INTERACTION OF A TURBULENT WIND WITH OCEAN SURFACE WAVES–NUMERICAL MODELING

A Dissertation

by

YING-PO LIAO

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	James M. Kaihatu
Co-Chair of Committee,	Reza Sadr
Committee Members,	Bjarne Stroustrup
	Hamn-Ching Chen
	Scott A. Socolofsky
Head of Department,	Sharath Girimaji

December 2016

Major Subject: Ocean Engineering

ABSTRACT

Wave-modeling can be categorized in terms of different scales and theoretical frameworks. This dissertation focuses on the numerical modeling of wind-wave generation and its effects on wave growth and propagations. As categorized by scales and methodologies, wind-wave modeling in this dissertation covers two main topics: 1) *Large-scale modeling: wind-wave development in real seas.* As a phase-average model, SWAN is employed to study the wind-wave environment in the Persian Gulf and Qatar. The wind-wave generation is parameterized as source terms in a spectral model. The special wind condition, called shamal, is particularly investigated. An experimental tower is installed around Doha Port, and by using video imagery, the in situ wave features are extracted and compared. 2) *Small-scale modeling: detailed wave development using CFD (Computational Fluid Dynamics).* A curvilinear surface-fitted moving grid model for three-dimensional Navier-Stokes equations is developed and used to simulate linear and non-linear waves with fully nonlinear surface conditions. Also, by simplifying it to a fixed rectilinear grid based on Cartesian formulations, a DNS (Direct Numerical Simulation) model is developed with an air-water fully-coupled domain and improved coupled interface conditions. By using this DNS model, the detail of wind-wave generation is investigated from still water and the applied top shear wind.

For the second topic, the CFD problems are solved by an in-house numerical tool, SPX. SPX is a general PDE (Partial Differential Equations) framework, devel-oped by using C++1y (shortened form of C++11/14/17), currently aiming at the structural domain. It is designed by modern software methodologies, such as generic programming, meta-programming and object-oriented programming. In addition,

concept-based generic programming, an ongoing advanced software technology, is first introduced into the PDE numerical tool design. By using these modern design methodologies, all significant components used for solving PDE, particularly for fluid and wave problems, are all implemented in SPX. These components include high-performance numerical array, implicit solvers, grids, differential basis and operators, time integrators, and system infrastructures such as serializations and timer. On structured domain, a general PDE can be expressed by the arbitrary combination of any general differential operator and any arithmetic operator, which is the most challenging part of SPX design. This research proposes a general stencil operator design that integrates with the concept-based expression template. It is successfully demonstrated that the proposed design can automatically deduce the resulting stencils to represent the resulting field operator by giving an arbitrary PDE expression at any given grid point. With the deduced stencils, the user-defined PDE expression is therefore, numerically-solvable by using any solver. In consequence, SPX can be easily applied to any user-defined PDE problem on structural grids with arbitrary user-specified numerical components. Its design shows high flexibility and re-usability without sacrificing efficiency. The development of SPX, therefore, justifies the success of C++-Concept applications on the large-scale numerical framework design.

DEDICATION

To my parents, my sister and my husband, who always love me, encourage me, inspire me, and have firmly supported me every step of the way.

ACKNOWLEDGMENTS

First of all, I wish to thank my supervisor, Dr. James M. Kaihatu, for introducing me to this challenging and interesting topic. He has firmly and generously supported me throughout all of my research works. In addition, I would like to thank my Co-Chair, Dr. Reza Sadr, for his collaboration on and contribution to on-site experiments.

Second, I wish to thank my committee members for their tremendous help and advice. Thanks to Dr. Bjarne Stroustrup, for introducing me to advanced C++ technology and for his dedicated advice on my works. Thanks to Dr. Hamn-Ching Chen for his selfless advice on all of my CFD-related questions. Thanks to Dr. Scott A. Socolofsky for his comments on my dissertation and defense.

Special thanks to Dr. Mei-Ying Lin and Dr. Wu-Ting Tsai for their unselfish support, including engaging in private discussion with me when needed and making available their research notes and code.

Finally, I sincerely appreciate my family, all of my friends, and lab mates. You always keep me company and encourage me when times get tough.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

I am indebted to Ms. Pamela G. Posey, Oceanography Division, Naval Research Laboratory at Stennis Space Center, MS, USA, for providing the COAMPS hindcast wind fields.

The Part 1 work was supported by Dr. Arindam Singha and my Co-Chair, Dr. Reza Sadr of the Department of Mechanical Engineering at Texas A& M University at Qatar. They performed on-site experiments and collected wind observations and wave videos.

The Part 2 work, regarding the SPX framework development, was supported by Dr. Andrew Sutton, the assistant professor of the Department of Computer Science at University of Akron. He directly contributed `spx::array`, endeavored to develop a stable version of C++-Concept compiler, and provided a very useful Origin package.

Funding Sources

This work was supported by the Qatar National Research Fund (a member of the Qatar Foundation), under Grants NPRP 8-634-2-269 and NPRP 5-543-2-220.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

The process by which wind generates waves requires sophisticated physical mechanisms to describe and spans many spatial and temporal scales. The generation and evolution of wind-waves have been studied for almost 100 years, yet many questions remains. How do waves grow by air flow? Does air turbulence play a role? How does the surface wave interact with mean flow and ocean upper boundary layer? What is the wave age-dependent process for wind-wave growth from linear to exponential growth rate, from short waves to long waves, and from small scale to large scale? What is the fundamental connection between small scale wind-wave growth to the time-dependent spectrum in an actual sea?

This dissertation will investigate specific small and large scale problems in wind-wave generation and propagation. Given the scale-dependent nature of wind-wave generation and the different types of wave modeling, we will involve two approaches: 1) a large-scale study using spectral wave modeling for the wind-wave propagation in a real sea, and 2) a small-scale study using computational fluid dynamics (CFD) that solves three-dimensional Navier-Stokes. For the first approach, the phase-averaged spectral wave model SWAN ("Simulating WAves Nearshore"; [10, 62]) is used to study the wind-wave conditions in Persian Gulf and Qatar, while in the second approach the CFD work will be performed with our in-house developed SPX. SPX is a general numerical framework for solving partial differential equation (PDE) on structural grids, which can generally solve any user-defined problems. The PDE tool is developed as a generalizable, flexible utility for the numerical solution of partial differential equations, and as such will comprise a major portion of this dissertation.

The dissertation consists of three parts. We will briefly introduce each part as

follows:

- *Part 1—A Large Scale Study: Wind-Waves in Persian Gulf and Qatar.* In this part we will use numerical models to investigate the wind-wave propagation in Persian Gulf and Qatar, with particular focus on the unique wind-wave conditions in this area known as "shamal". The SWAN model is central to this work. This phase-averaged model is expressed in terms of wave action density, within which energy sources, sink and redistribution functions are described. The model is used to uncover the long term climatology for the basin. Moreover, the effects of bathymetry, swell boundary conditions, hindcasting domain size, and the selection of wind source, will be investigated. One wind source is from local measurements taken from an experimental pier installed in the nearshore area around Qatar. In addition to winds, cameras were also used to provide imagery of the waves. Therefore, in this section we also employ video imagery analysis to extract wave properties from these images.

- *Part 2—SPX: A Generic PDE Framework for Structrual Grids using C++11/14/17 and Concept-Based Design.* The goal of SPX is to allow users to easily solve any PDE on the structured domain. It allows user to 1) arbitrarily pick any one of two types of domain: rectilinear or curvilinear, 2) generally define any differential operator and build equation for every individual node, 3) select differential schemes such as finite different or spectral method for any order differentiations (SPX also supports mixed schemes), 4) choose any time integration scheme (an ODE solver) for transient problems, and 5) choose any implicit linear or nonlinear solvers for stationary problems. Thanks to the major advance of C++1y (shortened form of C++11/14/17), the design of SPX is aligned with the state-of-the-art concept-based generic programming, as well

as emphasizes the modern features provided by the new C++ standards. They allow SPX to use a powerful infrastructural features such as high-performance numerical arrays and the automatic deduction of any general stencil differential operator at compile-time, using concept-based expression templates.

- *Part 3—A Small Scale Study: Detailed Wave Development using CFD.* In this part, a small scale study for the generation of wind-waves is conducted using CFD. Unlike other approaches, which use simplifying assumptions and make use of coarse parameterizations, DNS provides an *ab initio* solution in which waves can be grown from a flat surface by the upper air turbulent flow that is driven by applying an upper shear wind. By using SPX, a CFD model is developed to solve the three-dimensional Navier-Stokes equations with surface-fitted curvilinear moving grid, and nonlinear surface stress conditions, by using pseudospectral method. In addition, a simplified version is also developed for the use of DNS modeling in which a fixed rectilinear grid is employed with Cartesian formulations with the air-water fully coupled interfacial conditions. Having the results of DNS modeling, the growth of surface wave elevation and the evolution of many interface properties can be examined accordingly.

