$ and $h_j(t)$ from above equations and write these SWD specific amplitudes to the *SWD-file*.

In addition SWD requires output of $\frac{dc_j(t)}{dt}$ and $\frac{dh_j(t)}{dt}$. It follows from differentiation that

$$\frac{dc_j(t)}{dt} = \frac{d\bar{A}_j(t)}{dt}, \qquad \frac{dh_j(t)}{dt} = \frac{d\bar{B}_j^\zeta(t)}{dt}$$

It may be tempting to evaluate the right-hand side of these equations using some kind of finite-difference schemes. However, as pointed out in the SWD theory section, higher accuracy is obtained if these expressions are evaluated from the dynamic and kinematic free surface conditions. This is because these expressions can be evaluated using analytical spatial gradients as a replacement for the temporal gradient. In practice these terms can often be extracted from the right-hand side of the ODE equation system.

Noting that the physical boundary conditions explicitly define the temporal slopes of these amplitudes, this is also important information regarding temporal interpolation to be applied in the application program. This is accounted for in the SWD API assuming sound input of $\frac{dc_j(t)}{dt}$ and $\frac{dh_j(t)}{dt}$.

In practice, a copy of the subroutine *swd_write_shape_1_or_2.f90* may be adapted for convenient SWD output. This code takes care of the low level C-stream output.

Note: It was assumed that the wave train propagates in the positive x-direction in *HOS-Ocean*, the same convention as assumed in SWD. If the wave moves in the negative direction in *HOS-Ocean*, we must flip the sign of the x-axis to comply with the SWD convention. Hence, we then must apply $c_j(t) = A_j(t)$ and $h_j(t) = B_j^\zeta(t)$ as a result of conjugating twice.

Case 2: The variable depth formulation by Gouin et al. in relation to SWD

The formulation applied by [GDF16] considers long-crested potential flow waves propagating over a variable water depth. The local water depth $d(x)$ is defined as

$$d(x) = d_0 - \beta(x), \qquad d_0 > 0, \qquad d(x) > 0$$

where d_0 is a constant and the spatial variation is prescribed by the function $\beta(x)$. From equations (8) and (9) in [GDF16] the wave potential is split into two components

$$\phi(\mathbf{x}, t) = \phi_{d_0}(\mathbf{x}, t) + \phi_\beta(\mathbf{x}, t)$$

$$\phi_{d_0}(\mathbf{x}, t) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{ A_j(t)\, \frac{\cosh k_j(z + d_0)}{\cosh k_j d_0} e^{ik_j x} \Big\}, \qquad k_j = j\Delta k_x$$

$$\phi_\beta(\mathbf{x}, t) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{ B_j(t)\, \frac{\sinh k_j z}{\cosh k_j d_0} e^{ik_j x} \Big\}$$

$$\zeta(x, y, t) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{ H_j(t)\, e^{ik_j x} \Big\}$$

The motivation for this split is that ϕ_{d_0} represents the classical formulation if $\beta(x) \equiv 0$. The potential ϕ_β has only minor contribution at the free surface, in fact the potential vanish at $z = 0$. At the sea bed the fraction $\frac{\sinh k_j z}{\cosh k_j d_0}$ is close to -1. Hence ϕ_β has similar numerical properties at the sea bed as ϕ_{d_0} has close to the free surface and vice versa. Attractive numerical properties using this split are demonstrated in [GDF16].

If we compare the above formulation to *shape 3* as defined in the theory section of our documentation:

$$\phi(\mathbf{x}, t) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{ c_j(t)\, e^{-ik_j x} \Big\} e^{k_j z} + \sum_{j=0}^{\hat{n}} \mathcal{R}\rceil\Big\{ \hat{c}_j(t)\, e^{-ik_j x} \Big\} e^{-k_j z}$$

$$\zeta(x, t) = \sum_{j=0}^{n} \mathcal{R}\rceil\Big\{ h_j(t)\, e^{-ik_j x} \Big\}$$

we can apply the SWD formulation if and only if the kinematics in the physical space is identical to the original formulation presented above by [GDF16]. To satisfy this requirement it follows from pure algebra that:

$$c_j(t) = \gamma_j(t) + \hat{\gamma}_j(t), \qquad \hat{c}_j(t) = \gamma_j(t) e^{-2k_j d_0} - \hat{\gamma}_j(t), \qquad h_j(t) = \bar{H}_j(t)$$

$$\hat{n} = n, \qquad \gamma_j(t) = \frac{1 + \tanh k_j d_0}{2} \bar{A}_j(t), \qquad \hat{\gamma}_j(t) = \frac{e^{-k_j d_0}}{1 + e^{-2k_j d_0}} \bar{B}_j(t)$$

$\bar{A}_j(t)$, $\bar{B}_j(t)$ and $\bar{H}_j(t)$ denote the complex conjugate of $A_j(t)$, $B_j(t)$ and $H_j(t)$ respectively. The conjugate account for the change of sign present in the exponential terms related to x.

Very similar numerical properties are observed for the SWD formulation. For large wave numbers the main contribution to $c_j(t)$ derives from $A_j(t)$, while the main contribution to $\hat{c}_j(t)$ derives from $B_j(t)$. At wave crests the main contribution derives from $c_j(t)$ and at the sea floor from $\hat{c}_j(t)$.

When the wave generator has established $A_j(t)$, $B_j(t)$ and $H_j(t)$ at the end of each time step, the program may in an SWD output routine calculate $c_j(t)$, $\hat{c}_j(t)$ and $h_j(t)$ from above equations. Then write the SWD specific amplitudes to the *SWD file*.

In addition SWD requires output of $\frac{dc_j(t)}{dt}$, $\frac{d\hat{c}_j(t)}{dt}$ and $\frac{dh_j(t)}{dt}$. It follows from differentiation that

$$\frac{dc_j(t)}{dt} = \frac{d\gamma_j(t)}{dt} + \frac{d\hat{\gamma}_j(t)}{dt}, \qquad \frac{d\hat{c}_j(t)}{dt} = e^{-2k_j d_0} \frac{d\gamma_j(t)}{dt} - \frac{d\hat{\gamma}_j(t)}{dt}$$

$$\frac{d\gamma_j(t)}{dt} = \frac{1+\tanh k_j d_0}{2}\frac{d\bar{A}_j(t)}{dt}, \qquad \frac{d\hat{\gamma}_j(t)}{dt} = \frac{e^{-k_j d_0}}{1+e^{-2k_j d_0}}\frac{d\bar{B}_j(t)}{dt}$$
$$\frac{dh_j(t)}{dt} = \frac{d\bar{H}_j(t)}{dt}$$

It may be tempting to evaluate $\frac{dA_j(t)}{dt}$, $\frac{dB_j(t)}{dt}$ and $\frac{dH_j(t)}{dt}$ using some kind of finite-difference schemes. However, as pointed out in the SWD theory section, higher accuracy is obtained if these expressions are evaluated from the dynamic and kinematic free surface conditions. This is because these expressions can be evaluated using analytical spatial gradients as a replacement for the temporal gradient.

In practice these terms can often be extracted from the right-hand side of the ODE equation system. Consequently, the reconstructed flow field in SWD, satisfies the free surface boundary conditions to a higher accuracy.

A copy of the subroutine *swd_write_shape_3.f90* may be adapted for convenient SWD output. This code takes care of the low level C-stream output.

Note: It was assumed that the wave train propagates in the positive x-direction in the original formulation, the same convention as assumed in SWD. If the wave moves in the negative direction, it is in general **not possible** to just flip the wave propagation direction due to the varying sea floor, unless the sequence of sea floor offset points are reversed when writing the SWD file.

Note: For most sea floor topologies, it is mainly the low frequency components of $\hat{c}_j(t)$ or $(B_j(t))$ that contributes to the flow field. Consequently, the wave generator may specify $\hat{n} < n$ to reduce the computational cost.

7.1.2 Programming

Fortran

Fortran programmers may apply the following section for how to write a sound SWD file from a *wave generator*. The handling of making a C binary stream is demonstrated.

For each spectral formulation this repository provides source code demonstrating how to create proper SWD-files with minimal effort. Implementation or adaption of this source code to the actual *wave generator* is expected to be a very small job.

Spectral model	Example source files for wave generators
shape 1 or 2	*swd_write_shape_1_or_2.f90*
shape 3	*swd_write_shape_3.f90*
shape 4 or 5	*swd_write_shape_4_or_5.f90*
shape 6	*swd_write_shape_6.f90*

swd_write_shape_1_or_2.f90

```
module swd_write_shape_1_or_2_def

use, intrinsic :: iso_c_binding,   only: c_char, c_int, c_float, c_null_char

implicit none
private

```

(continues on next page)

(continued from previous page)

```
! Please adjust wp to the working precision applied in the wave generator
integer, parameter :: wp = kind(1.0d0) ! (double precision)


! This module provides a class for writing SWD files for shape 1 and 2.
!
! Written by Jens Bloch Helmers, August, 1. 2019
!
!------------------------------------------------------------------------------

!##############################################################################
!
!              B E G I N    P U B L I C    Q U A N T I T I E S
!
!------------------------------------------------------------------------------
!
public :: swd_write_shape_1_or_2
!
!------------------------------------------------------------------------------
!
!                E N D    P U B L I C    Q U A N T I T I E S
!
!##############################################################################

type swd_write_shape_1_or_2
    integer :: amp    ! Type of spectral amplitudes to be provided
    integer :: lu     ! Unit number associated with swd file
    integer :: nsteps ! Expected number of time steps
    integer :: it     ! Current time steps
    integer :: n      ! Number of points in physical space (x-dir)
    integer :: n_swd  ! n in swd theory description (=int(n/2))
contains
    procedure :: init    ! Open the swd file and write the header section
    procedure :: update  ! Add spectral data at current time step
    procedure :: update_fft  ! Alternative output method based on FFTW r2c data
    procedure :: close   ! close the file
end type swd_write_shape_1_or_2

contains

!==============================================================================

subroutine init(self, file, prog, cid, grav, Lscale, amp, &
                n, order, dk, dt, nsteps, d)
class(swd_write_shape_1_or_2), intent(out) :: self  ! Object to create
character(len=*), intent(in)  :: file  ! Requested name for new SWD file
character(len=*), intent(in)  :: prog  ! Name of actual wave generator including␣
↪version number
character(len=*), intent(in)  :: cid   ! Text describing actual input parameters to␣
↪the
                                       ! wave generator. Hint: write(cid, nml=wave_
↪input_namelist)
real(wp),         intent(in)  :: grav  ! Acceleration of gravity applied in wave␣
↪generator
real(wp),         intent(in)  :: lscale! Number of length units per meter applied in␣
↪wave generator
                                       ! E.g. if US feet was applied: Lscale = 1/0.
↪3048 = 3.2808
```

(continues on next page)

(continued from previous page)

```
integer,           intent(in)  :: amp   ! Flag to indicate type of provided spectral␣
↪amplitudes
integer,           intent(in)  :: n     ! n as defined in the theory documentation
integer,           intent(in)  :: order ! Expansion order applied in wave generator. (
↪<0 for fully non-linear)
real(wp),          intent(in)  :: dk    ! Spacing of k-numbers in x-direction
real(wp),          intent(in)  :: dt    ! Spacing of time steps in the swd file
integer,           intent(in)  :: nsteps ! Total number of time steps to be stored in␣
↪the swd file.
real(wp),          intent(in), optional  :: d ! Water depth (if not present, deep␣
↪water (shape 1) is assumed)
!
integer :: nid
integer, parameter :: fmt = 100
integer :: shp
integer, parameter :: nstrip = 0
! We need some C compatible characters
character(kind=c_char, len=:), allocatable :: ccid
character(kind=c_char, len=30) :: cprog
character(kind=c_char, len=20) :: cdate
!
open(newunit=self % lu, file=file, access='stream', form='unformatted', &
     status='replace', action='write', convert='little_endian')
! The CONVERT option is an Intel, GFortran, HP and IBM extension.
! Modify this parameter for other compilers if your compiler complains.
!
self % amp = amp
self % nsteps = nsteps
self % it = 0
self % n = n
self % n_swd = n/2
!
shp = 1
if (present(d)) then
   if (d > 0.0_wp) then
      shp = 2
   end if
end if
!
nid = len_trim(cid) + 1  ! one extra for C null character
allocate(character(len=nid) :: ccid)
ccid = trim(cid) // c_null_char
cdate = timestamp() // c_null_char
cprog = trim(prog) // c_null_char
!
write(self % lu) 37.0221_c_float
write(self % lu) int(fmt, c_int)
write(self % lu) int(shp, c_int)
write(self % lu) int(amp, c_int)
write(self % lu) cprog
write(self % lu) cdate
write(self % lu) int(nid, c_int)
write(self % lu) ccid(:nid)
write(self % lu) real(grav, c_float)
write(self % lu) real(lscale, c_float)
write(self % lu) int(nstrip, c_int)
write(self % lu) int(nsteps, c_int)
```

(continues on next page)