2. PART 1—A LARGE SCALE STUDY: WIND-WAVES IN PERSIAN GULF AND QATAR*

2.1 SWAN model

The SWAN model ("Simulating WAves Nearshore"; [10, 62]) is a third generation wind wave model for coastal regions. The wave evolution problem is solved in terms of action density spectrum (a statistical quantity related to the energy spectrum) in different dimensions. The SWAN model also includes the effects of currents and bathymetry of the wave evolution. Because the presence of an ambient current causes frequency-shifting, energy density is not conserved, while action density is conserved and is thus the dependent variation in SWAN. Action density spectrum $N(\sigma, \theta)$ is related to energy density spectrum $E(\sigma, \theta)$ by $N(\sigma, \theta) = E(\sigma, \theta)/\sigma$. Instead of absolute radian frequency, action density is varied with respect to relative radian frequency σ, as well as the wave direction θ. In spherical coordinates, the spectral action balance equation used in SWAN can be formulated with respect to geospatial reference of coordinates

$$\frac{\partial}{\partial t}N + \frac{\partial}{\partial \lambda}C_\lambda N + (\cos\varphi)^{-1}\frac{\partial}{\partial \varphi}C_\varphi N + \frac{\partial}{\partial \sigma}C_\sigma N + \frac{\partial}{\partial \theta}C_\theta N = \frac{S}{\sigma} \tag{2.1}$$

where action density N is propagated with respect to several independent variables—time t, longitude λ, latitude φ, relative radian frequency σ, and the wave direction θ. The first term on the left hand side is the local rate of change of action density in

*Part of the content reported in this chapter is reprinted with permission from: 1) "The effect of wind variability and domain size in the Persian Gulf on predicting nearshore wave energy near Doha, Qatar", February 2016. Applied Ocean Research, 55, 18-36, Copyright by ©Elsevier Ltd. 2) "Numerical Investigation of Wind Waves in the Persian Gulf: Bathymetry Effects", January 2016. Journal of Atmospheric and Oceanic Technology, 33(1), 17-31, Copyright by ©American Meteorological Society.

time, the second and the third terms describe the spatial variations of action density with propagation velocities C_λ and C_φ, respectively along with λ and φ dimensions. The fourth term represents shifting of relative radian frequency due to currents, with propagation velocity C_σ along with σ dimension. The fifth term stands for the depth-induced and current-induced refraction in which action density transports with propagation velocity C_θ along the θ dimension.

On the right hand side, $S = S(\sigma, \theta)$ represents the energy sources and sinks. These include the effects of: wave generation (e.g., linear and exponential wind growth); wave dissipation (e.g., whitecapping, bottom friction, and depth-induced breaking); and nonlinear wave-wave interactions (e.g., triad and quadruplet wave-wave interactions). For these simulations, the SWAN model is run in "third-generation" mode, indicating that there are no *apriori* restrictions on the spectral evolution [29], apart from those which govern the existence of the spectrum.

The wind source term is represented by two parts:

$$S = S(\sigma, \theta) = \alpha + \beta E(\sigma, \theta) \tag{2.2}$$

where the first term stands for the linear initial growth stage [57] and the second term stands for the exponential growth stage in terms of Miles' feedback mechanism [48]. The value α evaluated from SWAN is formulated by [12]:

$$\alpha = \begin{cases} \dfrac{0.0015}{g^2 2\pi} \left[u_* \cos(\theta - \theta_{wind})\right]^4 G & \forall\, |\theta - \theta_{wind}| \leq 90^\circ \\ 0 & \forall\, |\theta - \theta_{wind}| > 90^\circ \end{cases} \tag{2.3}$$

where the cut-off function G is given by

$$G = \exp\left[-\left(\frac{\sigma}{\sigma^*_{PM}}\right)^{-4}\right] \quad \text{with} \quad \sigma^*_{PM} = 2\pi\frac{0.13g}{28u_*} \tag{2.4}$$

where θ_{wind} is the angle of winds, σ^*_{PM} is the peak frequency of the Pierson and Moskowitz spectrum, and u_* is the friction velocity that can be converted from 10-m height wind velocity U_{10} by using

$$u_*^2 = C_D U_{10}^2 \quad \text{with} \quad C_D = \begin{cases} 1.2875 \times 10^{-3} & \forall U_{10} < 7.5\text{m/s} \\ (0.8 + 0.065U_{10}) \times 10^{-3} & \forall U_{10} < 7.5\text{m/s} \end{cases} \tag{2.5}$$

On the other hand, for the exponential growth stage, the value β is given by [72, 43]:

$$\beta = \max\left\{0, 0.25\frac{\rho_a}{\rho_w}\left[28\frac{u_*}{c}\cos\left(\theta - \theta_{wind}\right) - 1\right]\right\} \tag{2.6}$$

where c is the phase speed, ρ_a is the air density, and ρ_w is the water density.

SWAN Cycle III version 40.92 was used for this study. Any adjustable parameters used to constrain physical processes were set to default values as defined in the model documentation [22].

2.2 Literature reviews for wind-wave study for Persian Gulf

The Persian Gulf is a large body of water located bordered by Iran, Kuwait, Iraq, Saudi Arabia, Bahrain, Qatar, and the United Arab Emirates. It is connected to the Indian Ocean by the Strait of Hormuz. The Gulf holds a significant portion of the world's oil reserves, and the Strait of Hormuz is considered a critical chokepoint for energy security; it is estimated that as many as 17 million barrels of crude oil per day passed through the straits in 2011 [85].

A major component of the meteorological environment in this area is the shamal,

a strong northwesterly wind caused by cold fronts passing over the mountains of eastern Turkey and northern Iraq. It is a seasonal wind event occurring primarily during summer and winter. The winter shamal season generally lasts from November to February with an average speed of $5(m/s)$, while the summer shamal season lasts from June to September with a slightly weaker average speed of $3(m/s)$ [19, 83]. Gusts associated with these magnitudes can result in energetic wind seas, which would hamper marine traffic. Figure 2.1 shows the area of interest.

Attention has been primarily focused on the hydrodynamics of the area [20, 83]. Before 2010, there have been relatively few studies of wind wave processes in the Gulf, particularly with respect to the response to the shamal [53, 54, 59], which all are based on the WAM model ("The WAve Model"; [30]). However, comparing to WAM, SWAN incorporates source terms which have more relevance to shallow water processes (e.g. depth limited breaking); [63, 31] show that the SWAN model outperforms the WAM model for when compared to coastal cases. SWAN is therefore much appropriate to be used for modeling regional oceans, such as Persian Gulf. [49] employed SWAN to perform wave hindcast for Persian Gulf in which winds from ECMWF and a spatially-constant in-situ wind station are used as wind sources. Wave assimilation using SWAN and in-situ measurements for this area has been studied since 2012 [50, 51].

The effects of bathymetric variation in wave modeling have been widely studied. Many regional studies have, in recent years, used SWAN. [25] discussed the complex bathymetric effects for the Scripps Canyon. [40] performed a sensitivity analysis focusing on the regional wave response upon a complex bathymetry. [64] applied a multi-level approach in three different levels of detail of computational grid and bathymetry resolution. [58] studied the alongshore and cross-shore wave and hydrodynamic sensitivity, while [47] derived optimization schemes using genetic

algorithms to reduce the bathymetric sampling required in nearshore wave and hydrodynamic modeling. Both [86] and [55] involve in data assimilation for nearshore wave modeling in which bathymetry effects are discussed.

There are fewer studies concerned with basin-scale characteristics of waves in the Persian Gulf, mainly because proper characterization requires long-term hindcasting using reliable and high-resolution wind data. [52] published the first wave atlas for Persian Gulf. A 10-year (1992-2001) hindcasting was conducted using the MIKE 21 SW and ECMWF wind. Similarly, by a 25-year hindcasting (1984-2008) using SWAN and ECMWF wind, [41] studied the wave features both spatially and temporally, not only plotting the wave power spatial distribution, but also investigating the seasonal variations of wave power, as well as the decadal trends using time series of annual average values. However, wave atlas shows only the most expected values and does not usually offer other statistics. [56] proposed a statistical study but it is only for the north Persian Gulf.

In section 2.3 we will introduce the study area and the data used for modeling, including the observation tower built on the maintenance pier near Doha Port for the on-site data observations. Two long-term basin-scale investigations are respectively discussed in section 2.4 for climatological and statistical properties and, in section 2.5 for the effects of bathymetry. In section 2.6 we propose a multi-level modeling approach and develop a series of model configurations to investigate the effects of domain size, swell boundary conditions, and the effects of different wind forces on the climatological results. Finally, in section 2.7 the methodology of video imagery is developed and employed to extract the wave properties from videos of the free surface taken at the maintenance pier, which can be used to compare with numerical results.

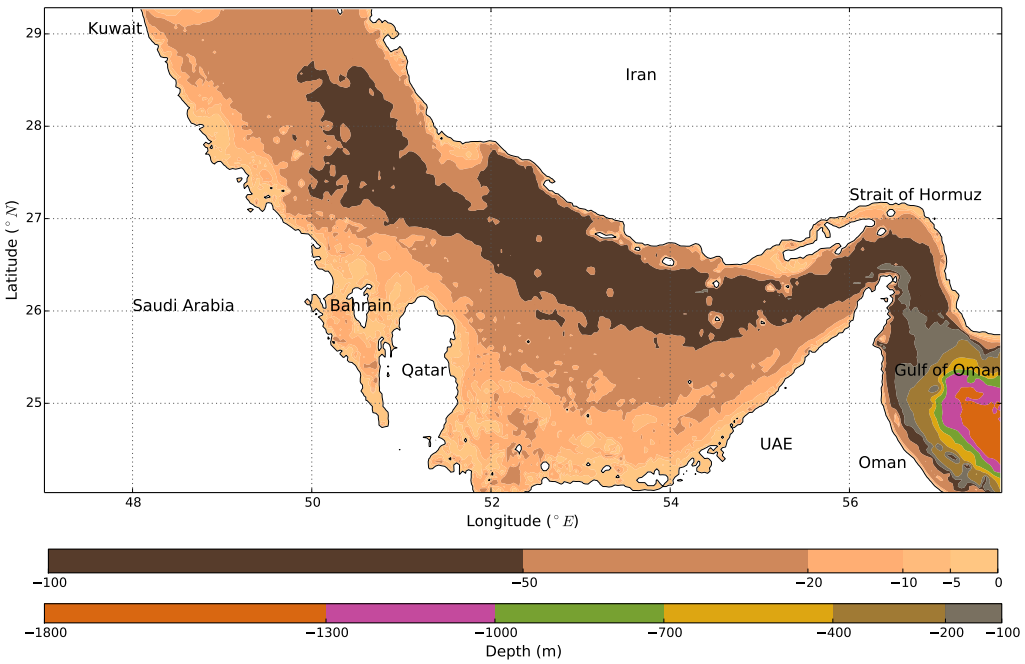


Figure 2.1: Bathymetry and computational domain of the study area

2.3 Data preparation

2.3.1 Study area

The Perisan Gulf is a long, flat and shallow semi-enclosed basin, located between $24-30°$N and $48-57°$E. The basin width varies from 56 to $338km$, while its central axis is about $990km$ long. Its total area is about $226,000km^2$ and the average water depth is around $35m$ [21]. The only large-scale passages are Straits of Hormuz, located in the south of the basin with a narrow opening $56km$ and which connects the Persian Gulf to the Gulf of Oman and the Arabian Sea.

2.3.2 The maintenance pier

As shown in Figure 2.2, a maintenance pier at $(25.247448°N, 51.630375°E)$, built to service approach lights at Doha International Airport, was the site of an instal-

lation of instruments intended to measure winds and ocean waves [71]. The water depth at this site is 2.1 m. The instrumentation installed at the site included three sonic anemometers and two video cameras mounted on an aluminum tower, as well as a weather station mounted a short distance away from the tower. Details of the anemometers can be found in [71]. Of interest in this study is the weather station (a Davisnet Vantage Pro 2), which was mounted at an elevation of 4.5 m above mean sea level. Wind information from the weather station consisted of 15 minute averages of velocity (in m/s) and direction (measured from north in 22.5° intervals).

Figure 2.2: Maps of the Persian Gulf, Qatar, and the instrument location.

2.3.3 Bathymetry data

The ETOPO1 database [5] is used for the bathymetric data in all cases. The bathymetry in this database has a resolution of one arc-minute in both latitude and longitude, and a vertical resolution of around $10m$. There are no other freely

available sources of bathymetric information at higher resolution. For our purposes, this is not an issue since we will be investigating relative performance of the model given varying amounts of bathymetric information. Moreover, [55] shows that for flat basin and straight coastlines without bathymetric details does not changes the sensitivity for the nearshore wave modeling using SWAN.

2.3.4 Hindcast wind fields

Modeled hindcast wind fields were used to provide spatially-dense wind fields for the area. We use two different hindcast data sets for this study: Coupled Atmosphere Mesoscale Prediction System (COAMPS) and Climate Forecast System Reanalysis (CFSR). In addition to hindcast wind fields, measured winds from a weather station (denoted QTRSTA) mounted on the maintenance pier were also used as the spatially-constant wind source.

1. *COAMPS[32]*: an ocean-atmosphere coupled mesoscale weather model developed by the Naval Research Laboratory. The model combines the physics of mesoscale atmospheric dynamics with a data assimilation scheme to increase the accuracy of the solution via updated boundary conditions. Data from the model consists of grid-based wind speed vectors at a spatial resolution of $0.2^\circ \times 0.2^\circ$. Furthermore, the temporal resolution was 12 hours.

2. *CFSR[2]*: developed by the National Center for Environmental Prediction (NCEP). The data set will be referred to as NCEP hereafter. While COAMPS is an analysis data set, NCEP is a reanalysis, with more data used to increase the accuracy of the windfields. NCEP provides a grid-based 10-m wind vector data at a spatial resolution of $0.3125^\circ \times 0.3123^\circ$ and a temporal resolution of 6 hours.

3. *QTRSTA*: Wind velocity components u and v deduced from the weather station mesurements were converted from their values at 4.5 m elevation to 10 m height according to a one-seventh power law [60]:

$$U_{10} = U\left(\frac{10}{4.5}\right)^{1/7} \tag{2.7}$$

where U_{10} is the velocity component at 10-m height and U is the velocity component measured by the weather station at 4.5-m height. The missing data are filled by inverse-distance interpolation from the neighbor weather stations.

2.4 Long-term wind-wave climatology

Given a deficiency of wave measurements, hindcasting is a typical approach to provide wave climate information. Wind-wave hindcasting is usually driven by an archived wind field, a reanalysis or operational dataset assimilated with meteorological observations, typically provided by global meteorological or oceanic institutes, i.e., ECMWF (European Center for Medium Range Weather Forecasts). Statistical techniques can be applied to the results to provide climatological characterization.

In this study, a 5-year (2004-2008) hindcasting exercise is performed using COAMPS. The parameters for model configuration are listed as Table 2.1. The time step of computation is selected as 20 min throughout 5-year simulation from 2004/01/01 to 2008/12/31. Regular discretization is used for both spatial and spectral domain. The extent of computational domain coincides with the area of bathymetry, bounded between $24°02' - 29°17'$N and $47°01' - 57°42'$E, and discretized into 641×315 cells with $1' \times 1'$ spatial resolution. For the spectral domain, the spectral direction is equally divided into 36 subdivisions around a circle in which $\triangle\theta = 360°/36 = 10°$; the frequency dimension is bounded between 0.06 Hz and 1 Hz and for the grid resolution 36 discrete frequencies are logarithmically distributed along σ space. The

results are outputted every 3 hours.

Table 2.1: Model configuration for long-term hindcasting.

Parameter	Value
Origin (° E, ° N)	$47°1', 24°2'$
x-length (longtitude)	$10°41'$
y-length (latitude)	$5°15'$
Number of x-cells	641
Number of y-cells	315
Δx	$1'$
Δy	$1'$
Δt (min)	20
Frequency range (Hz)	0.06–1
Frequency subdivisions	36 (in log space)
θ subdivisions	36 (in linear space)
$\Delta\theta$	$10°$

As a result of these computations, each grid point in the computational domain has its own 5-year time series for every physical parameter, i.e., wind speed $U(m/s)$, peak period $T_p(s)$, and significant wave height $H_s(m)$. For long-term climates, the joint probability of these parameters can be all described by Weibull distribution, among others. The spatial variation of these parameters is of interest. Therefore, the first step is to compute the regression of Weibull distribution for each grid point within the basin region for these parameters. A two-parameter Weibull probability density distribution P for a random variable ϕ, i.e., U, T_p, or H_s, can be formulated as

$$P(\phi) = \left(\frac{C}{B}\right)\left(\frac{\phi}{B}\right)^{C-1} \exp\left[-\left(\frac{\phi}{B}\right)^{C}\right] \tag{2.8}$$

where B is scaling parameter and C is shape parameter. Note that B is dimensional. i.e., m/s for U, s for T_p, and m for H_s, while C is dimensionless. The scaling parameter B implies the characteristic magnitude of the distribution, i.e., it is linearly proportional to mean, median and mode. The parameter B also determines the spread of distribution, i.e., given a fixed value of C, larger B implies broader distribution with a wider extent of coverage. On the other hand, the shape parameter C determines the shape of function curve. Given a fixed value of B, C does not change the extent of coverage, but it changes the shape. For example, typical distribution for U, T_p, and H_s has a bell shape curve, which means $C > 1$. The larger value of C, the narrower the shape, and vice versa.

2.4.1 Results

By using a two-parameter Weibull regression, the scaling parameter B and shape parameter C for wind speed U, significant wave height H_s, and peak period T_p are computed at each grid point and plotted as seasonal contour maps.

To plot seasonal contour maps, the first step is to identify the definition of "shamel season". However, there is no consistent definition of month period for wintertime and summertime shamal seasons [19, 83]. Wintertime shamal is commonly known as a flexible range beginning from November, while summertime shamal, much rarely mentioned in literatures, is known as a flexible range from June to September. In order to make sure the period of wintertime and summertime shamal, we run the monthly Weibull regressions for U, H_s and T_p over 5 years, and plot the contour maps for B and C for each month. By comparing the patters, we found that the patterns out of November to March are pretty similar, and also the patterns out of June and July are similar. The other periods show distinct pattern changes. Therefore, in this research we use the definitions for the periods of shamal seasons:

- *Winter*: November to March
- *Spring*: April to May
- *Summer*: June to July
- *Fall*: August to October

According to the definitions, the seasonal contour plots can be obtained. Figure 2.3 and Figure 2.4 show the seasonal contour maps for wind speed U respectively for Weibull scaling parameter B and shape parameter C; Similarly, Figure 2.5 and Figure 2.6 for significant wave height H_s; Figure 2.7 and 2.8 for peak period T_p. The discussions and conclusions can be remarked as below.