(continued from previous page)

```
write(self % lu) real(dt, c_float)
write(self % lu) int(order, c_int)
write(self % lu) int(self % n_swd, c_int)
write(self % lu) real(dk, c_float)
if (shp == 2) then
   write(self % lu) real(d, c_float)
end if

contains

  function timestamp() result(timestr)
    ! Return a string with the current time in the form YYYY-MM-DD hh-mm-ss
    character (len = 19) :: timestr

    integer :: y, m, d, h, n, s
    integer :: values(8)

    call date_and_time (values = values)

    y = values(1)
    m = values(2)
    d = values(3)
    h = values(5)
    n = values(6)
    s = values(7)

    write (timestr, '(i4.4,a,i2.2,a,i2.2,a,i2.2,a,i2.2,a,i2.2)' ) &
         y,'-',m,'-',d,' ',h,':',n,':',s

  end function timestamp
!
end subroutine init

!===============================================================================

subroutine update(self, h, ht, c, ct)
! Output of temporal functions as defined in the shape 1 and 2 classes
class(swd_write_shape_1_or_2), intent(inout) :: self   ! Object to update
complex(wp),           intent(in) :: h(0:)   ! h(0:n_swd) temporal amp
complex(wp),           intent(in) :: ht(0:)  ! dh/dt(0:n_swd) temporal amp
complex(wp), optional, intent(in) :: c(0:)   ! c(0:n_swd) temporal amp
complex(wp), optional, intent(in) :: ct(0:)  ! dc/dt(0:n_swd) temporal amp
!
self % it = self % it + 1
call dump(h)
call dump(ht)
if (self % amp < 3) then
    call dump(c)
    call dump(ct)
end if

contains

    subroutine dump(array)
        complex(wp), intent(in) :: array(0:)
        integer :: j
        do j = 0, self % n_swd
```

(continues on next page)

(continued from previous page)

```
            write(self % lu) cmplx(array(j), kind=c_float)
        end do
    end subroutine dump
!
end subroutine update

!==============================================================================

subroutine update_fft(self, h_fft, ht_fft, c_fft, ct_fft)
! Alternative to using the 'update' routine.
! Output of temporal functions as defined in the shape 1 and 2 classes
! Input arrays are in the form returned by FFTW r2c 1D transforms
class(swd_write_shape_1_or_2), intent(inout) :: self   ! Object to update
complex(wp),           intent(in) :: h_fft(:)   ! h_fft(1:n/2+1) temporal amp
complex(wp),           intent(in) :: ht_fft(:)  ! dh/dt(1:n/2+1) temporal amp
complex(wp), optional, intent(in) :: c_fft(:)   ! c(1:n/2+1) temporal amp
complex(wp), optional, intent(in) :: ct_fft(:)  ! dc/dt(1:n/2+1) temporal amp
!
self % it = self % it + 1
call dump_fft(h_fft)
call dump_fft(ht_fft)
if (self % amp < 3) then
    call dump_fft(c_fft)
    call dump_fft(ct_fft)
end if

contains

  subroutine dump_fft(array)
    complex(wp), intent(in) :: array(:)
    integer :: jx
    !-------------------------------------------------------------------------
    ! Explicit loops to demonstrate element ordering for all languages.
    !-------------------------------------------------------------------------
    ! zero-frequency component
    write(self % lu) cmplx(real(array(1)), kind=c_float)
    ! all frequency up to (not including) nyquist freq
    do jx = 2, self % n/2
       write(self % lu) cmplx(2.0_wp*conjg(array(jx)), kind=c_float)
    end do
    ! nyquist frequency
    if (mod(self % n, 2) == 0) then
       write(self % lu) cmplx(real(array(self % n/2 + 1)), kind=c_float)
    else
       write(self % lu) cmplx(2.0_wp*conjg(array(self % n/2 + 1)), kind=c_float)
    end if
  end subroutine dump_fft
!
end subroutine update_fft

!==============================================================================

subroutine close(self)
class(swd_write_shape_1_or_2), intent(inout) :: self   ! Object to update
!
if (self % it /= self % nsteps) then
    print*, "WARNING: from swd_write_shape_1_or_2 :: close"
```

(continues on next page)

(continued from previous page)

```
    print*, "Specified number of time steps = ", self % nsteps
    print*, "Number of provided time steps = ", self % it
end if
!
close(self % lu)
!
end subroutine close

!=======================================================================================

end module swd_write_shape_1_or_2_def
```

swd_write_shape_3.f90

```
module swd_write_shape_3_def

use, intrinsic :: iso_c_binding,   only: c_char, c_int, c_float, c_null_char

implicit none
private

! Please adjust wp to the working precision applied in the wave generator
integer, parameter :: wp = kind(1.0d0) ! (double precision)


! This module provides a class for writing SWD files for shape 3.
!
! Written by Jens Bloch Helmers, August, 1. 2019
!
!---------------------------------------------------------------------------------

!#################################################################################
!
!              B E G I N    P U B L I C    Q U A N T I T I E S
!
!---------------------------------------------------------------------------------
!
public :: swd_write_shape_3
!
!---------------------------------------------------------------------------------
!
!               E N D    P U B L I C    Q U A N T I T I E S
!
!#################################################################################

type :: swd_write_shape_3
    integer :: amp    ! Type of spectral amplitudes to be provided
    integer :: lu     ! Unit number associated with swd file
    integer :: nsteps ! Expected number of time steps
    integer :: it     ! Current time steps
    integer :: n      ! Number of spectral components in x-dir
    integer :: nh     ! Number of auxilary shapes (hat(n) in theory manual)
    integer :: nsf    ! Number of sea floor points
contains
```

(continues on next page)

(continued from previous page)

```
    procedure :: init    ! Open the swd file and write the header section
    procedure :: update  ! Add spectral data at current time step
    procedure :: close   ! close the file
end type swd_write_shape_3

real(wp), parameter :: pi = 3.
↪14159265358979323846264338327950288419716939937510582097494_wp

contains

!=============================================================================

subroutine init(self, file, prog, cid, grav, Lscale, amp, &
                n, nh, order, dk, dt, nsteps, isf, nsf, xsf, zsf)
class(swd_write_shape_3),  intent(out) :: self  ! Object to create
character(len=*),  intent(in)  :: file  ! Requested name for new SWD file
character(len=*),  intent(in)  :: prog  ! Name of actual wave generator including␣
↪version number
character(len=*),  intent(in)  :: cid   ! Text describing actual input parameters to␣
↪the
                                        ! wave generator. Hint: write(cid, nml=wave_
↪input_namelist)
real(wp),          intent(in)  :: grav  ! Acceleration of gravity applied in wave␣
↪generator
real(wp),          intent(in)  :: lscale! Number of length units per meter applied in␣
↪wave generator
                                        ! E.g. if US feet was applied: Lscale = 1/0.
↪3048 = 3.2808
integer,           intent(in)  :: amp   ! Flag to indicate type of provided spectral␣
↪amplitudes
integer,           intent(in)  :: n     ! n as defined in the theory documentation
integer,           intent(in)  :: nh    ! hat(n) as defined in the theory␣
↪documentation
integer,           intent(in)  :: order ! Expansion order applied in wave generator. (
↪<0 for fully non-linear)
real(wp),          intent(in)  :: dk    ! Spacing of k-numbers in x-direction
real(wp),          intent(in)  :: dt    ! Spacing of time steps in the swd file
integer,           intent(in)  :: nsteps ! Total number of time steps to be stored in␣
↪the swd file.
integer,           intent(in)  :: isf    ! Flag to indicate geometric description of␣
↪the sea floor
                                         ! 0: Piecewise linear sea floor
integer,           intent(in)  :: nsf    ! Number of offset points describing the sea␣
↪floor
real(wp),          intent(in)  :: xsf(:) ! X-positions of sea floor points
real(wp),          intent(in)  :: zsf(:) ! Z-positions of sea floor points
!
integer :: nid, i
integer, parameter :: fmt = 100
integer, parameter :: shp = 3
integer, parameter :: nstrip = 0
! We need some C compatible characters
character(kind=c_char, len=:), allocatable :: ccid
character(kind=c_char, len=30) :: cprog
character(kind=c_char, len=20) :: cdate

call input_sanity_checks
```

(continues on next page)

(continued from previous page)

```
open( newunit=self % lu, file=file, access='stream', form='unformatted', &
      status='replace', action='write', convert='little_endian')
! The CONVERT option is an Intel, gfortran, HP and IBM extension.
! Modify this parameter for other compilers if your compiler complains.

!
self % amp = amp
self % nsteps = nsteps
self % it = 0
self % n = n
self % nh = nh
self % nsf = nsf
!
ccid = trim(cid) // c_null_char
nid = len(ccid)
cdate = timestamp() // c_null_char
cprog = trim(prog) // c_null_char
!
write(self % lu) 37.0221_c_float
write(self % lu) int(fmt, c_int)
write(self % lu) int(shp, c_int)
write(self % lu) int(amp, c_int)
write(self % lu) cprog
write(self % lu) cdate
write(self % lu) int(nid, c_int)
write(self % lu) ccid
write(self % lu) real(grav, c_float)
write(self % lu) real(lscale, c_float)
write(self % lu) int(nstrip, c_int)
write(self % lu) int(nsteps, c_int)
write(self % lu) real(dt, c_float)
write(self % lu) int(order, c_int)
write(self % lu) int(n, c_int)
write(self % lu) int(nh, c_int)
write(self % lu) real(dk, c_float)
write(self % lu) int(isf, c_int)
write(self % lu) int(nsf, c_int)
do i = 1, nsf
    write(self % lu) real(xsf(i), c_float)
end do
do i = 1, nsf
    write(self % lu) real(zsf(i), c_float)
end do

contains

  function timestamp() result(timestr)
    ! Return a string with the current time in the form YYYY-MM-DD hh-mm-ss
    character (len = 19) :: timestr

    integer :: y, m, d, h, n, s
    integer :: values(8)

    call date_and_time (values = values)

    y = values(1)
```

(continues on next page)

(continued from previous page)

```
    m = values(2)
    d = values(3)
    h = values(5)
    n = values(6)
    s = values(7)

    write (timestr, '(i4.4,a,i2.2,a,i2.2,a,i2.2,a,i2.2,a,i2.2)' ) &
         y,'-',m,'-',d,' ',h,':',n,':',s

  end function timestamp
!
  subroutine input_sanity_checks
    if (nsf > 0) then
        ! Sea floor is completely submerged...
        if (any(zsf(1:nsf) >= 0.0_wp)) then
            print*, "ERROR: from swd_write_shape_3 :: init"
            print*, "All zsf(:) should be negative"
            print*, "zsf(1:nsf) = ", zsf(1:nsf)
            stop
        end if
        ! Sea floor x-range is in domain [0, 2pi/dk]
        if (abs(xsf(1)) > epsilon(xsf)) then
            print*, 'ERROR from swd_write_shape_3 :: init'
            print*, 'xsf(1) should be 0.0'
            print*, 'xsf(1) = ', xsf(1)
            stop
        end if
        if (abs(xsf(nsf) - 2*pi/dk) > 1.0e-4_wp * 2*pi/dk) then
            print*, 'ERROR from swd_write_shape_3 :: init'
            print*, 'xsf(nsf) should be 2*pi/dk'
            print*, 'xsf(nsf) = ', xsf(nsf)
            stop
        end if
        ! Sea floor x-values should be monotonic increasing
        do i = 2, nsf
            if (xsf(i-1) >= xsf(i)) then
                print*, 'ERROR from swd_write_shape_3 :: init'
                print*, 'xsf(1:nsf) should be monotonic increasing'
                print*, 'xsf(1:nsf) = ', xsf(1:nsf)
                stop
            end if
        end do
    end if
    if (nsf==0 .and. nh/=0) then
        print*, 'ERROR from swd_write_shape_3 :: init'
        print*, 'nh should be 0 if nsf is 0'
        print*, 'nsf, nh = ', nsf, nh
        stop
    end if
    if (nsf>0 .and. nh<=0) then
        print*, 'ERROR from swd_write_shape_3 :: init'
        print*, 'nh should be positive if nsf is positive'
        print*, 'nsf, nh = ', nsf, nh
        stop
    end if

  end subroutine input_sanity_checks
```

(continues on next page)

(continued from previous page)

```
end subroutine init

!================================================================================

subroutine update(self, h, ht, c, ct, ch, cht)
! Output of temporal functions as defined in the shape 3 class
class(swd_write_shape_3), intent(inout) :: self   ! Object to update
complex(wp),           intent(in) :: h(0:)   ! h(0:n) temporal amp
complex(wp),           intent(in) :: ht(0:)  ! dh/dt(0:n) temporal amp
complex(wp), optional, intent(in) :: c(0:)   ! c(0:n) temporal amp
complex(wp), optional, intent(in) :: ct(0:)  ! dc/dt(0:n) temporal amp
complex(wp), optional, intent(in) :: ch(0:)  ! hat(c)(0:nh) temporal amp
complex(wp), optional, intent(in) :: cht(0:) ! d hat(c)/dt(0:nh) temporal amp
!
self % it = self % it + 1
call dump(h, self % n)
call dump(ht, self % n)
if (self % amp < 3) then
    call dump(c, self % n)
    call dump(ct, self % n)
    if (self % nsf > 1) then
        call dump(ch, self % nh)
        call dump(cht, self % nh)
    end if
end if

contains

    subroutine dump(array, n)
        complex(wp), intent(in) :: array(:)
        integer,     intent(in) :: n
        integer :: j
        do j = 0, n
            write(self % lu) cmplx(array(j), kind=c_float)
        end do
    end subroutine dump
!
end subroutine update

!================================================================================

subroutine close(self)
class(swd_write_shape_3), intent(inout) :: self   ! Object to update
!
if (self % it /= self % nsteps) then
    print*, "WARNING: from swd_write_shape_3 :: close"
    print*, "Specified number of time steps = ", self % nsteps
    print*, "Number of provided time steps = ", self % it
end if
!
close(self % lu)
!
end subroutine close

!================================================================================
```

(continues on next page)