- According to Figure 2.3, due largely to the different shamal seasons, the largest and second largest average wind speed U can be found in winter and summer, respectively. For the seasonal spatial features, the radial distribution can be found in the wintertime covering whole basin area in which peak resides at the Iran side. In spring season the magnitude of winds basically retains the distribution similar to winter, but with weakened magnitudes. In addition, strong winds in spring are slightly down shifted to south pass. At the tip of Straits of Hormuz, $B \approx 6.0(m/s)$, which is larger than winter season ($B \approx 5.5(m/s)$).

  Because of summertime shamal, the summertime pattern is largely different from wintertime pattern. The wind speed in summer rises again and has the second largest B values subsequent to winter season. The strong winds in summer are with long and wide distributions covering only the northern area of the entire basin. Because of north-coming shamals, the pattern shows large winds are distributed over most of northern area. In spite of the same coming directions of winds in both summertime and wintertime shamal seasons, the

magnitude of summertime shamal is however much weaker and has no enough power to bring a southern radial distribution with a distinct peak value. In fall season, the wind is pretty calm for all basin area.

- Figure 2.4 shows the contour map of Weibull shape parameter C for wind speed U. It is found that irrespective of which season, higher C values are generally distributed along south coast near Saudi Arabia, which implies narrower shape of Weibull curves. It corresponds to the distributions of B in which for all seasons smaller winds are distributed along south coast.

- According to Figure 2.5, the seasonal distributions of H_s are basically associated with the patterns of U. High wind regions correspond to large H_s, and vice versa. Similar to wind speed U patterns, shape parameter B for H_s has radial patterns in which peak values reside at the locations near Iran-side coast. Low waves are mainly distributed along the southern coastlines. Because of shamal seasons, winter and summer respectively have the largest and the second largest H_s, in terms of large B values.

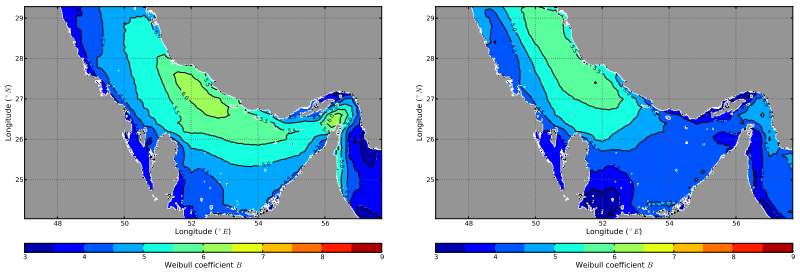
- Similar to U distributions, because of low waves usually distributed in nearshore areas, for all seasons higher C values are distributed along coastlines, particularly along the southern coast. It is worth noting that at the eastern nearshore region of Qatar, C has particularly high values (narrow shape), i.e., $C > 2.2$ for most of time in a year except summer since more strong winds are at northern basin. The reason of sharper distributions at this area is because of low H_s at lee side of waves for north-coming winds. For example, at Doha Port, $H_s < 0.6m$ results in pretty sharp curves of Weibull regression, i.e., $C = 2.482$ for winter, $C = 2.271$ for summer, and $C = 2.513$ for non-shamal season (regression for combined values of spring and fall).

- The seasonal contour maps of B for T_p shows evidence of long-fetch wind wave development. Because of the shallow, flat, and long basin, for all seasons the magnitudes of T_p appear to increase southward along the central axis of Gulf, which results in spatial distributions almost constant across the gulf, varying primarily along the long axis. Due to the seasonal characteristics of the shamal, the largest and second largest T_p can be respectively found during winter, ranging in $B = 3.5 - 5.25(s)$, and the summer, ranging in $B = 3.0 - 4.75(s)$. Similarly, for spring and fall, scaling parameters have the range of $B = 3.0 - 4.5(s)$ and $B = 3.0 - 4.0(s)$, respectively.

- According to Figure 2.8, it can be found that, within the main basin area (inside the Straits), the patterns of seasonal contour maps of C for T_p roughly follow the distributions of parameter B of wind speed U. The area of stronger winds (higher B in Figure 2.3) corresponds to narrower shape (higher C) of T_p, and vice versa. In addition, for all seasons the symmetric patterns can be found centering at the area of peak values. In winter, for example, a span-axial symmetrical pattern can be found in which $C \approx 3.0$ at the top and bottom of the basin and gradually rise to $C \approx 4.5$ at the north of Qatar. That is, along the long axis of the basin the shape of Weibull regression curve of T_p gradually changes from wide to narrow, and back to wide. This fact implies that stronger winds result in more concordant wave periods, irrespective of the magnitude of the actual T_p values.

2.5 The effects of bathymetry

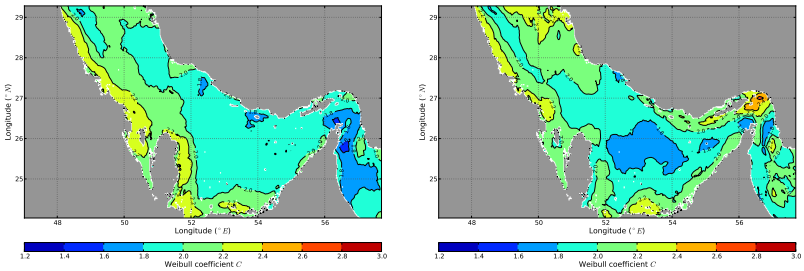
The next study concerns understanding the role of bathymetrically induced refraction and breaking in the Persian Gulf, as well as the local effects near Qatar, on the seasonal variability of wave statistics. Long-term hindcasting helps us to discover



(a) spring (b) summer

(c) fall (d) winter

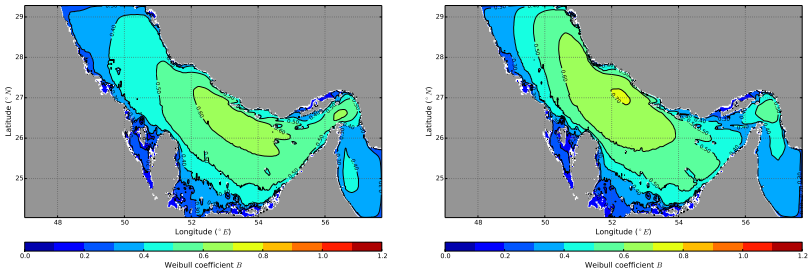
Figure 2.3: Weibull scaling parameter B–wind speed (m/s)



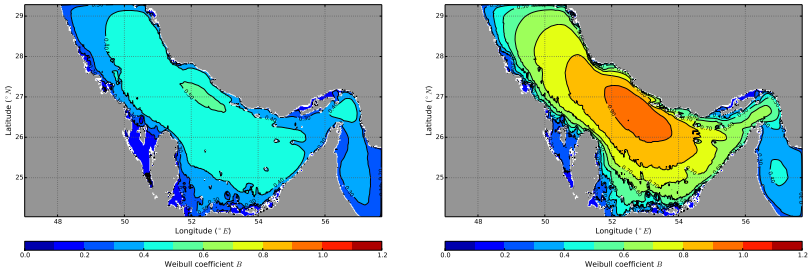
(a) spring (b) summer

(c) fall (d) winter

Figure 2.4: Weibull shape parameter C–wind speed U

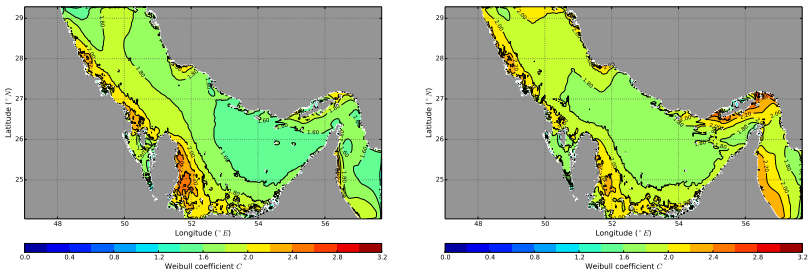


(a) spring (b) summer

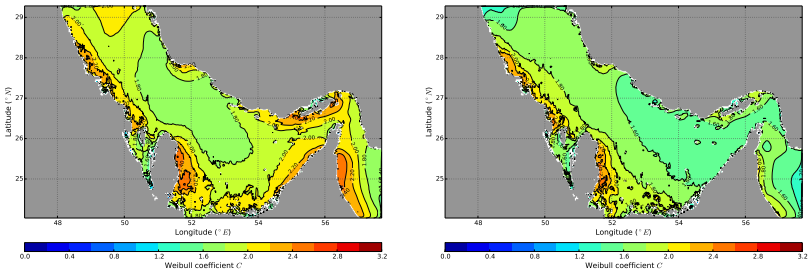


(c) fall (d) winter

Figure 2.5: Weibull scaling parameter B–significant wave height H_s (m)

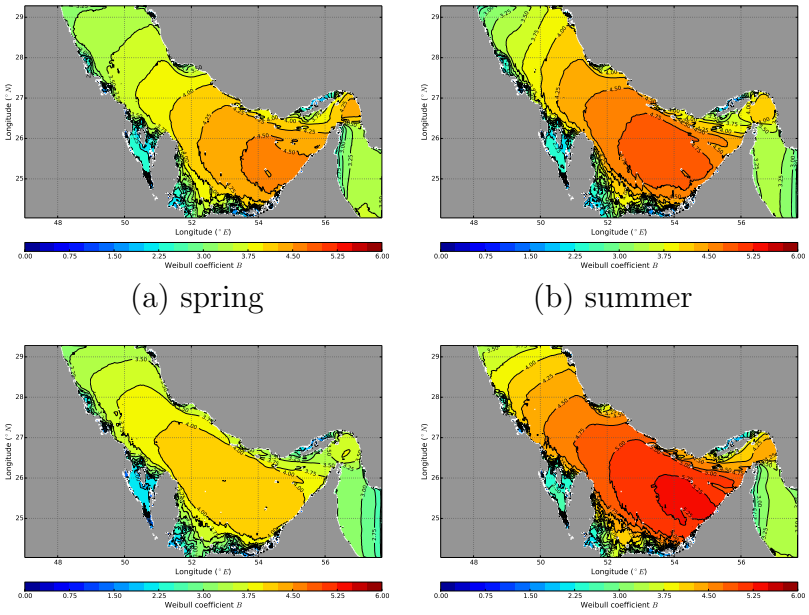


(a) spring (b) summer



(c) fall (d) winter

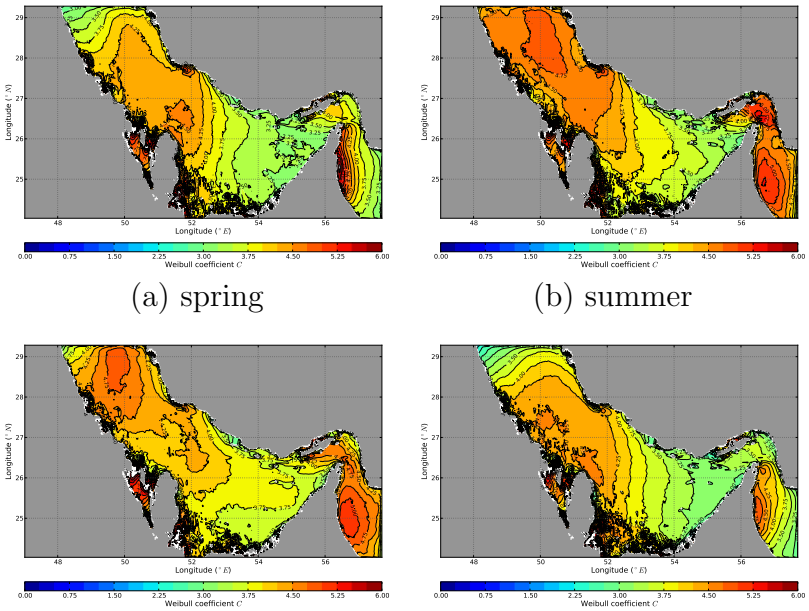
Figure 2.6: Weibull shape parameter C–significant wave height H_s (m)



(a) spring (b) summer

(c) fall (d) winter

Figure 2.7: Weibull scaling parameter B–peak period T_p (s)



(a) spring (b) summer

(c) fall (d) winter

Figure 2.8: Weibull shape parameter C–peak period T_p

the general properties for wind wave processes, but has often been done at discrete locations rather than over spatial domain. With the high spatial resolution data sources used for wind forcing, the general spatial features of wave climates at the basin scale can also be determined.

In this study we employ the identical model setting for long-term hindcasting used in section 2.4. However, to investigate the effects of bathymetry, in addition to the normal hindcasting case (denoted as **origin** below), two alternative scenarios are carried out as well. One is to switch off the refraction (denoted as **noRefc** below) and the other is to switch off depth-induced wave breaking (denoted as **noBrek** below). The remainder of the configuration is identical to the **origin** case. Although the absence of observations disallow us from comparing the results to determine absolute skill, we can obtain valuable information on model skill by relative comparisons between the standard hindcast simulations and the alternative cases.

2.5.1 Results

Here we examine the basin-scale differences in wave energy between **noRefc**, **noBrek** cases and **origin**. The average energy density per unit horizontal area is defined by

$$E = \frac{1}{8}\rho g H_s^2 \tag{2.9}$$

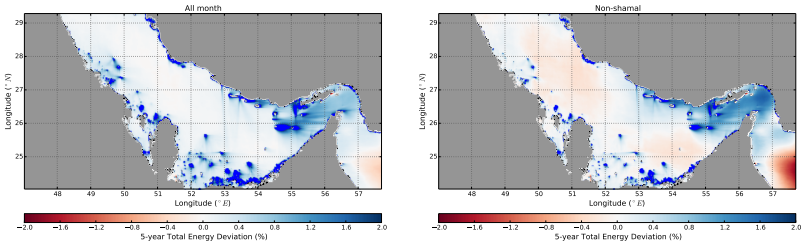
where the specific weight ρg is regarded as a constant in time, and the energy E is in the unit of J/m^2. Therefore, given a grid point at $\mathbf{x}_i$, the 5-year total energy deviation (TED) for case **noBrek** or for case **noRefc** can be approximated by

$$TED(\mathbf{x}_i) = \frac{\sum_n \bar{E}(\mathbf{x}_i, t_n) - \sum_n E(\mathbf{x}_i, t_n)}{\sum_n E(\mathbf{x}_i, t_n)} = \frac{\sum_n \bar{H}_s^2(\mathbf{x}_i, t_n) - \sum_n H_s^2(\mathbf{x}_i, t_n)}{\sum_n H_s^2(\mathbf{x}_i, t_n)} \tag{2.10}$$

where n loops over all values during the length of the simulation. The overbar $\bar{(\cdot)}$ indicates the values given by case **noBrek** or case **noRefc**, while variables without an overbar refer to the values from reference case **origin**. Therefore, as long as the time series of significant wave height H_s is given, the mean energy deviation can be computed.

Figure 2.9 and Figure 2.10 show the maps of percentage of 5-year TED, respectively for case **noBrek** and case **noRefc**. The 5-year total energy deviation TED is calculated for each grid point and plotted as seasonal contour maps. TED due to wave breaking is in the range of $\pm 2\%$, mainly distributed in the Straits of Hormuz, behind islands, and the nearshore regions. Depth-induced wave breaking in the main basin is not immediately apparent, but in the non-shamal season it is manifest in the transition region between the Straits of Hormuz and the Gulf of Oman due to the steep drop of depth. On the other hand, TED due to refraction (**noRefc**) is in the range of $\pm 20\%$, an order of magnitude larger than that due to breaking (**noBrek**). In addition to the regions behind islands and in the nearshore area, non-zero TED due to refraction also can be found in most shallow area of the main basin, particularly in the southern area $(24 - 26^{\circ}N)$ in the east of Qatar $(51.5 - 55^{\circ}E)$. In contrast to wave breaking, depth-induced refraction is affected by the effect of bathymetry in the offshore region, as well as by the effect of coastlines. For example, in the northeastern region of Qatar, TED is $< -20\%$ in the wintertime of all 5 years.

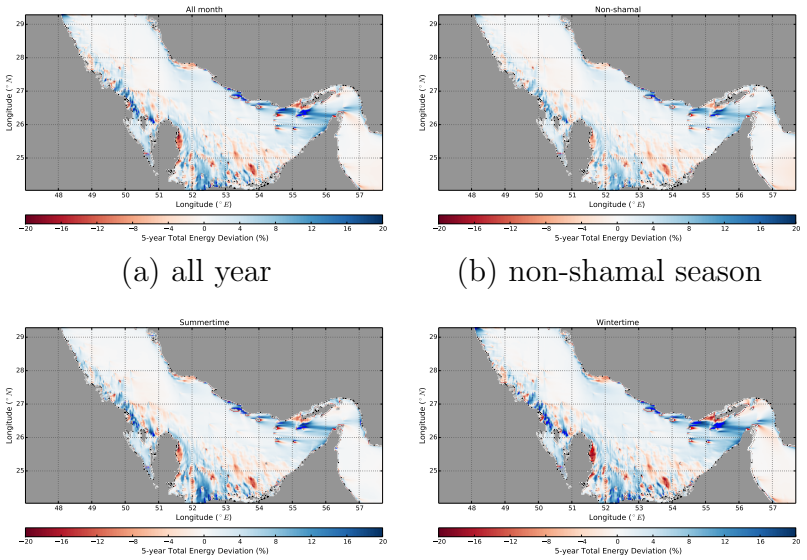
In summary, understanding the effect of shamals on the wind waves in Persian Gulf is important for the energy industry. Compared to recent studies, this work is the first to employ the high-resolution COAMPS wind field, as well as long-term hindcasting, to quantitatively characterize the wind-wave seasonal and spatial features due to bathymetric effects.