(continued from previous page)

```
end module swd_write_shape_3_def
```

swd_write_shape_4_or_5.f90

```
module swd_write_shape_4_or_5_def

use, intrinsic :: iso_c_binding,   only: c_char, c_int, c_float, c_null_char

implicit none
private

! Please adjust wp to the working precision applied in the wave generator
integer, parameter :: wp = kind(1.0d0) ! (double precision)


! This module provides a class for writing SWD files for shape 4 and 5.
!
! Written by Jens Bloch Helmers, August, 1. 2019
!
!-------------------------------------------------------------------------------

!###############################################################################
!
!              B E G I N    P U B L I C    Q U A N T I T I E S
!
!-------------------------------------------------------------------------------
!
public :: swd_write_shape_4_or_5
!
!-------------------------------------------------------------------------------
!
!                E N D    P U B L I C    Q U A N T I T I E S
!
!###############################################################################

type swd_write_shape_4_or_5
    integer :: amp    ! Type of spectral amplitudes to be provided
    integer :: lu     ! Unit number associated with swd file
    integer :: nsteps ! Expected number of time steps
    integer :: it     ! Current time steps
    integer :: nx     ! Number of points in physical space (x-dir)
    integer :: ny     ! Number of points in physical space (y-dir)
    integer :: nx_swd ! nx in swd theory description (=int(nx/2))
    integer :: ny_swd ! ny in swd theory description (=int(ny/2))
contains
    procedure :: init    ! Open the swd file and write the header section
    procedure :: update  ! Add spectral data at current time step
    procedure :: update_fft  ! Alternative output method based on FFTW r2c data
    procedure :: close   ! close the file
end type swd_write_shape_4_or_5

contains

!===============================================================================
```

(continues on next page)

(continued from previous page)

```
subroutine init(self, file, prog, cid, grav, Lscale, amp, &
                nx, ny, order, dkx, dky, dt, nsteps, d)
class(swd_write_shape_4_or_5),  intent(out) :: self  ! Object to create
character(len=*),  intent(in)  :: file  ! Requested name for new SWD file
character(len=*),  intent(in)  :: prog  ! Name of actual wave generator including␣
↪version number
character(len=*),  intent(in)  :: cid   ! Text describing actual input parameters to␣
↪the
                                        ! wave generator. Hint: write(cid, nml=wave_
↪input_namelist)
real(wp),          intent(in)  :: grav  ! Acceleration of gravity applied in wave␣
↪generator
real(wp),          intent(in)  :: lscale! Number of length units per meter applied in␣
↪wave generator
                                        ! E.g. if US feet was applied: Lscale = 1/0.
↪3048 = 3.2808
integer,           intent(in)  :: amp   ! Flag to indicate type of provided spectral␣
↪amplitudes
integer,           intent(in)  :: nx    ! nx as defined in the theory documentation
integer,           intent(in)  :: ny    ! ny as defined in the theory documentation
integer,           intent(in)  :: order ! Expansion order applied in wave generator. (
↪<0 for fully non-linear)
real(wp),          intent(in)  :: dkx   ! Spacing of k-numbers in x-direction
real(wp),          intent(in)  :: dky   ! Spacing of k-numbers in y-direction
real(wp),          intent(in)  :: dt    ! Spacing of time steps in the swd file
integer,           intent(in)  :: nsteps ! Total number of time steps to be stored in␣
↪the swd file.
real(wp),          intent(in), optional  :: d ! Water depth (if not present, deep␣
↪water (shape 4) is assumed)
!
integer :: nid
integer, parameter :: fmt = 100
integer :: shp
integer, parameter :: nstrip = 0
! We need some C compatible characters
character(kind=c_char, len=:), allocatable :: ccid
character(kind=c_char, len=30) :: cprog
character(kind=c_char, len=20) :: cdate
!
open( newunit=self % lu, file=file, access='stream', form='unformatted', &
      status='replace', action='write', convert='little_endian')
! The CONVERT option is an Intel, GFortran, HP and IBM extension.
! Modify this parameter for other compilers if your compiler complains.

!
self % amp = amp
self % nsteps = nsteps
self % it = 0
self % nx = nx
self % ny = ny
self % nx_swd = nx/2
self % ny_swd = ny/2
!
shp = 4
if (present(d)) then
   if (d > 0.0_wp) then
```

(continues on next page)

(continued from previous page)

```
      shp = 5
   end if
end if
!
nid = len_trim(cid) + 1  ! one extra for C null character
allocate(character(len=nid) :: ccid)
ccid = trim(cid) // c_null_char
cdate = timestamp() // c_null_char
cprog = trim(prog) // c_null_char
!
write(self % lu) 37.0221_c_float
write(self % lu) int(fmt, c_int)
write(self % lu) int(shp, c_int)
write(self % lu) int(amp, c_int)
write(self % lu) cprog
write(self % lu) cdate
write(self % lu) int(nid, c_int)
write(self % lu) ccid(:nid)
write(self % lu) real(grav, c_float)
write(self % lu) real(lscale, c_float)
write(self % lu) int(nstrip, c_int)
write(self % lu) int(nsteps, c_int)
write(self % lu) real(dt, c_float)
write(self % lu) int(order, c_int)
write(self % lu) int(self % nx_swd, c_int)
write(self % lu) int(self % ny_swd, c_int)
write(self % lu) real(dkx, c_float)
write(self % lu) real(dky, c_float)
if (shp == 5) then
   write(self % lu) real(d, c_float)
end if

contains

  function timestamp() result(timestr)
    ! Return a string with the current time in the form YYYY-MM-DD hh-mm-ss
    character (len = 19) :: timestr

    integer :: y, m, d, h, n, s
    integer :: values(8)

    call date_and_time (values = values)

    y = values(1)
    m = values(2)
    d = values(3)
    h = values(5)
    n = values(6)
    s = values(7)

    write (timestr, '(i4.4,a,i2.2,a,i2.2,a,i2.2,a,i2.2,a,i2.2)' ) &
         y,'-',m,'-',d,' ',h,':',n,':',s

  end function timestamp
!
end subroutine init
```

(continues on next page)

(continued from previous page)

```
!================================================================================

subroutine update(self, h, ht, c, ct)
! Output of temporal functions as defined in the shape 4 and 5 classes
class(swd_write_shape_4_or_5), intent(inout) :: self   ! Object to update
complex(wp),           intent(in) :: h(-self % ny_swd:, 0:)   ! h(-ny_swd:ny_swd,␣
↪0:nx_swd) temporal amp
complex(wp),           intent(in) :: ht(-self % ny_swd:, 0:)  ! dh/dt(-ny_swd:ny_swd,␣
↪0:nx_swd) temporal amp
complex(wp), optional, intent(in) :: c(-self % ny_swd:, 0:)   ! c(-ny_swd:ny_swd,␣
↪0:nx_swd) temporal amp
complex(wp), optional, intent(in) :: ct(-self % ny_swd:, 0:)  ! dc/dt(-ny_swd:ny_swd,␣
↪0:nx_swd) temporal amp
!
self % it = self % it + 1
call dump(h)
call dump(ht)
if (self % amp < 3) then
    call dump(c)
    call dump(ct)
end if

contains

    subroutine dump(array)
        complex(wp), intent(in) :: array(-self % ny_swd:, 0:)
        integer :: jx, jy
        !-------------------------------------------------------------------------
        ! Explicit loops to demonstrate element ordering for all languages.
        !-------------------------------------------------------------------------
        do jx = 0, self % nx_swd
            do jy = -self % ny, self % ny_swd
                write(self % lu) cmplx(array(jy, jx), kind=c_float)
            end do
        end do
    end subroutine dump
!
end subroutine update

!================================================================================

subroutine update_fft(self, h_fft, ht_fft, c_fft, ct_fft)
! Alternative to using the 'update' routine.
! Output of temporal functions as defined in the shape 4 and 5 classes
! Input arrays are in the form returned by FFTW r2c 2D transforms
class(swd_write_shape_4_or_5), intent(inout) :: self   ! Object to update
complex(wp),           intent(in) :: h_fft(:, :)   ! h_fft(1:nx/2+1, 1:ny) temporal␣
↪amp
complex(wp),           intent(in) :: ht_fft(:, :)  ! dh/dt(1:nx/2+1, 1:ny) temporal␣
↪amp
complex(wp), optional, intent(in) :: c_fft(:, :)   ! c(1:nx/2+1, 1:ny) temporal amp
complex(wp), optional, intent(in) :: ct_fft(:, :)  ! dc/dt(1:nx/2+1, 1:ny) temporal␣
↪amp
!
self % it = self % it + 1
call dump_fft(h_fft)
call dump_fft(ht_fft)
```

(continues on next page)

(continued from previous page)

```
if (self % amp < 3) then
    call dump_fft(c_fft)
    call dump_fft(ct_fft)
end if

contains

  subroutine dump_fft(array)
    complex(wp), intent(in) :: array(:,:)
    integer :: jx, jy
    real(wp) :: sc
    !-----------------------------------------------------------------------------
    ! Explicit loops to demonstrate element ordering for all languages.
    !-----------------------------------------------------------------------------
    do jx = 1, self % nx/2 + 1
       sc = 2.0_wp ! scaling factor between fft and swd format
       if (jx == 1) sc = 1.0_wp ! special case for zero-frequency
       if (mod(self % nx, 2) == 0 .and. jx == self % nx/2 + 1) sc = 1.0_wp ! special␣
↪case for nyquist-frequency for even nx
       ! the first negative freq (-ny_swd)
       if (mod(self % ny, 2) == 0) then
          write(self % lu) cmplx(0.5_wp*sc*conjg(array(jx,self % ny - self % ny/2 +␣
↪1)), kind=c_float)
       else
          write(self % lu) cmplx(sc*conjg(array(jx,self % ny - self % ny/2 + 1)),␣
↪kind=c_float)
       end if
       ! all other negative freqs up to -1
       do jy = self % ny - self % ny/2 + 2, self % ny
          write(self % lu) cmplx(sc*conjg(array(jx,jy)), kind=c_float)
       end do
       ! 0-freq and all positive freqs
       do jy = 1, self % ny - self % ny/2
          write(self % lu) cmplx(sc*conjg(array(jx,jy)), kind=c_float)
       end do
       ! if even ny, nyquist freq must be on positive side as well
       if (mod(self % ny, 2) == 0) then
          write(self % lu) cmplx(0.5_wp*sc*conjg(array(jx,self % ny - self % ny/2 +␣
↪1)), kind=c_float)
       end if
    end do
  end subroutine dump_fft
!
end subroutine update_fft

!================================================================================

subroutine close(self)
class(swd_write_shape_4_or_5), intent(inout) :: self   ! Object to update
!
if (self % it /= self % nsteps) then
    print*, "WARNING: from swd_write_shape_4_or_5 :: close"
    print*, "Specified number of time steps = ", self % nsteps
    print*, "Number of provided time steps = ", self % it
end if
!
close(self % lu)
```

(continues on next page)

(continued from previous page)

```
!
end subroutine close

!===============================================================================

end module swd_write_shape_4_or_5_def
```

swd_write_shape_6.f90

```
module swd_write_shape_6_def

use, intrinsic :: iso_c_binding,   only: c_char, c_int, c_float, c_null_char

implicit none
private

! Please adjust wp to the working precision applied in the wave generator
integer, parameter :: wp = kind(1.0d0) ! (double precision)


! This module provides a class for writing SWD files for shape 6.
!
! Written by Jens Bloch Helmers, November, 17. 2019
!
!-------------------------------------------------------------------------------

!###############################################################################
!
!              B E G I N    P U B L I C    Q U A N T I T I E S
!
!-------------------------------------------------------------------------------
!
public :: swd_write_shape_6
!
!-------------------------------------------------------------------------------
!
!                E N D    P U B L I C    Q U A N T I T I E S
!
!###############################################################################

type :: swd_write_shape_6
    integer :: amp    ! Type of spectral amplitudes to be provided
    integer :: lu     ! Unit number associated with swd file
    integer :: nsteps ! Expected number of time steps
    integer :: it     ! Current time steps
    integer :: n      ! Number of spectral components in x-dir
    integer :: nh     ! Number of auxilary shapes (hat(n) in theory manual)
    integer :: nsf    ! Number of sea floor points
contains
    procedure :: build   ! Build the file in one go....
end type swd_write_shape_6

real(wp), parameter :: pi = 3.
↪14159265358979323846264338327950288419716939937510582097494_wp
```

(continues on next page)

(continued from previous page)

```
contains

!================================================================================

subroutine build(self, file, prog, cid, grav, lscale, depth, nwaves, awaves, kwaves, &
                 dwaves, pwaves)
class(swd_write_shape_6),  intent(out) :: self  ! Object to create
character(len=*),  intent(in)  :: file   ! Requested name for new SWD file
character(len=*),  intent(in)  :: prog   ! Name of actual wave generator including␣
↪version number
character(len=*),  intent(in)  :: cid    ! Text describing actual input parameters to␣
↪the
                                         ! wave generator. Hint: write(cid, nml=wave_
↪input_namelist)
real(wp),          intent(in)  :: grav   ! Acceleration of gravity applied in wave␣
↪generator
real(wp),          intent(in)  :: lscale ! Number of length units per meter applied␣
↪in wave generator
                                         ! E.g. if US feet was applied: Lscale = 1/0.
↪3048 = 3.2808
real(wp),          intent(in)  :: depth  ! Constant water depth (<0 indicates␣
↪infinite depth)
integer,           intent(in)  :: nwaves ! n as defined in the theory documentation
real(wp),          intent(in)  :: awaves(:) ! Single wave amplitude for each component
real(wp),          intent(in)  :: kwaves(:) ! Wave number for each component
real(wp),          intent(in)  :: dwaves(:) ! Direction gamma as defined in doc. for␣
↪each wave component (rad)
real(wp),          intent(in)  :: pwaves(:) ! Phase delta as defined in doc. (rad)
!
integer :: nid, i
integer, parameter :: fmt = 100
integer, parameter :: shp = 6
integer, parameter :: amp = 1
integer, parameter :: order = 0
integer, parameter :: nstrip = 0
integer, parameter :: nsteps = 0
real(wp), parameter :: dt = -1.0_wp
! We need some C compatible characters
character(kind=c_char, len=:), allocatable :: ccid
character(kind=c_char, len=30) :: cprog
character(kind=c_char, len=20) :: cdate
!
open( newunit=self % lu, file=file, access='stream', form='unformatted', &
      status='replace', action='write' )
!
!
nid = len_trim(cid) + 1  ! one extra for C null character
allocate(character(len=nid) :: ccid)
ccid = trim(cid) // c_null_char
cdate = timestamp() // c_null_char
cprog = trim(prog) // c_null_char
!
write(self % lu) 37.0221_c_float
write(self % lu) int(fmt, c_int)
write(self % lu) int(shp, c_int)
write(self % lu) int(amp, c_int)
```

(continues on next page)

(continued from previous page)

```
write(self % lu) cprog
write(self % lu) cdate
write(self % lu) int(nid, c_int)
write(self % lu) ccid(:nid)
write(self % lu) real(grav, c_float)
write(self % lu) real(lscale, c_float)
write(self % lu) int(nstrip, c_int)
write(self % lu) int(nsteps, c_int)
write(self % lu) real(dt, c_float)
write(self % lu) int(order, c_int)
write(self % lu) int(nwaves, c_int)
write(self % lu) real(depth, c_float)
do i = 1, nwaves
    write(self % lu) real(awaves(i), c_float)
    write(self % lu) real(kwaves(i), c_float)
    write(self % lu) real(dwaves(i), c_float)
    write(self % lu) real(pwaves(i), c_float)
end do
close(self % lu)
!
contains

  function timestamp() result(timestr)
    ! Return a string with the current time in the form YYYY-MM-DD hh-mm-ss
    character (len = 19) :: timestr

    integer :: y, m, d, h, n, s
    integer :: values(8)

    call date_and_time (values = values)

    y = values(1)
    m = values(2)
    d = values(3)
    h = values(5)
    n = values(6)
    s = values(7)

    write (timestr, '(i4.4,a,i2.2,a,i2.2,a,i2.2,a,i2.2,a,i2.2)' ) &
         y,'-',m,'-',d,' ',h,':',n,':',s

  end function timestamp
!
end subroutine build

!===============================================================================

end module swd_write_shape_6_def
```

Note: If your *wave generator* applies a different numerical precision than `double precision` you need to adjust the kind parameter `wp` defined in the beginning of the source code.

Assuming your wave generator will support the spectral shape_X formulation where X is one of the shape models as defined in the *theory* section, you may write something like this in your *wave generator*.

```
program my_wave_generator
...
use swd_write_shape_X_def, only: swd_write_shape_X   ! Replace X with the index of␣
↪the actual model
...
type(swd_write_shape_X) :: swd_X
...
swd_X % init('my_wave.swd', nx, dk, ...)  ! All time independent swd-format␣
↪parameters for shape X
...
do (temporal integration loop)
    ...
    swd_X % update(h, ht, c, ct)   ! Relevant spectral amplitudes for this time step␣
↪for shape X
end do
swd_X % close()
...
program my_wave_generator
```

If your *wave generator* also supports another spectral formulation `shp=y`, just add a similar `swd_y` variable of `type(swd_write_shape_y)` and provide the relevant parameters for that model too.

Following this approach, Fortran source code dealing with the low level binary C-stream output for the various spectral formulations are listed in this table

C / C++ / Python

Writing C-streams using these programming languages are straight forward. The logical ordering is the same as for the fortran examples given above. An example from open source is the raschii Python package for making non-linear regular waves.

Note: Any 2-dimensional arrays should be written using the Fortran element ordering described in the *SWD format* description.

7.2 Application programs

There are no restrictions on programming styles applied in *application programs*. However, the SWD API is object oriented in order to simplify its application. An instance of a specific class takes care of all related kinematics and swd-file operations. The application programmer do not consider I/O operations at all.

The API interface to be applied in *application programs* is basically the very same for all of the following programming languages: Fortran-2008, C/C++ and Python-2/3. Fully working *application programs* demonstrating most features of the API in all of these languages follows

7.2.1 application_swd_api.py

```
from __future__ import print_function   # Supports print() in Python 2.7 too

from spectral_wave_data import SpectralWaveData, SwdError  # Supports Python 2.x and
↪3.x


# First we define some constructor parameters...

# Relate the application and SWD coordinate systems
x0 = 0.0; y0 = 0.0; t0 = 0.0; beta = 0.0

# Optional: impl=0 means apply recommended implementation based on header of swd file.
impl = 0

# Optional: Density of water needed for pressure calculations
rho = 1025.0

# Optional: number of spectral components to apply in calculations.
nsumx = -1; nsumy = -1   # Negative: apply all components from swd-file.

# Optional: Select temporal interpolation scheme
ipol = 0   # C^2 continous

# Optional: Select expansion order for calculating kinematics above z=0
norder = 0   # Apply same order as specified in SWD file

# Optional: Control handling of zero-frequency components in SWD file
dc_bias = False   # Suppress contributions from zero-frequency

# Constructor
try:
    swd = SpectralWaveData('stokesTest_shallowWater.swd', x0, y0, t0, beta,
                           rho=rho, nsumx=nsumx, nsumy=nsumy, impl=impl,
                           ipol=ipol, norder=norder, dc_bias=dc_bias)
except SwdError as e:
    print(str(e))
    # Do whatever necessary...
    raise

# Time domain simulation...
t = 0.0;  dt = 0.1;  tmax = 1.0;
while t < tmax:

    # Tell the swd object current application time...
    swd.update_time(t)

    # Application coordinate where we need kinematics...
    x = 4.3;  y = 5.4;  z = -1.7;

    # Current set of provided kinematic quantities...

    # Velocity potential
    print("phi = ", swd.phi(x, y, z))

    # Stream function
```

(continues on next page)

(continued from previous page)

```
    print("varphi = ", swd.stream(x, y, z))

    # partial(phi) / partial t
    print("phi_t = ", swd.phi_t(x, y, z))

    # particle velocity
    print("grad_phi = ", swd.grad_phi(x, y, z))

    # 2nd order gradients of phi
    print("phi_xx, phi_xy, phi_xz, phi_yy, phi_yz, phi_zz = ", swd.grad_phi_2nd(x, y,␣
↪z))

    # Local (Euler) acceleration
    print("acc_euler = ", swd.acc_euler(x, y, z))

    # particle acceleration
    print("acc_particle = ", swd.acc_particle(x, y, z))

    # surface elevation
    print("zeta = ", swd.elev(x, y))

    # local time derivative of wave elevation
    print("\partial(zeta)/\partial t = ", swd.elev_t(x, y))

    # spatial gradient of wave elevation
    print("grad_zeta = ", swd.grad_elev(x, y))

    # 2nd order gradients of wave elevation
    print("zeta_xx, zeta_xy, zeta_yy = ", swd.grad_elev_2nd(x, y))

    # Total pressure
    print("pressure = ", swd.pressure(x, y, z))

    # Vertical distance from z=0 to sea floor (<0 if infinite)
    print("local depth = ", swd.bathymetry(x, y))

    # Unit normal vector of sea floor into the ocean at (x,y)
    print("nx, ny, nz = ", swd.bathymetry_nvec(x, y))

    # For a specific (x,y,z) at this t, return a CSV file on how particle velocity,
    # elevation and pressure converge as a function of number of spectral components
    swd.convergence(x, y, z, 'dump.csv')

    t += dt

# To save storage for an interesting event you may create a new SWD file
# containing only the time steps within a specified time window.
swd.strip(tmin=100.0, tmax=200.0, file_swd='my_new.swd')

# ============================================================
# The meth swd.get(name) returns the value of parameter 'name'
# from the swd file. Three examples are given below...
# ============================================================

# Extract the cid string from the SWD file
# (contains typical the content of the input file applied in the wave generator)
print("cid = ", swd.get('cid'))
```

(continues on next page)

(continued from previous page)

```
# The shp parameter from the swd file
print("shp = ", swd.get('shp'))

# Time step in SWD file
print("dt = ", swd.get('dt'))

# Name of actual implementation class
print("cls_name = ", swd.get('class'))

# Close SWD file and free related memory
swd.close()
```

7.2.2 application_swd_api.cpp

```
#include <string>
#include <iostream>
#include <stdlib.h> /* for exit */

#include "SpectralWaveData.h"    // Version for C++

int main()
{

// Actual swd file
std::string file_swd = "my_wave.swd";

// Constructor parameters to relate the application and SWD coordinate systems
double x0 = 0.0, y0 = 0.0, t0 = 0.0, beta = 0.0;

// impl=0: apply recommended implementation based on header of swd file.
int impl = 0;

// Density of water needed for pressure calculations
double rho = 1025.0;

// Number of spectral components to apply in calculations.
int nsumx = -1, nsumy = -1;   // Negative: apply all components from swd-file.

// Optional: Select temporal interpolation scheme
int ipol = 0;   // C^2 continous

// Optional: Select expansion order for kinematics above z=0
int norder = 0;   // Apply same order as specified in SWD file

// Optional: Control handling of zero-frequency components in SWD file
bool dc_bias = false;   // Suppress contributions from zero-frequency

//=============
// Constructor
//=============
SpectralWaveData swd = SpectralWaveData(file_swd, x0, y0, t0, beta,
                                        rho, nsumx, nsumy, impl, ipol,
                                        norder, dc_bias);
```

(continues on next page)

(continued from previous page)

```
//=====================================
// Time domain simulation...
//=====================================
double t = 0.0, dt = 0.1, tmax = 2.0;
while (t < tmax) {

    // Tell the swd object current application time...
    try {
        swd.UpdateTime(t);
    } catch (SwdInputValueException& e) {  //Could be t > tmax from file.
        std::cout << typeid(e).name() << std::endl << e.what() << std::endl;
        // If we will try again with a new value of t
        // we first need to call: swd.ExceptionClear()
        exit(EXIT_FAILURE);  // In this case we just abort.
    }

    // Application coordinate where we need kinematics...
    double x = 4.3, y = 5.4, z = -1.7;

    // Current set of provided kinematic quantities...

    // Velocity potential
    double wave_potential = swd.Phi(x, y, z);

    // Stream function
    double stream_function = swd.Stream(x, y, z);

    // partial(phi) / partial t
    double wave_ddt_potential = swd.DdtPhi(x, y, z);

    // Particle velocity  (vector_swd is a (x,y,z) struct)
    vector_swd wave_velocity = swd.GradPhi(x, y, z);

    // 2nd order gradients of potential  (vector_2nd_phi_swd is a (xx,xy,xz,yy,yz,zz)␣
↪struct)
    vector_2nd_phi_swd  grad_phi_2nd = swd.GradPhi2nd(x, y, z);

    // Local acceleration
    vector_swd euler_acceleration = swd.AccEuler(x, y, z);

    // Particle acceleration
    vector_swd particle_acceleration = swd.AccParticle(x, y, z);

    // Wave elevation
    double wave_elevation = swd.Elev(x, y);

    // Local time derivative of wave elevation
    double wave_ddt_elevation = swd.DdtElev(x, y);

    // Spatial gradient of wave elevation
    vector_swd wave_grad_elevation = swd.GradElev(x, y);

    // 2nd order gradients of elevation  (vector_2nd_elev_swd is a (xx,xy,yy) struct)
    vector_2nd_elev_swd grad_elev_2nd = swd.GradElev2nd(x, y);

    // Total pressure
    double wave_pressure = swd.Pressure(x, y, z);
```

(continues on next page)

(continued from previous page)

```
    // Distance from z=0 to sea floor (<0 if infinite)
    double local_depth = swd.Bathymetry(x, y);

    // Unit normal vector of sea floor into the ocean at(x, y)
    vector_swd  nvec_sf = swd.BathymetryNvec(x, y);

    // For a specific(x, y, z) at this t, return a CSV file on how particle velocity,
    // elevation and pressure converge as a function of number of spectral components
    swd.Convergence(x, y, z, "dump.csv");

    t += dt;
}

// To save storage for an interesting event you may create a new SWD file
// containing only the time steps within a specified time window.
// In this case we only care about the time interval [100.0, 200.0]
swd.Strip(100.0, 200.0, "my_new.swd");

/*
============================================================
There are 4 methods returning metadata from object...
  swd.GetChr(swd, name)
  swd.GetInt(swd, name)
  swd.GetBool(swd, name)
  swd.GetReal(swd, name)
where name is the requested parameter.
Five examples are given below...
============================================================
*/

// Extract the cid string from the SWD file
// (contains typical the content of the input file applied in the wave generator)
std::string cid = swd.GetChr("cid");

// Name of actual implementation class
std::string cls_name = swd.GetChr("class");

// Extract shp parameter from object
int shp = swd.GetInt("shp");

// Extract dc_bias parameter from object
bool dc_bias2 = swd.GetBool("dc_bias");

// Extract time step as applied in SWD file
double dt_swd_file = swd.GetReal("dt");

// Automatic destruction
return 0;
}
```

7.2.3 application_swd_api.c

```
#include <stdbool.h> /* bool */
#include <stdio.h> /* for fprintf and stderr */
#include <stdlib.h> /* for exit */

#include "spectral_wave_data.h"    // Namespace convention: swd_api_*

int main() {

// First we define some constructor parameters...

// Name of actual SWD-file
char *file_swd = "StokesTest_shallowWater.swd";

// Relate the application and SWD coordinate systems
double x0 = 0.0, y0 = 0.0, t0 = 0.0, beta = 0.0;

// impl=0: apply recommended implementation based on header of swd file.
int impl = 0;

// Number of spectral components to apply in calculations.
int nsumx = -1, nsumy = -1;   // Negative: apply all components from the swd-file.

// Density of water needed for pressure calculations
double rho = 1025.0;

// A pointer to the object containing the SWD class
void *swd;

// Select temporal interpolation scheme
int ipol = 0;   // C^2 continous

// Select expansion order for kinematics above z=0
int norder = 0;   // Apply same order as specified in SWD file

// Control handling of zero-frequency components in SWD file
bool dc_bias = false;   // Suppress contributions from zero-frequency

// Constructor
swd = swd_api_allocate(file_swd, x0, y0, t0, beta, rho, nsumx, nsumy,
                       impl, ipol, norder, dc_bias);
if (swd_api_error_raised(swd)) {
    fprintf(stderr, swd_api_error_get_msg(swd));
    exit(EXIT_FAILURE); /* indicate failure.*/
}

// Time simulation loop...
double t = 0.0, dt = 0.1, tmax = 2.0;
while (t < tmax) {
    // Tell the swd object current application time...
    swd_api_update_time(swd, t);
    if (swd_api_error_raised(swd)) {   // t is too big...
        fprintf(stderr, swd_api_error_get_msg(swd));
        exit(EXIT_FAILURE);
    }
```