(a) all year (b) non-shamal season

(c) summer (d) winter

Figure 2.9: Five-year total energy deviation (%) for case **noBrek**



(a) all year (b) non-shamal season

(c) summer (d) winter

Figure 2.10: Five-year total energy deviation (%) for case **noRefc**

2.6 The effects of swell boundary, wind sources, and domain size

As addressed in section 2.3.4, one practical consideration of wave modeling in a large, yet confined, area such as the Persian Gulf includes the treatment of the wind forcing, particularly when different sources are used. Measurements of winds generally provide relatively high temporal resolution (less than one hour between successive measurements) but are coarsely resolved in space, and are mostly located in coastal areas, e.g., the QTRSTA wind source. In contrast, wind information from most available databases generated by model hindcasts replicate the spatial variability of windfields but often have relatively coarse output temporal resolution, e.g., COAMPS [32] and NCEP wind sources [2].

Another consideration is the configuration of the numerical grids and the implementation of boundary conditions. This becomes a concern when determining nesting configurations for models in order to propagate swell generated from remote weather events to the nearshore. Recent work [64] has shown that multiple nests are needed to reliably capture swell fields in the Southern California Bight, as outer swells are brought into the inner domain by ensuing nested grids. The SWAN model includes utilities in the code which greatly facilitates grid nesting, thus allowing for high resolution only in areas where it is warranted (coastal areas, for example) without using curvilinear or finite element grids (which typically require additional gridding software). While swell may be an important consideration for areas bordering the Pacific Ocean [64], it is not evident how important swell might be for a confined area such as the Persian Gulf.

In this study, we investigate wind and swell waves around the coast of Qatar during October and November 2010, in an effort to determine the importance of various modeling procedures and physical processes on nearshore wave energy. In

particular, we wish to determine the following:

- The importance of the characteristics (spatial, temporal) of the wind forcing on the nearshore wave environment.
- The importance of remotely-generated swell and wind sea on the nearshore waves.

A multi-level grid setup is used for the model, in which a succession of nested grids are used to propagate waves from their generation in the larger Gulf area to the coast of Qatar. A variety of wind fields are used, from hindcast fields from the COAMPS model and NCEP, to local observations QTRSTA located at the end of a maintenance pier. We use the winds to force waves and determine the effect of various grid configurations, nesting options and source of winds on the wave statistics at the maintenance pier. The end result is the establishment of a modeling methodology for prediction of the nearshore wave environment which accounts for the relevant processes affecting wind wave generation for the area.

The time frame used for this study encompasses October and November 2010. While [8] has determined that the month of October is generally a weak shamal month, this time frame was chosen because data for this time from all three sensor platforms (sonic anemometers, video cameras and weather station) were made available.

2.6.1 Multilevel cases and grid configuration

The numerical grid used for the SWAN model was set up in spherical coordinates, which best suits the large size of the Persian Gulf as well as the format of the bathymetric input. To study the effects of grid nesting and boundary condition specification on the nearshore waveheights, three levels of nested computing domains

were developed. Level-1 (L1) is the outermost domain, which coincides with the area of the entire Gulf; Level-2 (L2) covers the Qatar peninsula; Level-3 (L3) is the smallest domain only covering the area of our pier and Doha port. Waves can be seamlessly propagated from outer domain (offshore) to inner domain (nearshore) by applying nested swell boundary conditions, a built-in option in SWAN. L1, L2, and L3 grids are shown as Figure 2.11, while Table 2.2 lists the detailed properties for each domain.

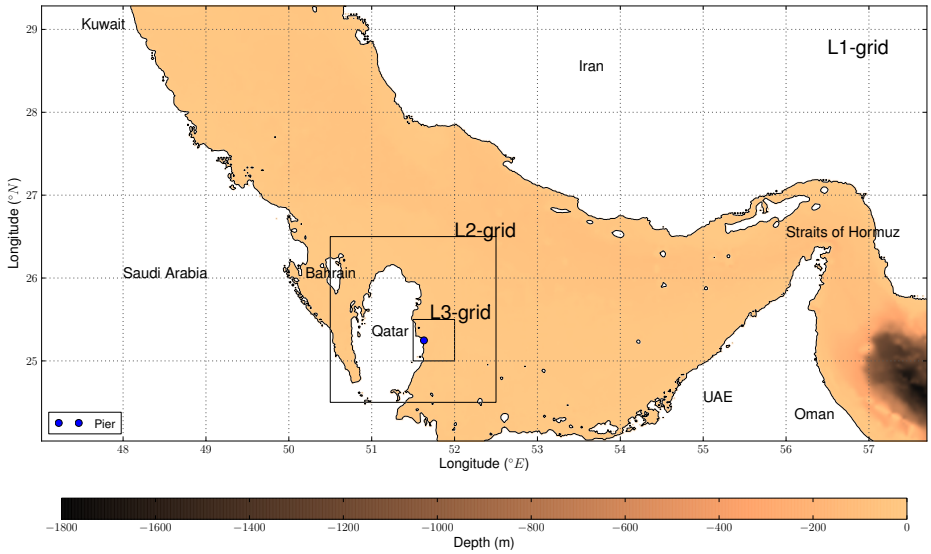


Figure 2.11: Bathymetry and computational domain of the study area

The grid resolution of L1 is identical to the resolution of bathymetry (one arc-minute in latitude and longitude). The grid resolution of L2 is four times that of L1, while the resolution of L3 is four times that of L2. The bathymetry used in

Table 2.2: Nested domains

Grid	L1	L2	L3
Origin (° E, ° N)	$47°1', 24°2'$	$50°30', 24°30'$	$51°30', 25°$
x-length (longtitude)	$10°41'$	$2°$	$30'$
y-length (latitude)	$5°15'$	$2°$	$30'$
Number of x-cells	641	480	480
Number of y-cells	315	480	480
Δx	$1'$	$0.25'$	$0.125'$
Δy	$1'$	$0.25'$	$0.125'$
Δt (min)	20	10	5

these simulations is at the one arc-minute input resolution for all grids; the SWAN model automatically interpolated the bathymetry to the resolution germane to the computational grid.

2.6.2 Model testing

In this section we outline our testing scheme for determining the importance of forcing characteristics (wind and incoming waves) on nearshore wave conditions. As mentioned previously, we anticipate that the dominant factors affecting the prediction of nearshore conditions at the pier are:

1. *The nature of wind forcing.* As described earlier, we use COAMPS analysis and NCEP re-analysis hindcast winds, as well as QTRSTA winds, as our wind sources. The QTRSTA data is applied as a spatially-constant field with high temporal resolution, while the COAMPS and NCEP hindcasts are spatially-variable wind fields with coarser temporal resolution.

2. *The inclusion of boundary conditions from larger grids.* While related to the question of domain size, this factor addresses the importance of incoming swell

and wind sea on the wave environment inside a modeled domain. In this test we control the application of boundary conditions along L2 and L3 model domains to determine their effect on the prediction of nearshore wave conditions. We note here that this testing is hierarchical; either no boundary conditions were applied to the L2 or L3 grids, or boundary conditions were applied to both grids. For the cases that use QTRSTA as wind forcing in L2 and L3 domains, plus any boundary conditions from outer domain

3. *The size of the modeled domain.* L2 and L3 respectively represent our two primary domain sizes, with L1 generally involved only in supplying forcng conditions to L2. Variation of the domain sizes help determine whether local domain modeling is tenable over an area as generally calm as the Persian Gulf, or if the entire gulf area must be included.

Table 2.3 lists all possible combinations for the three listed testing conditions, which comprise a total of eight cases for wave hindcasting. As only one of the above factors was altered for each run, it is possible to perform inter-comparisons between various simulations. For example, by comparing the differences between waves purely driven by winds (cases 3, 5, 7, 10, 12, and 14) and waves driven by winds and incoming waves from the boundaries (cases 4, 6, 8, 9, 11, 13, 15 and 16) , the effect of incoming waves on the nearshore wave predictions can be determined.