(continues on next page)

(continued from previous page)

```
    // Application coordinate where we need kinematics...
    double x = 4.3, y = 5.4, z = -1.7;

    // Current set of provided kinematic quantities...

    // Velocity potential
    double phi = swd_api_phi(swd, x, y, z);

    // Stream function
    double varphi =  swd_api_stream(swd, x, y, z);

    // partial(phi) / partial t
    double phi_t = swd_api_phi_t(swd, x, y, z);

    // Particle velocity  (vector_swd is a (x,y,z) struct)
    vector_swd  grad_phi = swd_api_grad_phi(swd, x, y, z);

    // 2nd order gradients of potential  (vector_2nd_phi_swd is a (xx,xy,xz,yy,yz,zz)␣
↪struct)
    vector_2nd_phi_swd  grad_phi_2nd = swd_api_grad_phi_2nd(swd, x, y, z);

    // Local (Euler) acceleration
    vector_swd  acc_euler = swd_api_acc_euler(swd, x, y, z);

    // Particle acceleration
    vector_swd  acc_particle = swd_api_acc_particle(swd, x, y, z);

    // Wave elevation
    double elev = swd_api_elev(swd, x, y);

    // Local time derivative of wave elevation
    double elev_t = swd_api_elev_t(swd, x, y);

    // Spatial gradient of wave elevation
    vector_swd  grad_elev = swd_api_grad_elev(swd, x, y);

    // 2nd order gradients of elevation  (vector_2nd_elev_swd is a (xx,xy,yy) struct)
    vector_2nd_elev_swd  grad_elev_2nd = swd_api_grad_elev_2nd(swd, x, y);

    // Total pressure
    double wave_pressure = swd_api_pressure(swd, x, y, z);

    // Vertical distance from z=0 to sea floor (<0 if infinite)
    double local_depth = swd_api_bathymetry(swd, x, y);

    // Unit normal vector of sea floor into the ocean at(x, y)
    vector_swd  nvec_sf = swd_api_bathymetry_nvec(swd, x, y);

    // For a specific(x, y, z) at this t, return a CSV file on how particle velocity,
    // elevation and pressure converge as a function of number of spectral components
    swd_api_convergence(swd, x, y, z, "dump.csv");
    if (swd_api_error_raised(swd)) { /* in rare cases you could not open the CSV file.
↪..*/
        fprintf(stderr, swd_api_error_get_msg(swd));
        exit(EXIT_FAILURE);
    }
```

(continues on next page)

(continued from previous page)

```
    t += dt;
}

{

// To save storage for an interesting event you may create a new SWD file
// containing only the time steps within a specified time window.
// In this case we only care about the time interval [100.0, 200.0]
swd_api_strip(swd, 100.0, 200.0, "stripped.swd");
if (swd_api_error_raised(swd)) { /* if t=[100.0, 200.0] is not a proper interval...*/
    fprintf(stderr, swd_api_error_get_msg(swd));
    exit(EXIT_FAILURE);
}

/*
============================================================
There are 4 functions returning a value from the swd file...
   swd_api_get_chr(swd, name)
   swd_api_get_int(swd, name)
   swd_api_get_bool(swd, name)
   swd_api_get_real(swd, name)
where name is the requested parameter in the swd file.
Five examples are given below...
============================================================
*/

// Extract the cid string from the SWD file
// (contains typical the content of the input file applied in the wave generator)
char *ciddata = swd_api_get_chr(swd, "cid");

// Name of actual implementation class
char *cls_name = swd_api_get_chr(swd, "class");

// The nsteps parameter from the swd file
int nsteps = swd_api_get_int(swd, "nsteps");

// The nsteps parameter from the swd file
bool dc_bias = swd_api_get_bool(swd, "dc_bias");

// Time step in SWD file
double dt_swd_file = swd_api_get_real(swd, "dt");

}

// Destructor
swd_api_close(swd);

return 0;
}
```

7.2.4 application_swd_api.f90

```
program application_swd_api

use, intrinsic :: iso_fortran_env, only: output_unit

use kind_values, only: wp => kind_swd_interface

use spectral_wave_data_def, only: spectral_wave_data
use spectral_wave_data_allocate_def, only: spectral_wave_data_allocate

implicit none

integer :: norder, impl, ipol, nsumx, nsumy, nx, ny, shp
real(wp) ::  x0, y0, t0, t, dt, tend, beta, x, y, z, rho
logical :: dc_bias
character(len=200) :: file_swd
character(len=:), allocatable :: cid

! Polymorphic object (type unknown at compile time) It can deal with any shape classes
class(spectral_wave_data), allocatable :: swd

! First we define some constructor parameters...

! Name of actual swd file
file_swd = "stokesTest_shallowWater.swd"

! Relate the application and SWD coordinate systems
x0 = 0.0_wp;  y0 = 0.0_wp;  t0 = 0.0_wp;  beta = 0.0_wp

! Optional: impl=0 means apply recommended implementation based on header of swd file.
impl = 0

! Optional: Density of water needed for pressure calculations
rho = 1025.0_wp

! Optional: number of spectral components to apply in calculations.
nsumx = -1;  nsumy = -1   ! Negative: apply all components from swd-file.

! Optional: Select temporal interpolation scheme
ipol = 0   ! C^2 continous

! Optional: For z>0 apply same series expansion as in wave generator
norder = 0

! Optional: Control zero-frequency signals from the wave generator
dc_bias = .false.   ! Do not include zero-frequency signals

!-------------------------------------------------------------------------
! Allocate and construct actual type of swd object...
!-------------------------------------------------------------------------
call spectral_wave_data_allocate(swd, file_swd, x0, y0, t0, beta, rho=rho,  &
                                 nsumx=nsumx, nsumy=nsumy, impl=impl,       &
                                 ipol=ipol, norder=norder, dc_bias=dc_bias)
if (swd % error % raised()) then
    print*, swd % error % get_msg()    ! Stop if any errors occured.
    stop
```

(continues on next page)

(continued from previous page)

```
end if
!------------------------------------------------------------------------------
! Time domain simulation....
!------------------------------------------------------------------------------
tend = 5.0_wp;  dt = 1.0_wp;  t = 0.0_wp
do
    if (t > tend) exit

    ! Tell the swd object current application time...
    call swd % update_time(t)
    if (swd % error % raised()) then
        print*, swd % error % get_msg()    ! Stop if t is out of bounds
        stop
    end if

    ! Application coordinate where we need kinematics...
    x=4.3_wp;  y=5.4_wp;  z=-1.7_wp

    ! Complete set of kinematic functions....

    !  Velocity potential
    print*, 'phi = ', swd % phi(x,y,z)

    !  Stream function
    print*, 'varphi = ', swd % stream(x,y,z)

    !  partial(phi) / partial t
    print*, 'phi_t = ', swd % phi_t(x,y,z)

    ! particle velocity
    print*, 'grad_phi = ', swd % grad_phi(x,y,z)

    ! 2nd order gradients of phi
    print*, 'phi_xx, phi_xy, phi_xz, phi_yy, phi_yz, phi_zz = ', swd % grad_phi_2nd(x,
↪y,z)

    ! Local (Euler) acceleration
    print*, 'acc_euler = ', swd % acc_euler(x,y,z)

    ! Particle acceleration
    print*, 'acc_particle = ', swd % acc_particle(x,y,z)

    ! Surface elevation
    print*, 'elev = ', swd % elev(x,y)

    ! local time derivative of wave elevation
    print*, 'elev_t = ', swd % elev_t(x,y)

    ! spatial gradient of wave elevation
    print*, 'grad_elev = ', swd % grad_elev(x,y)

    ! 2nd order gradients of wave elevation
    print*, 'elev_xx, elev_xy, elev_yy = ', swd % grad_elev_2nd(x,y)

    ! Total pressure
    print*, 'pressure = ', swd % pressure(x,y,z)
```

(continues on next page)

(continued from previous page)

```
    ! Vertical distance from z=0 to sea floor (<0 if infinite)
    print*, 'local depth = ', swd % bathymetry(x,y)

    ! Unit normal vector of sea floor into the ocean at (x,y)
    print*, 'nx, ny, nz = ', swd % bathymetry_nvec(x,y)

    ! For a specific (x,y,z) at this t, return a CSV file on how particle velocity,
    ! elevation and pressure converge as a function of number of spectral components
    call swd % convergence(x, y, z, 'dump.csv')

    t = t + dt
end do

! To save storage for an interesting event you may create a new SWD file
! containing only the time steps within a specified time window.
! In this case we only care about the time interval [100.0, 200.0]
call swd % strip(tmin=100.0_wp, tmax=200.0_wp, file_swd='stripped.swd')

!------------------------------------------------------------------------
! There are 3 methods returning a value from the swd file...
!   swd.get_chr(swd, name)
!   swd.get_int(swd, name)
!   swd.get_real(swd, name)
! where name is the requested parameter in the swd file.
! Three examples are given below...
!------------------------------------------------------------------------

! Extract the cid string from the SWD file
! (contains typical the content of the input file applied in the wave generator)
cid = swd % get_chr("cid")

! The shp parameter from the swd file
shp = swd % get_int("shp")

! Time step in SWD file
dt = swd % get_real("dt")

! Name of actual implementation class
print*, "cls_name = ", swd % get_chr('class')

! Close SWD file and release memory
call swd % close
!
end program application_swd_api
```

7.2.5 Compiling and linking

For using precompiled binaries checkout the PREREQUISITES.md file in the root of the GitHub repository.

Due to applications of object oriented features, only modern Fortran compilers are able to compile the source code. Any ISO standard conforming Fortran-2008 compiler can be applied. In practice you need a very recent Fortran compiler to successfully compile this package. It will successfully compile using gfortran 9.x and Intel 2019/2020 compilers. Older compilers will not work due to lack of ISO standard support.

Regarding C++, C and Fortran applications the source code from this repository is written in ISO-standard Fortran-2008, C and C++. Consequently you may compile and link the source from this repository directly with your appli-

cation if you apply standard conforming compilers. There are Cmake files to demonstrate how to do this for the test applications listed above. These scripts have been tested on Windows-10 and Linux-Ubuntu.

Note: When compiling for C/C++ these compilers must be binary compatible with the actual Fortran compiler. E.g. If you apply GNU compilers for C/C++ you should apply gfortran. If you apply Microsoft/Intel C/C++ compilers you should apply Intel Fortran.

Note: For gfortran you need gfortran-10 if you need the possibility to have several concurrent SWD objects connected to the same SWD file. When the official gfortran-10 soon will be released our precompiled Linux libraries and Python package will be recompiled and released. For Intel compilers this is not an issue.

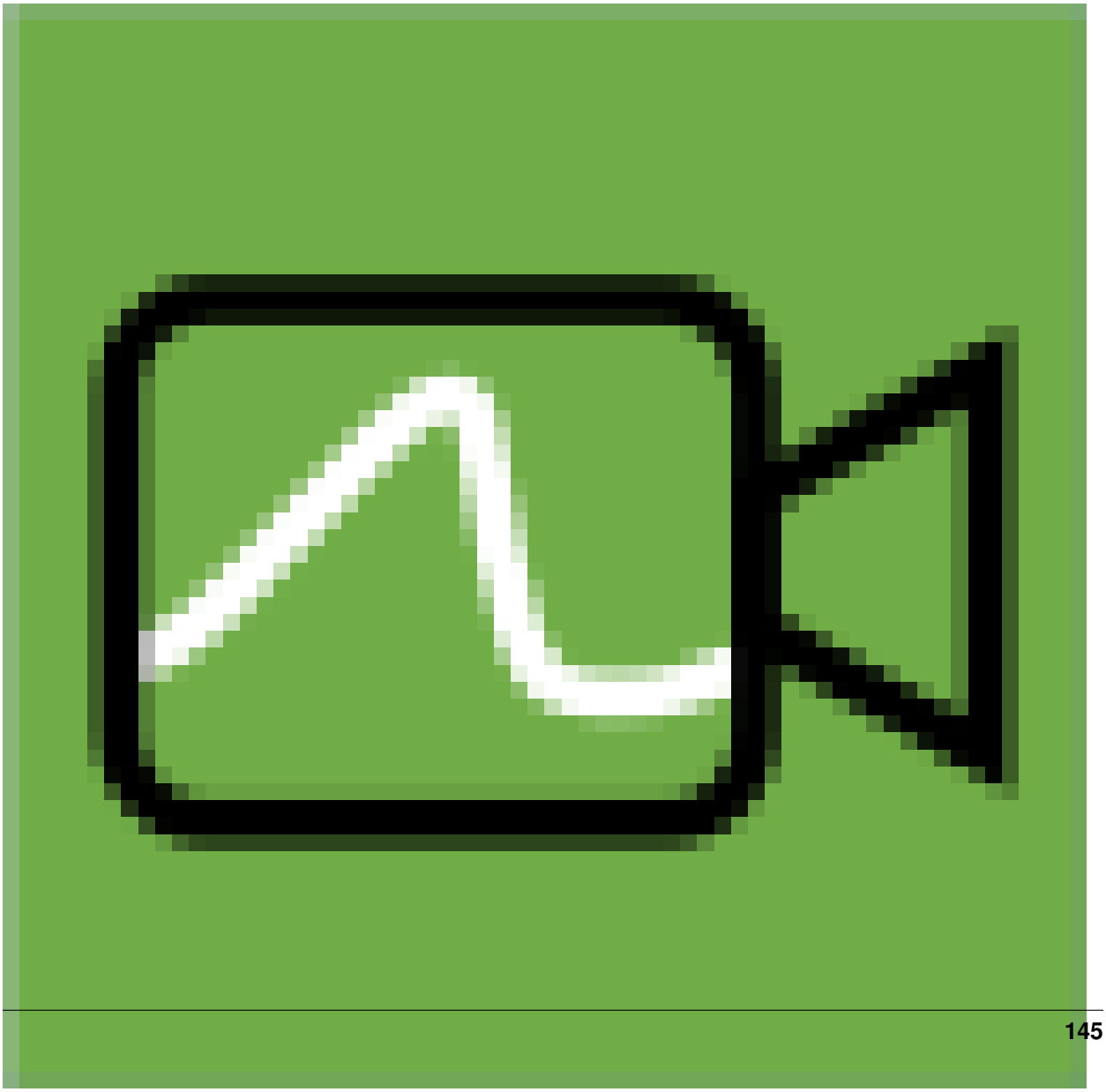
CHAPTER

EIGHT

TOOLS

In this section we describe some auxiliary tools related to **spectral_wave_data**.

8.1 swd2vtk



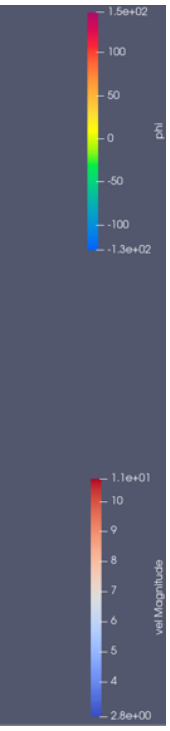
A program **swd2vtk** is available for advanced three-dimensional visualization of SWD wave fields. For a user specified domain a set of VTK-files is

produced. The user selects a set of scalar and vector fields to be generated. VTK-files can be animated or further post-processed using the open-source program ParaView or other tools. More details are described in the *swd2vtk user's guide*

Binary distributions of **swd2vtk**, for Windows and Linux, can be downloaded from the release tab of the GitHub repository **spectral_wave_data**. The binaries are distributed under the MIT-license. Hence, you can basically apply **swd2vtk** as you like for free.

8.1.1 Swd2vtk User's Guide

Contents



In a nutshell

The program **swd2vtk** is provided for making advanced three-dimensional visualization of SWD wave kinematics. The program is a command line tool. The usage is outlined in this diagram below.

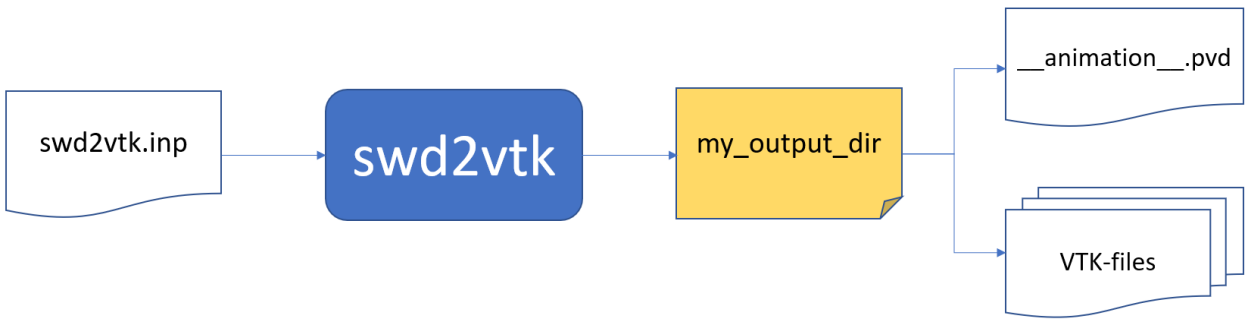


Fig. 2: Based on the contents of a text file, typical called `swd2vtk.inp`, **swd2vtk** will generate a set of VTK files in a user specified subdirectory. In that directory there is a file `__animation__.pvd` containing links to the relevant VTK files for different time steps. This file can be loaded into .e.g. ParaView for making advanced animations.

The input file could be named anything but in this document we apply the name `swd2vtk.inp`. If **swd2vtk** is in your path you can execute it using the command:

```
swd2vtk swd2vtk.inp
```

The VTK files produced are of the new generation format (XML) and not the legacy VTK format. **swd2vtk** may produce *ascii*, *binary* or *raw* VTK files.

Other relevant post-processing tools for the VTK-files are e.g. Mayavi and VTK for python.

Note: The computational effort and file storage requirements for running **swd2vtk** depends significantly on the user specified spatial and temporal resolution.

Example input file

The parameters in this *example file* are explained in the next sections.

swd2vtk input example

Below is an example input file (`swd2vtk.inp`) for **swd2vtk**.

```
&swd2vtk   ! Mandatory namelist tag
!------------------------------------------------------------------------
! Files
!------------------------------------------------------------------------
file_swd = 'stream_wave_H18.50m_T12.79s_D32.00m_Nfft50.swd'
path_vtk = 'vtk_files' ! A subdirectory for all VTK files
vtk_form = 'raw'   ! 'ascii', 'binary' or 'raw'
!------------------------------------------------------------------------
! Time and speed parameters
!------------------------------------------------------------------------
tstart = 0.0   ! Start animation at t_application = tstart
tend = 6.0     ! End animation at t_application = tend
dt = 0.2       ! Animation time step. (Only dt>0 supported)
speedx = 0.0   ! Speed of vtk domain [m/s] along positive x-axis
time_fmt = '(f0.2)'  ! Fortran formatting of animation time
!------------------------------------------------------------------------
! Swd constructor parameters
!------------------------------------------------------------------------
x0_swd = 0.0       ! x0 seed for swd constructor
y0_swd = 0.0       ! y0 seed for swd constructor
t0_swd = 0.0       ! t0 seed for swd constructor
beta_swd = 180.0   ! Angle between x-axis of SWD and application system.
impl_swd = 0       ! Select SWD implementation class (0=automatic)
norder_swd = 0     ! Apply default (shape specific) expansion above z=0.
ipol_swd = 0       ! Temporal interpolation scheme (Default C^2-continous)
nsumx_swd = -1     ! Number of applied spectral components in x-dir (<0 => Apply all)
nsumy_swd = -1     ! Number of applied spectral components in y-dir (<0 => Apply all)
dc_bias_swd = .true. ! Include zero frequency terms if true
rho_swd = 1025.0     ! Density of water  (rho_swd=0.102 => meter water column␣
↪pressure)
!------------------------------------------------------------------------
! Scalar fields to include
!------------------------------------------------------------------------
scl_prs = T     ! Wave pressure
scl_stream = T  ! Stream function
scl_phi = T     ! Potential distribution
scl_phi_t = T   ! d(phi)/dt Earth fixed Euler derivative
scl_phi_xx = T  ! d^2(phi)/dx^2
scl_phi_xy = T  ! d^2(phi)/dxdy
scl_phi_xz = F  ! d^2(phi)/dxdz
scl_phi_yy = T  ! d^2(phi)/dy^2
scl_phi_yz = F  ! d^2(phi)/dydz
```

(continues on next page)

(continued from previous page)

```
scl_phi_zz = T  ! d^2(phi)/dz^2
scl_elev  = T   ! elev       (only on free surface sheet)
scl_elev_t = T  ! d(elev)/dt (only on free surface sheet)
scl_elev_x = T  ! d(elev)/dx (only on free surface sheet)
scl_elev_y = T  ! d(elev)/dy (only on free surface sheet)
scl_elev_xx = T ! d^2(elev)/dx^2 (only on free surface sheet)
scl_elev_xy = F ! d^2(elev)/dxdy (only on free surface sheet)
scl_elev_yy = T ! d^2(elev)/dy^2 (only on free surface sheet)
!------------------------------------------------------------------------
! Vector fields to include
!------------------------------------------------------------------------
vec_vel = T     ! Particle velocity
vec_acc_e = T   ! Local acceleration (Euler)
vec_acc_p = T   ! Particle acceleration (Lagrange)
!------------------------------------------------------------------------
! Free surface sheet (a rectangular domain)
!------------------------------------------------------------------------
if_free_surf = T   ! T=True, F=False
xc_free_surf = 0.0    ! X-center of free surf sheet (in vtx domain coordinates)
yc_free_surf = 0.0    ! Y-center of free surf sheet (in vtx domain coordinates)
xlen_free_surf = 29.2 ! Length of domain in x-dir
ylen_free_surf = 29.2  ! Length of domain in y-dir
npanx_free_surf = 20 ! Number of panels in x-dir
npany_free_surf = 2 ! Number of panels in y-dir
!------------------------------------------------------------------------
! Sea floor (a rectangular domain, only used for finite depth calculations)
!------------------------------------------------------------------------
if_floor = T  ! T=True, F=False
xc_floor = 0.0    ! X-center of sea floor sheet (in vtx domain coordinates)
yc_floor = 0.0    ! Y-center of sea floor sheet (in vtx domain coordinates)
xlen_floor = 29.2 ! Length of domain in x-dir
ylen_floor = 29.2  ! Length of domain in y-dir
npanx_floor = 10 ! Number of panels in x-dir
npany_floor = 2 ! Number of panels in y-dir
!------------------------------------------------------------------------
! 3D-volume domain (a rectangular domain extruded to free surface)
!------------------------------------------------------------------------
if_vol3d = T   ! T=True, F=False
xc_vol3d = -24.0    ! X-center of 3d-volume (in vtx domain coordinates)
yc_vol3d = 0.0    ! Y-center of 3d-volume (in vtx domain coordinates)
zmin_vol3d = -8.0 ! 3d-volume only include water above this z-level.
xlen_vol3d = 8.0 ! Length of 3d-volume in x-dir
ylen_vol3d = 29.2 ! Length of 3d-volume in y-dir
npanx_vol3d = 3 ! Number of 3d-volume cells in x-dir
npany_vol3d = 4 ! Number of 3d-volume cells in y-dir
npanz_vol3d = 5 ! Number of 3d-volume cells in z-dir
z_exp_vol3d = 1.1  ! 1.0=constant spaceing in z-dir, >1.0 => larger cells at␣
↪increasing depth
!------------------------------------------------------------------------
! Vertical cuts (sheet extending from z=z_vcut to free surface)
!------------------------------------------------------------------------
n_vcut = 2       ! Number of vertical cuts
xc_vcut = 0.0  0.0  ! x-position of center of each sheet.
yc_vcut = 0.0  0.0  ! y-position of center of each sheet.
zmin_vcut = -10.0 -10.0  ! Lowest z points of each sheet.
len_vcut = 29.2  29.2 ! Horizontal length of each sheet
beta_vcut = 90.0  0.0  ! Angle (deg) between sheet normal vector and x-axis. (e.g. 0.
↪0 => yz plane, 90.0 => xz plane)
```

(continues on next page)

(continued from previous page)

```
npanxy_vcut = 10 10 ! Number of cells in horizontal direction for each sheet
npanz_vcut = 20 20  ! Number of cells in vertical direction for each sheet
z_exp_vcut = 1.1  1.1  ! 1.0=constant spaceing in z-dir, >1.0 => larger cells at␣
↪increasing depth
!---------------------------------------------------------------------
! Horizontal cuts (Rectangular horizontal sheets)
!---------------------------------------------------------------------
n_hcut = 2       ! Number of horizontal cuts
xc_hcut = 0.0  0.0  ! x-position of center of each sheet.
yc_hcut = 0.0  0.0  ! y-position of center of each sheet.
z_hcut = -5.0 -10.0  ! Z-position of each sheet
xlen_hcut = 29.2  29.2 ! Horizontal length of each sheet in x-dir
ylen_hcut = 29.2  29.2 ! Horizontal length of each sheet in y-dir
npanx_hcut = 10 10 ! Number of cells in x-direction for each sheet
npany_hcut = 10 10 ! Number of cells in y-direction for each sheet
!---------------------------------------------------------------------
! Body box (display a box representing a structure)
!---------------------------------------------------------------------
n_box = 2   ! Number of boxes
xc_box = 15.0  -10.0  ! x-position of center of each box.
yc_box = 0.0  0.0  ! y-position of center of each box.
zc_box =  10.0 -10.0  ! z-position of center of each box.
xlen_box = 15.0  1.0  ! Length of each box in x-dir
ylen_box = 5.0  1.0  ! Length of each box in y-dir
zlen_box = 5.0  50.0  ! Length of each box in z-dir
!---------------------------------------------------------------------
! End of file. (Don't remove the namelist tag below)
!---------------------------------------------------------------------
/
```

Coordinate systems

Swd2vtk has an application specific coordinate system $(\bar{x}, \bar{y}, \bar{z})$ related to the SWD coordinate system (x, y, z) as described in the *theory* section of this documentation. The related constructor parameters are specified in the input file `swd2vtk.inp`.

To make it possible to simulate the SWD wave field near a ship moving with a constant speed along the $\bar{x}$-axis, an **animation coordinate system** is introduced. This system is moving with a constant speed `speedx` along the $\bar{x}$-axis and coincide with the application coordinate system at the application time $\bar{t} = 0$. All locations of objects specified in `swd2vtk.inp` and in the resulting VTK-files are with respect to the **animation coordinate system**. All kinematics presented in the VTK files (e.g. $\frac{\partial \phi}{\partial \bar{t}}$ and particle velocities) are with respect to the earth fixed application coordinate system.

The reference speed `speedx` may take any real value. Negative values indicates that the 'ship' is moving in the negative x-direction.