2.6.3 Results

Both the hindcast winds (COAMPS and NCEP) and measured (QTRSTA) winds were input into the SWAN model for the various grid configurations listed above and run, with output at the location of the pier measurement station. The wave bulk parameters of interest - significant wave height H_s, peak period T_p and mean direction θ_m - were output by the model once every three hours and histograms of

Table 2.3: Simulation cases

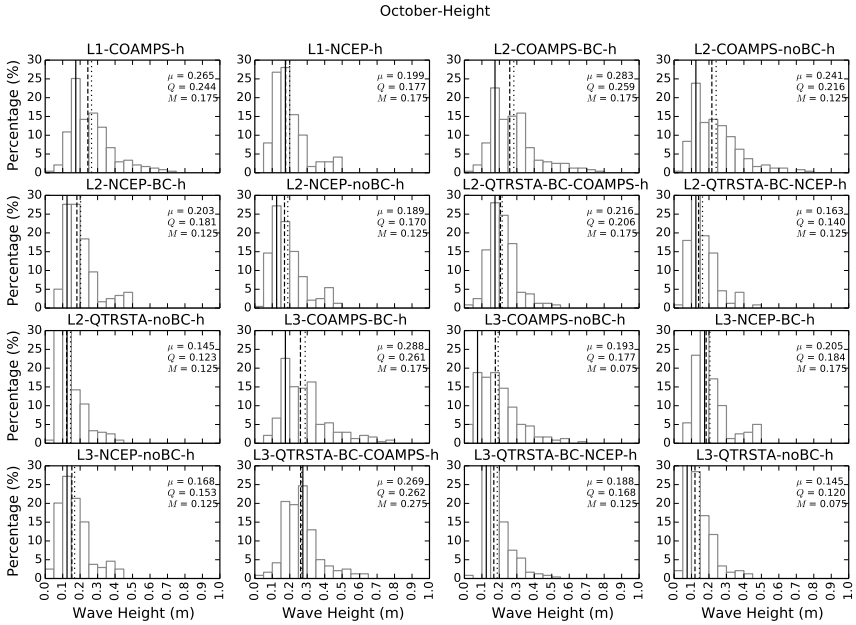
No.	Case Name	Grid			Wind Force			Boundary Conditions		
		L1	L2	L3	COAMPS	NCEP	QTRSTA	open	swell-COAMPS	swell-NCEP
1	L1-COAMPS	X			X			X		
2	L1-NCEP	X				X		X		
3	L2-COAMPS-noBC		X		X			X		
4	L2-COAMPS-BC		X		X				X	
5	L2-NCEP-noBC		X			X		X		
6	L2-NCEP-BC		X			X				X
7	L2-QTRSTA-noBC		X				X	X		
8	L2-QTRSTA-BC-COAMPS		X				X		X	
9	L2-QTRSTA-BC-NCEP		X				X			X
10	L3-COAMPS-noBC			X	X			X		
11	L3-COAMPS-BC			X	X				X	
12	L3-NCEP-noBC			X		X		X		
13	L3-NCEP-BC			X		X				X
14	L3-QTRSTA-noBC			X			X	X		
15	L3-QTRSTA-BC-COAMPS			X			X		X	
16	L3-QTRSTA-BC-NCEP			X			X			X

percent occurrence derived. In addition, we also output the significant swell height H_{swl}, in which we defined swell as wave energy with a frequency less than 0.167 Hz [67].

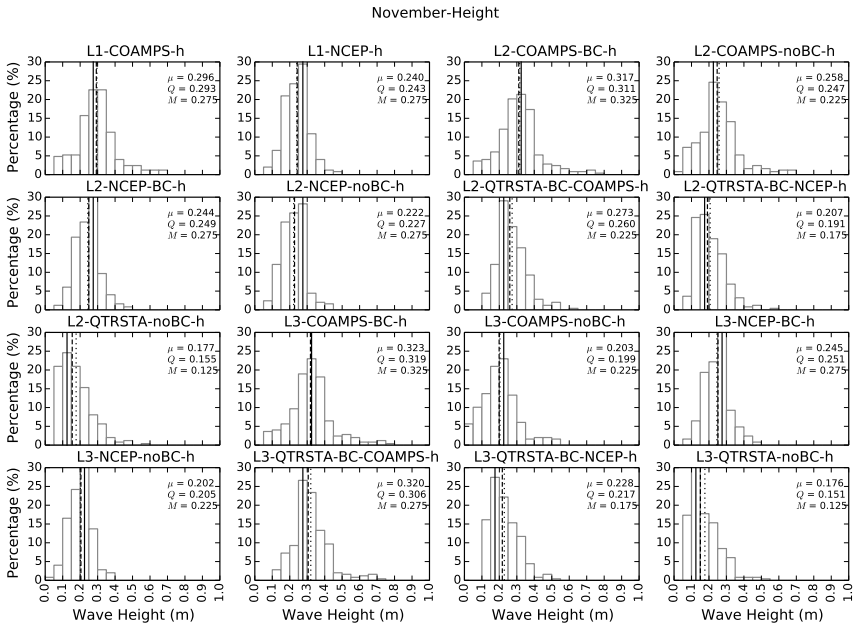
Monthly statistics (for October and November 2010, separately) are plotted as figures for T_p, H_s, H_{swl}, and θ_m, respectively. Figure 2.12 shows the example for H_s. In addition to histograms, correlation coefficients and RMSDs between results of the various run configurations were calculated to help quantify the similarities in the dependencies of these wave parameters on aspects of the forcing (boundary conditions or wind variability).

Our first set of comparisons looked at the effect of using varying wind fields. The results for significant wave height H_s show different values for the peak of the distributions for either COAMPS or QTRSTA runs, with a higher percentage of wave heights. This is thought to be a possible indication of the effect of strong basin-scale events during November, as only the COAMPS or NCEP winds include any basin-scale events. The wave peak periods T_p and mean angles θ_m also display some evidence of waves generated by a strong basin-scale wind; hindcast winds show longer period and narrower distribution of wave angles than QTRSTA.

The second set of comparisons concerned the use of boundary conditions. By comparing the cases with and without boundary conditions (`L2-COAMPS-BC` to `L2-COAMPS-noBC`, for example) it can be seen that more energetic conditions exist at the measurement pier with the boundary conditions included; in many cases, the peaks of the distributions of H_s are shifted to higher values. These trends appear to be independent of the wind fields used, and reflect the use of COAMPS or NCEP winds for the L1 domain for the simulations for L2 and L3 domains. Peak periods also show the effect of boundary conditions; those simulations with boundary conditions tend to have a higher precentage of wave conditions at the nearshore location with periods



(a) October



(b) November

Figure 2.12: Monthly Statistics of Occurrences–Significant Wave Height H_s. Dotted line: mean μ; Dashed line: median Q; Solid line: mode M.

exceeding $4s$, and also have distribution shapes similar to that from the basin-scale L1 grid.

Finally, we investigated the effect that domain size (L2 or L3 domains) have on the nearshore wave conditions. It appears that the effect of domain size (which is essentially the size of the domain over which the wind generates waves) is mitigated somewhat by the use of boundary conditions, which connects locally-generated waves with those generated remotely over L1; these serve to reduce the differences between L2 and L3 cases. For cases without boundary conditions, however, the effect of the domain size is clear, as conditions generated over L2 are distinctly different (for all parameters) than those generated over L3.

2.7 On-site study using video imagery

To verify the numerical results, we employ video imagery to extract wave properties from 35 videos taken by the camera mounted at the maintenance pier. For single camera video imagery, Figure 2.13 shows the preparation procedures before analyzing wave properties. Note that for the rectification steps we will follow [33]:

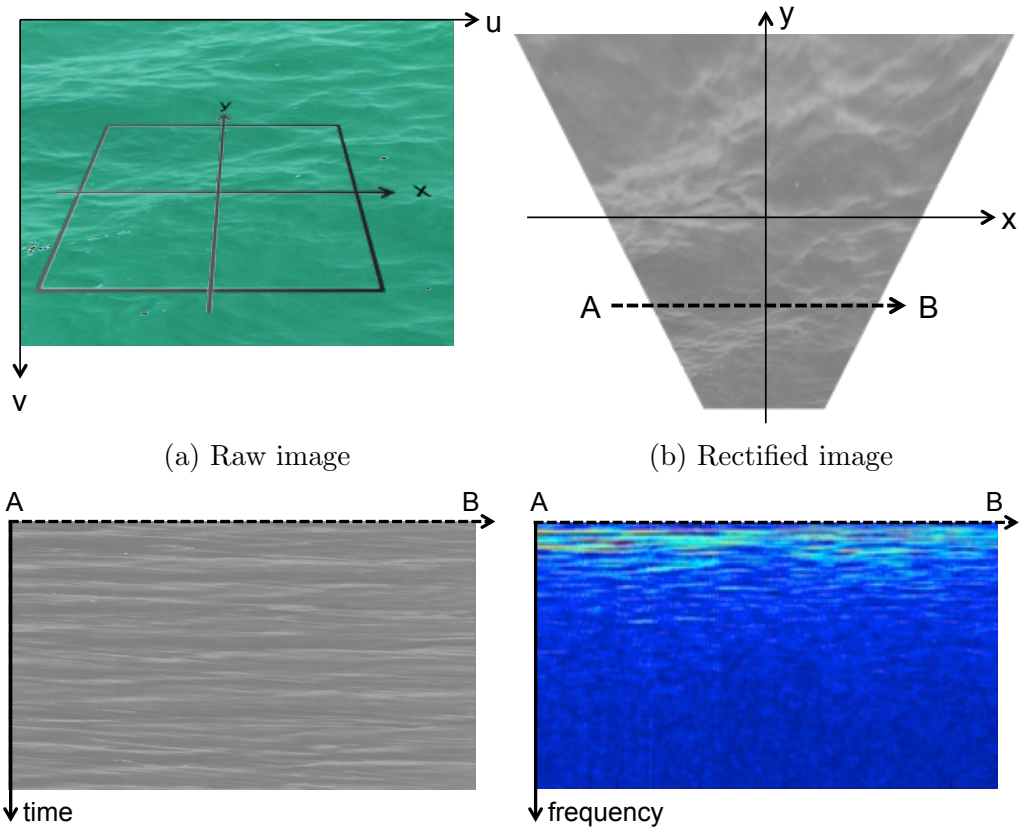
1. Perform lab camera calibration and calculate necessary parameters: effective focal length $f = 12.5(mm)$, $\lambda_u = 0.99995$, and $\lambda_v = 1$
2. Rectify the image by "re-sampling" technique in which the transformation functions are given by

$$\begin{cases} \text{camera to world:} & \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} L_1 - L_9 u & L_2 - L_{10} u \\ L_5 - L_9 v & L_6 - L_{10} u \end{bmatrix}^{-1} \begin{pmatrix} u - L_4 \\ v - L_8 \end{pmatrix} \\ \text{world to camera:} & u = \dfrac{L_1 x + L_2 y + L_4}{L_9 x + L_{10} y + 1} \qquad v = \dfrac{L_5 x + L_6 y + L_8}{L_9 x + L_{10} y + 1} \end{cases}$$

   where geometric coefficients L_n are function of camera parameters (f, λ_u, λ_v), camera position (x_c, y_c, z_c), and camera rotation angles (ϕ, τ, σ). For example, Figure 2.13(a) and Figure 2.13(b) respectively show the raw and rectified images.