Input parameters

All input values in `swd2vtk.inp` are specified in a Fortran namelist called `swd2vtk` as demonstrated in the header of the *example file*.

Consequently, comments may be inserted anywhere using the tag "!". Input variables may be specified in any order and in any column. Parameters not specified in `swd2vtk.inp` will keep a default value. Default values are given in the tables below. Characters are enclosed in quotes and logical/bools are specified using the letters T and F without quotes.

Fortran namelist can be parsed and generated from Python using the PyPI package f90nml. This package may also convert between YAML and namelist.

The following parameters are specified in `swd2vtk.inp`.

File related parameters

parameter	type	description
file_swd	char*200	Name of the actual SWD file. (may include relative or full path) No default (max 200 characters)
path_vtk	char*200	Name of the sub-directory to store VTK files. (may be full path) No default (max 200 characters)
vtk_form	char*10	Tag to identify the VTK output file format: 'raw', 'binary' or 'ascii'. Default is 'raw'

Time and speed parameters

parameter	type	description
tstart	real	Start animation output at application time = *tstart* [s]. (Default: *tstart=0.0*)
tend	real	End animation output at application time = *tend* [s]. (No default)
dt	real	Applied time step in animation output [s]. (No default)
speedx	real	Speed of animation coordinates relative to earth fixed application sys. [m/s]. Default is *speedx=0.0*
time_fmt	char*15	Fortran format specifier for time string. E.g. *time_fmt = '(f0.3)'* (three digits) Default: *time_fmt = '(f0.3)'*

SWD constructor parameters

parameter	type	description
x0_swd	real	SWD constructor parameter *x0*. (Default: *x0_swd = 0.0*)
y0_swd	real	SWD constructor parameter *y0*. (Default: *y0_swd = 0.0*)
t0_swd	real	SWD constructor parameter *t0*. (Default: *t0_swd = 0.0*)
beta_swd	real	SWD constructor parameter *beta*. (Default: *beta_swd = 0.0*)
impl_swd	integer	SWD constructor parameter *impl*. (Default: *impl_swd = 0*)
norder_swd	integer	SWD constructor parameter *norder*. (Default: *norder_swd = 0*)
ipol_swd	integer	SWD constructor parameter *ipol*. (Default: *ipol_swd = 0*)
nsumx_swd	integer	SWD constructor parameter *nsumx*. (Default: *nsumx_swd = -1*)
nsumy_swd	integer	SWD constructor parameter *nsumy*. (Default: *nsumy_swd = -1*)
dc_bias_swd	logical	SWD constructor parameter *dc_bias*. T or F. (Default: *dc_bias_swd = F*)
rho_swd	real	SWD constructor parameter *rho*. (Default: *rho_swd = 1025.0*) Pressure will have unit meter water column if *rho_swd = 0.102*.

Requested scalar fields

To decide which scalar fields should be evaluated and included in the VTK files we apply T/F flags representing True or False in the namelist. These fields are calculated using the SWD class methods `pressure()`, `phi()`, `phi_t()`, `stream()`, `elev()`, `elev_t()`, `grad_elev()`, `grad_phi_2nd()` and `grad_elev_2nd()`.

parameter	type	description
scl_prs	logical	Total wave pressure from `pressure()`. (Default: *scl_prs = F*)
scl_stream	logical	Stream function from `stream()`. (Default: *scl_stream = F*)
scl_phi	logical	Velocity potential ϕ from `phi()`. (Default: *scl_phi = F*)
scl_phi_t	logical	$\frac{\partial \phi}{\partial \bar{t}}$ from `phi_t()`. (Default: *scl_phi_t = F*)
scl_phi_xx	logical	$\frac{\partial^2 \phi}{\partial \bar{x}^2}$ from `grad_phi_2nd()`. (Default: *scl_phi_xx = F*)
scl_phi_xy	logical	$\frac{\partial^2 \phi}{\partial \bar{x} \partial \bar{y}}$ from `grad_phi_2nd()`. (Default: *scl_phi_xy = F*)
scl_phi_xz	logical	$\frac{\partial^2 \phi}{\partial \bar{x} \partial \bar{z}}$ from `grad_phi_2nd()`. (Default: *scl_phi_xz = F*)
scl_phi_yy	logical	$\frac{\partial^2 \phi}{\partial \bar{y}^2}$ from `grad_phi_2nd()`. (Default: *scl_phi_yy = F*)
scl_phi_yz	logical	$\frac{\partial^2 \phi}{\partial \bar{y} \partial \bar{z}}$ from `grad_phi_2nd()`. (Default: *scl_phi_yz = F*)
scl_phi_zz	logical	$\frac{\partial^2 \phi}{\partial \bar{z}^2}$ from `grad_phi_2nd()`. (Default: *scl_phi_zz = F*)
scl_elev	logical	Wave elevation ζ from `elev()`. (only on free surface) Default: *scl_elev = F*
scl_elev_t	logical	$\frac{\partial \zeta}{\partial \bar{t}}$ from `elev_t()`. (only on free surface) Default: *scl_elev_t = F*
scl_elev_x	logical	$\frac{\partial \zeta}{\partial \bar{x}}$ from `grad_elev()`. (only on free surface) Default: *scl_elev_x = F*
scl_elev_y	logical	$\frac{\partial \zeta}{\partial \bar{y}}$ from `grad_elev()`. (only on free surface) Default: *scl_elev_y = F*
scl_elev_xx	logical	$\frac{\partial^2 \zeta}{\partial \bar{x}^2}$ from `grad_elev_2nd()` (only on free surface). Default: *scl_elev_xx = F*
scl_elev_xy	logical	$\frac{\partial^2 \zeta}{\partial \bar{x} \partial \bar{y}}$ from `grad_elev_2nd()`. (only on free surface). Default: *scl_elev_xy = F*
scl_elev_yy	logical	$\frac{\partial^2 \zeta}{\partial \bar{y}^2}$ from `grad_elev_2nd()`. (only on free surface). Default: *scl_elev_yy = F*

Requested vector fields

To decide which vector fields should be evaluated and included in the VTK files we apply T/F flags representing True or False in the namelist. These fields are calculated using the SWD class methods `grad_phi()`, `acc_euler()` and `acc_particle()`.

parameter	type	description
vec_vel	logical	Particle velocity from `grad_phi()`. (Default: *vec_vel* = *F*)
vec_acc_e	logical	Local acceleration from `acc_euler()`. (Default: *vec_acc_e* = *F*)
vec_acc_p	logical	Particle acceleration from `acc_particle()`. (Default: *vec_acc_p* = *F*)

Free-surface sheet

Using the variable *if_free_surf* the user may request a thin sheet representing the instant position of the free-surface in a rectangular horizontal region.

The center of this region is specified by the parameters (x=`xc_free_surf`, y=`yc_free_surf`) with respect to the animation coordinate system. The length of this region along the application x-dir is *xlen_free_surf* and *ylen_free_surf* in the y-dir.

The number of evenly distributed cells in the x-dir and y-dir are *npanx_free_surf* and *npany_free_surf* respectively. For each vertex in this horizontal grid the instant wave elevation will be calculated. These wave elevations defines the z-location of this surface sheet.

Selected scalar and vector fields are evaluated in the cell nodes of the free-surface sheet.

parameter	type	description
if_free_surf	logical	Flag to indicate if the free-surface sheet should be generated. Default: *if_free_surf* = *T*
xc_free_surf	real	The x-mid-point of sheet (animation coord. system). No default.
yc_free_surf	real	The y-mid-point of sheet (animation coord. system). No default.
xlen_free_surf	real	The length of free-surface sheet in x-dir (animation coord. system). No default.
ylen_free_surf	real	The length of free-surface sheet in y-dir (animation coord. system). No default.
npanx_free_surf	integer	Number of cells (>0) in x-dir (animation coord. system). No default.
npany_free_surf	integer	Number of cells (>0) in y-dir (animation coord. system). No default.

Sea floor sheet

In case of finite depth simulations a sheet displaying the local bathymetry can be generated. This feature is automatically turned off in case of infinite water depth.

Using the variable *if_floor* the user may request a thin sheet representing the sea floor in a rectangular horizontal region.

The center of this region is specified by the parameters (x=`xc_floor`, y=`yc_floor`) with respect to the animation coordinate system. The length of this region along the application x-dir is *xlen_floor* and *ylen_floor* in the y-dir.

The number of evenly distributed cells in the x-dir and y-dir are *npanx_floor* and *npany_floor* respectively. For each vertex in this horizontal grid the local bathymetry is calculated. These values defines the z-location of the sea floor.

Selected scalar and vector fields are evaluated in the nodes of the sea floor sheet.

parameter	type	description
if_floor	logical	Flag to indicate if the sea floor sheet should be generated. Automatic F in case of infinite depth. Default: *if_floor* = *T*
xc_floor	real	The x-mid-point of sheet (animation coord. system). No default.
yc_floor	real	The y-mid-point of sheet (animation coord. system). No default.
xlen_floor	real	The length of sea floor sheet in x-dir (animation coord. system). No default.
ylen_floor	real	The length of sea floor sheet in y-dir (animation coord. system). No default.
npanx_floor	integer	Number of cells (>0) in x-dir (animation coord. system). No default.
npany_floor	integer	Number of cells (>0) in y-dir (animation coord. system). No default.

3D-volume block

Using the variable *if_vol3d* the user may request if a 3D structured domain of hexahedrons should be generated.

The extent of this domain is rectangular in the horizontal plane. In the vertical direction it occupies the space from *z*=*zmin_vol3d* up to the instant free-surface. The horizontal center of the domain is at (x=`xc_vol3d`, y=`yc_vol3d`). The horizontal length of this domain along the application x-dir is *xlen_vol3d* and *ylen_vol3d* in the y-dir.

The number of cells in the x-dir, y-dir and z-dir are *npanx_vol3d*, *npany_vol3d* and *npanz_vol3d* respectively. In the horizontal directions the cells are evenly spaced. In the vertical direction it is possible to have smaller cells closer to the free surface. The density of cells are controlled using the expansion factor *z_exp_vol3d*.

Selected scalar and vector fields are evaluated in the nodes of each cell.

parameter	type	description
if_vol3d	logical	Flag to indicate if a 3D solid volume block should be generated. Default: *if_vol3d = F*
xc_vol3d	real	The x-mid-point of the volume (animation coord. system). No default.
yc_vol3d	real	The y-mid-point of the volume (animation coord. system). No default.
zmin_vol3d	real	The lowest z-value of the block (animation coord. system). No default.
xlen_vol3d	real	The length of the block in x-dir (animation coord. system). No default.
ylen_vol3d	real	The length of the block in y-dir (animation coord. system). No default.
npanx_vol3d	integer	Number of cells (>0) in x-dir. No default.
npany_vol3d	integer	Number of cells (>0) in y-dir. No default.
npanz_vol3d	integer	Number of cells (>0) in z-dir. No default.
z_exp_vol3d	real	Expansion ratio for cell heights in z-dir. (1.0 = constant spacing) Default: *z_exp_vol3d = 1.1* (cell height for cell below increase 10%)

Vertical cuts

The user may specify a number (*n_vcut*) of vertical cut-planes from *z=zmin_vcut* up to the free-surface. The horizontal extent of this sheet is *len_vcut*. The horizontal center of each sheet is specified by the parameters (x=`xc_vcut`, y=`yc_vcut`) with respect to the animation coordinate system. The angle between the cut plane and the yz-plane is *beta_vcut*.

The number(s) of cells in the horizontal and vertical directions are *npanxy_vcut* and *npanz_vcut*, respectively. The spacing is constant in the horizontal direction. In the vertical direction it is possible to have smaller cells closer to the free surface. The density of cells are controlled using the expansion factor *z_exp_vcut*.

Because there may be more than one vertical cut, the **_vcut* parameters above are arrays. Consequently, if you specify *n_vcut=3* you may specify

```
yc_vcut = 0.0  10.0  20.0    ! Specify for all cuts in one line...
```

in the input file. Similar for the other parameters denoted with the tag (A) in the table below. As an alternative, in accordance with the Fortran namelist convention, you may specify:

```
yc_vcut(1) = 0.0    ! One-based indexing
yc_vcut(2) = 10.0
yc_vcut(3) = 20.0
```

Selected scalar and vector fields are evaluated in the cell nodes of each vertical cut.

parameter	type	description
n_vcut	integer	Number of vertical cuts to define. Default: *n_vcut = 0*
xc_vcut	real (A)	The x-mid-point(s) of vertical cuts (animation coord. system). No default.
yc_vcut	real (A)	The y-mid-point(s) of vertical cuts (animation coord. system). No default.
zmin_vcut	real (A)	The lowest z-value(s) of vertical cut(s) (animation coord. system). No default.
len_vcut	real (A)	The horizontal length(s) of vertical cuts. No default.
beta_vcut	real (A)	The angle(s) between cut plane(s) and the yz-plane [deg]. No default. e.g. 0.0 => yz plane, 90.0 => xz plane
npanxy_vcut	integer (A)	Number of cells (>0) in the horizontal direction for each cut. No default.
npanz_vcut	integer (A)	Number of cells (>0) in z-dir. No default.
z_exp_vcut	real (A)	Expansion ratio for cell heights in z-dir. (1.0 = constant spacing) Default: *z_exp_vcut = 1.1* (cell height for cell below increase 10%)

Horizontal cuts

The user may specify a number (*n_hcut*) of horizontal rectangular cut-planes.

The center of each cut plane is specified by the parameters (x=`xc_hcut`, y=`yc_hcut`) at *z=z_hcut* with respect to the animation coordinate system. The length of this region along the application x-dir is *xlen_hcut* and *ylen_hcut* in the y-dir.

The number(s) of evenly distributed cells in the x-dir and y-dir, are *npanx_hcut* and *npany_hcut*, respectively.

Because there may be more than one horizontal cut, the **_hcut* parameters above are arrays. Consequently, if you specify *n_hcut=3* you may specify

```
z_hcut = -5.0  -10.0  -15.0  ! Specify for all cuts in one line...
```

in the input file. Similar for the other parameters denoted with the tag (A) in the table below. As an alternative, in accordance with the Fortran namelist convention, you may specify:

```
z_hcut(1) = -5.0     ! One-based indexing
z_hcut(2) = -10.0
z_hcut(3) = -15.0
```

Selected scalar and vector fields are evaluated in the cell nodes of each horizontal cut.

parameter	type	description
n_hcut	integer	Number of horizontal cuts to define. Default: *n_hcut = 0*
xc_hcut	real (A)	The x-mid-point(s) of horizontal cuts (animation coord. system). No default.
yc_hcut	real (A)	The y-mid-point(s) of horizontal cuts (animation coord. system). No default.
z_hcut	real (A)	The constant z-value(s) of horizontal cut(s) (animation coord. system). No default.
xlen_hcut	real (A)	The length of horizontal cut(s) in x-dir (animation coord. system). No default.
ylen_hcut	real (A)	The length of horizontal cut(s) in y-dir (animation coord. system). No default.
npanx_hcut	integer (A)	Number of cells (>0) in x-dir (animation coord. system). No default.
npany_hcut	integer (A)	Number of cells (>0) in y-dir (animation coord. system). No default.

Virtual boxes

The user may specify a number (*n_box*) of virtual boxes (rectangular cuboids) for visual references in the animations.

These boxes may visualize the location and extent of ships, wave probes etc. to better see relevant wave profiles at different time instances. These boxes do not move relative to the animation coordinate system.

The virtual boxes do not affect the simulated wave kinematics. Wave kinematics is not evaluated on virtual box surfaces.

The center of each virtual box is specified by the parameters (x=`xc_box`, y=`yc_box`, z=`zc_box`) with respect to the animation coordinate system. The dimensions of each box in the x-dir, y-dir and z-dir are *xlen_box*, *ylen_box* and *zlen_box*, respectively.

Because there may be more than one box, the **_box* parameters above are arrays. Consequently, if you set *n_box=3* you may specify

```
xc_box = -50.0  0.0  50.0  ! Specify for all boxes in one line...
```

in the input file. Similar for the other parameters denoted with the tag (A) in the table below. As an alternative, in accordance with the Fortran namelist convention, you may specify:

```
xc_box(1) = -50.0    ! One-based indexing
xc_box(2) =   0.0
xc_box(3) =  50.0
```

parameter	type	description
n_box	integer	Number of virtual boxes to define. Default: *n_box = 0*
xc_box	real (A)	The x-mid-point(s) of boxes (animation coord. system). No default.
yc_box	real (A)	The y-mid-point(s) of boxes (animation coord. system). No default.
zc_box	real (A)	The z-mid-point(s) of boxes (animation coord. system). No default.
xlen_box	real (A)	The length of boxes in x-dir (animation coord. system). No default.
ylen_box	real (A)	The length of boxes in y-dir (animation coord. system). No default.
zlen_box	real (A)	The length of boxes in z-dir (animation coord. system). No default.

Download and Installation

Binary distributions of **swd2vtk**, for Windows and Linux, can be downloaded from the release tab of the GitHub repository **spectral_wave_data**.

Note: To run **swd2vtk** on windows you need to download and install the latest version of the Intel Redistributable Package for Fortran 2020. The Intel redistributable contains required DLLs. If you already have a very recent Intel Fortran compiler installed you may skip this installation.

Acknowledgement

swd2vtk applies the library VTKFortran by Stefano Zaghi (INSEAN) under the MIT-license.

8.2 Airy Waves

A general set of Airy waves can be written directly to a SWD file (*shape 6*) using the `tools.airy` module included in **spectral_wave_data**.

In this example we apply two wave components.

```
from spectral_wave_data.tools import airy

airy.write_swd('my_airy.swd',
               amps=[2.0, 3.0], dirs=[180.0, 70.0], phases=[50.0, 0.0], Twaves=[10.5,␣
↪13.0],
               depth=32.0, grav=9.81, is_deg_dirs=True, is_deg_phases=True)
```

The *documentation* of this function describes relevant optional parameters and further examples.

8.2.1 spectral_wave_data.tools.airy.write_swd

The following function available in the sub-package `spectral_wave_data.tools` generates a SWD-file for a general set of Airy waves. All conventions follow the definitions described by *shape 6*.

Examples

First we establish access to the function:

```
from spectral_wave_data.tools import airy
```

In the first example we assume that wave numbers are explicitly defined and the water depth is infinite. In all examples we assume that both phases and propagation directions are given in the unit [deg]

```
amps = [...]   # Single wave amplitudes for each component
dirs = [...]   # Wave propagation directions for each component
phases = [...] # Phase angles for each component
kwaves = [...] # Wave numbers for each component
airy.write_swd('my_airy.swd', amps, dirs, phases, kwaves=kwaves,
               depth=-1.0, is_deg_dirs=True, is_deg_phases=True)
```

In case wave numbers are implicitly defined by wave periods:

```
Twaves = [...] # Wave periods for each component
airy.write_swd('my_airy.swd', amps, dirs, phases, Twaves=Twaves,
               depth=-1.0, is_deg_dirs=True, is_deg_phases=True)
```

In case wave numbers are implicitly defined by wave frequencies:

```
omegas = [...] # Wave frequencies for each component
airy.write_swd('my_airy.swd', amps, dirs, phases, omegas=omegas,
               depth=-1.0, is_deg_dirs=True, is_deg_phases=True)
```

In case wave numbers are implicitly defined by wave lengths:

```
Lwaves = [...] # Wave lengths for each component
airy.write_swd('my_airy.swd', amps, dirs, phases, Lwaves=Lwaves,
               depth=-1.0, is_deg_dirs=True, is_deg_phases=True)
```

In case *Twaves* or *omegas* is applied the linear wave dispersion relation is applied to establish the corresponding wave numbers to be stored in the SWD file. This function is available from the same Python module:

8.3 Fenton-Stream Waves

The raschii Python package for making non-linear regular waves supports output of SWD files. An example of producing a Fenton-Stream wave is given by this code snippet.

```
import raschii

WaveModel, AirModel = raschii.get_wave_model('Fenton')
wave = WaveModel(height=5.0, depth=15.0, length=200.0, N=30)
wave.write_swd('my_fenton.swd', dt=0.05, nperiods=50)
```

This method applies the `shp=2` shape class. The module `swd_tools.py` in the raschii distribution demonstrates how to create SWD files directly from Python.

Raschii also has a graphical online demo.

Note: This feature in raschii will soon be released on PyPI. However, it is included in the "write_swd" branch of that repository.

8.4 Stokes Waves

The raschii Python package may also output Stokes waves in the SWD format.

```
import raschii

WaveModel, AirModel = raschii.get_wave_model('Stokes')
wave = WaveModel(height=5.0, depth=15.0, length=200.0, N=5)
wave.write_swd('my_stokes.swd', dt=0.05, nperiods=50)
```

This method applies the `shp=2` shape class.

Note: This feature in raschii will soon be released on PyPI. However, it is included in the "write_swd" branch of that repository.

8.5 WAMOD

DNVGL is releasing the program WAMOD for simulation of irregular seas based on HOS(M), the Higher-Order-Spectral-Method. The typical input is a wave spectrum and the output is a SWD file providing linear or nonlinear wave fields. Long and short crested seas can be simulated in finite or infinite water depth.

The new release of WASIM can apply the SWD API for incident wave kinematics and Froude-Krylov forces. There are no restrictions regarding which wave generator produced the SWD file. WASIM is a Rankine panel program for simulating wave induced responses of marine structures.

WAMOD and WASIM have commercial licenses and support.

8.6 Other programs

Other software utilizing SWD will be added to this page when they are released. Please contact the maintainers when new software appear.

CHAPTER

NINE

EXAMPLE WAVES

All waves presented on this page are available as SWD-files from the GitHub repository spectral_wave_data_waves. You can download them and evaluate all kinematics as defined in the *theory* section at any relevant location in space and time.

Examples will soon be uploaded

CHAPTER

TEN

VERIFICATION AND TESTING

It is important to know that for **spectral_wave_data** the concept of verification is much more relevant than validation.

The quality of a specific wave field should only depend on the *wave generator*. Hence, **spectral_wave_data** should only reproduce the kinematics as it is defined by the contents of the SWD file and the *mathematical spectral definition*. However, this reproduction should be of high accuracy.

10.1 The verification procedure

Each of the temporal interpolation schemes described in the *theory* section is consistent to a certain order. Consequently, for all spectral amplitudes defined by polynomials of the actual order, these schemes should produce exact temporal evaluations.

For each temporal interpolation scheme, random polynomial spectral amplitudes of the corresponding order are constructed and stored in SWD files. Consequently, the associated waves have no physical relevance. Wave kinematics are then evaluated using two different procedures:

1. The Python **spectral_wave_data** package, which apply the Fortran and C implementations, evaluates all supported kinematics from the constructed SWD files.
2. A python test script combines the analytical polynomial spectral amplitudes, with the actual mathematical spectral definition. By using symbolic mathematics (SymPy) all kinematics are evaluated analytically using symbolic differentiation of the formulas for wave potential and surface elevation. Consequently, the explicit formulas in the *theory* sections for e.g. particle velocities are not applied in these evaluations. The symbolic calculations are all evaluated directly in the application specific time and space coordinate system.

Using pytest all kinematics from both the **spectral_wave_data** package and the symbolic computations described above are calculated and compared. Any deviations are reported.

This verification procedures is applied for all spectral shape definitions and supported implementation schemes, all temporal interpolation schemes, and a representative set of different constructor parameters (x0, y0, t0 and beta).

For shape 6 (Airy waves) the auxilary tool included in **spectral_wave_data** is applied when writing the relevant test SWD files. Hence, `spectral_wave_data.tools.airy.write_swd()` is verified too.

The master/origin branch of this repository does not update unless all these verification tests pass successfully.

10.2 Testing

All tests are executed using pytest and scripts located in the `tests/python` directory of the **spectral_wave_data** repository.

In this directory there are some relevant scripts:

```
install_requirements_tests.bat
run_all_tests.bat
run_failed_test.bat
```

There are similar scripts for Linux. The script `install_requirements_tests.bat` will if needed, install all required Python packages (from PyPI) for running the tests and creating the test report.

Running `run_all_tests.bat` will run all tests and display a html report when finished. Normal output from the package and scripts are suppressed by pytest.

The script `run_failed_test.bat` reruns only the first failed test and provide detailed output including local variables from the python scripts.

Running all tests may take a long time but is required in order to submit a pull request for updating this GitHub repository.

New features, like new spectral formulations, will not be included in the master/origin branch before proper pytest-scripts have been established. If needed, we may assist...

Note: The test suite described in this section apply test features not supported in Python 2.7. Hence it is only possible to run these tests in Python 3.5 and newer.

The core package **spectral_wave_data** seems to run well on Python-2.7 too, but support for 2.x may be dropped in the future if dependency packages drop support for 2.x. This is a general trend in the Python community because January 1st, 2020, Python 2.7 officially reached the end of life and will no longer receive security updates, bug fixes, or other improvements going forward.

As a consequence we strongly recommend to apply Python-3 for future proof utilization of **spectral_wave_data**.

CHAPTER

ELEVEN

CONTACTS

If you have issues, or for some other reasons want to contact us please do so.

Bug reports should in general be submitted directly to the GitHub repository.

As for most open source projects, resources are limited. Consequently, we cannot promise to implement requested features. Especially within short deadlines.

However, if you have some pressing needs we will consider to do regular consulting work for a fair price. The results will be available in the official API with corresponding acknowledgement to your contribution.

We appreciate your feedback.

- Jens B. Helmers (jens dot bloch dot helmers at dnvgl dot com)
- Odin Gramstad (odin dot gramstad at dnvgl dot com)

CHAPTER

TWELVE

ACKNOWLEDGEMENTS

- Professor Alexey Slunyaev suggested a similar approach for coupling WASIM with his implementation of the Higher-Order-Spectral-Method back in the EU research project Extreme Seas. The proof of concept was demonstrated by simulating (in Norway) the ship response induced by a Peregrine breather wave simulated by HOSM in Russia. No time spent on merging programs. Weird but striking...
- DNVGL supported the initial development and public release of this API. DNVGL will continue this effort in collaboration with the industry and academia in an open-source environment. The goal is to improve development and communication in this important field of technology.

CHAPTER

THIRTEEN

MIT LICENSE

CHAPTER

FOURTEEN

CITATION

You may cite the **spectral_wave_data** repository using the following BibTeX entry.

```
@misc{SpectralWaveData2020,
  author = {Helmers, J.B. and Gramstad O.},
  title = {Spectral Wave Data},
  year = {2020},
  publisher = {GitHub},
  journal = {GitHub repository},
  howpublished = {\url{https://github.com/SpectralWaveData}},
  version = {1.0.0}
}
```

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex

BIBLIOGRAPHY

[DBTouzeF16] Guillaume Ducrozet, Félicien Bonnefoy, David [Le Touzé], and Pierre Ferrant. Hos-ocean: open-source solver for nonlinear waves in open ocean based on high-order spectral method. *Computer Physics Communications*, 203:245 – 254, 2016. URL: http://www.sciencedirect.com/science/article/pii/S0010465516300327, doi:https://doi.org/10.1016/j.cpc.2016.02.017.

[GDF16] Maïté Gouin, Guillaume Ducrozet, and P. Ferrant. Development and validation of a non-linear spectral model for water waves over variable depth. *European Journal of Mechanics - B/Fluids*, 57:115–128, 05 2016. doi:10.1016/j.euromechflu.2015.12.004.