3. Choose a scanline horizontally or vertically, and make a timestack. For example, a horizontal scanline $\overline{AB}$ is chosen in Figure 2.13(b), and Figure 2.13(c) shows its timestack in which every row represent a temporal record of $\overline{AB}$.

4. Perform FFT to each column (time dimension) to convert the timestack to frequency domain. For example, Figure 2.13(d) shows the resulting FFT spectrum of the timestack in Figure 2.13(c).



(a) Raw image (b) Rectified image

(c) Timestack (d) Timestack FFT

Figure 2.13: Video imagery preparations: (a) raw image; (b) rectify image in terms of transformation functions; (c) make a timestack from a scanline $\overline{AB}$; (d) apply FFT to timestack.

In terms of the timestack FFT coefficients $Y(x, f)$, 1D analysis can be performed. By [73], mean wave frequency f_{mean} and wavenumber k_x can be estimated. As long as

f_{mean} and k_x are available, wave speed and wave angle can be accordingly estimated as well. The frequency can be estimated by

1. Set up lower and higher bound to filter out low frequency trends and noise: $f_{min} = 0.05(Hz) \qquad f_{max} = 2.5(Hz)$.
2. Calculate mean frequency using weighted average method:

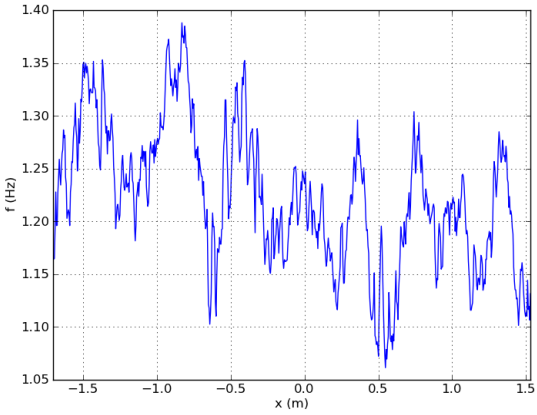
$$f_{mean}(x) = \frac{\sum_{i=f_{min}}^{f_{max}} |Y(x, f_i)| \, f_i(x)}{\sum_{i=f_{min}}^{f_{max}} |Y(x, f_i)|} \tag{2.11}$$

Wavenumber k_x can be estimated by using CEOF (Complex Empirical Orthogonal Function):

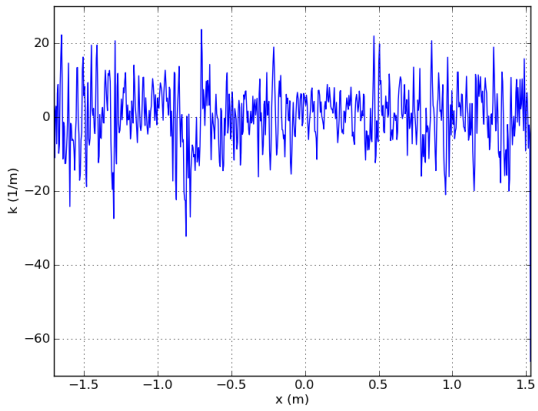
1. Calculate cross-spectrum matrix $Q_{ij} = \left\langle \overline{Y(x, f_i)} Y(x, f_j) \right\rangle$
2. Normalize Q_{ij} and obtain $\mathbf{Q}$
3. Perform eigenvalue analysis: $[\mathbf{V}_i, \lambda_i] = EOF(\mathbf{Q})$ where $\mathbf{V}_i$ and λ_i are respectively i-th eigenvector and eigenvalue.
4. Use the first mode (the largest percentage) $\mathbf{V}_1 = a(x) + ib(x)$ to estimate amplitude $A(x) = \sqrt{a^2(x) + b^b(x)}$ and phase $\phi(x) = \tan^{-1}\left(\frac{b(x)}{a(x)}\right)$
5. Calculate wavenumber vector $k_x(x) = \frac{d\phi(x)}{dx}$

Moreover, a block analysis can be done by repeat the above procedure to traverse each column and each row of the rectified image. An example of video analysis result is shown as Figure 2.14. The results show that the mean frequency can be captured clearly. The average value is about 1.20 to 1.25 Hz, and there is no value below 1 Hz, which implies that only pure wind sea waves are captured while the swells are in absence. It is because all the video sources taken on site are neither sufficient long nor with sufficient shooting range. Every video is only 12 seconds long, and the camera only shoots on the range smaller than 5 meters. It also accounts for the noisy spatial distribution of wave angle, and similarly for wavenumber results, since

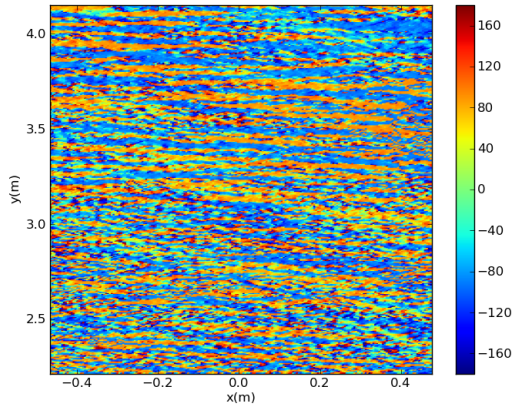
it is too difficult to capture concordant waves for pure wind sea within a such small shooting room. In addition to pure wind sea waves, to validate the numerical results, we also need more information regarding swells extracted from video. To achieve this, longer video and larger shooting room are expected in the future experiments. Furthermore, stereo video imagery technique is even more helpful since the wave height can be exactly identified. Having the wave heights, the spectrum is possible to be measured from video imagery and finer comparisons can be done.



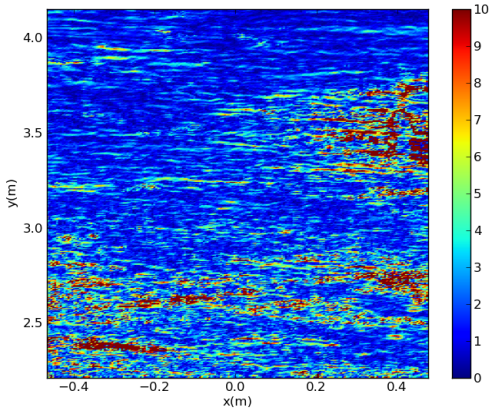
(a) Mean frequency f_{mean}



(b) Wavenumber k_x



(c) Wave angle (degree)



(d) Wavenumber magnitude $k = |\mathbf{k}|$

Figure 2.14: An example of video analysis result: (a) and (b) are taken along x-axis; (c) and (d) are block analysis results

3. PART 2—SPX: A GENERIC PDE FRAMEWORK FOR STRUCTRUAL GRIDS USING C++1Y AND CONCEPT-BASED DESIGN

3.1 Core principles and scopes

SPX is a general numerical framework for solving PDE on structured domain. According to general practice, there are three main parts for the core design: 1) grids and domain, which can be comprised of arbitrary dimensions; 2) general representations of differential operators; 3) implicit and explicit solvers. Typically implicit solvers include stationary solver based on stencil operators while explicit solvers are for ODE and time integrations that support different schemes. With the term "general", the features of SPX are:

- Support general differential operators, including the composite operator such as linear combinations. For example, the code `2*dx+dy(dz)` represnets $2\frac{\partial}{\partial x}[\cdot]+\frac{\partial}{\partial y}\left(\frac{\partial}{\partial z}[\cdot]\right)$, and can be applied to any node.

- Support both rectilinear and curvilinear domains. PDE can be generally built at any given node.

- Support abstract differential basis, i.e., periodic or non-periodic finite difference basis for any order differentiation, and spectral basis such as Fourier, Chebyshev, and Legendre basis. Any general linear differential operator can be composed by mixing of different schemes.

- Support commonly used time integration schemes. Time marching should be generically designed and should not be coupled with any other particular SPX components. Users can easily assign and switch time schemes, regardless the kind of differential equations, operators, grids and domains.

- Support linear and non-linear implicit solvers, or just simply called "solvers". The design idea is similar to time marching. Solvers should be designed as independent and general components. Users can easily specify any solvers for any type of equations and operators.

- Provide robust infrastructure, i.e., a highly-efficient numerical array.

Figure 3.1 shows the UML package diagram for the layout of SPX framework. In the following sections will explain the designs and implementations for each subsystem and subpackage.

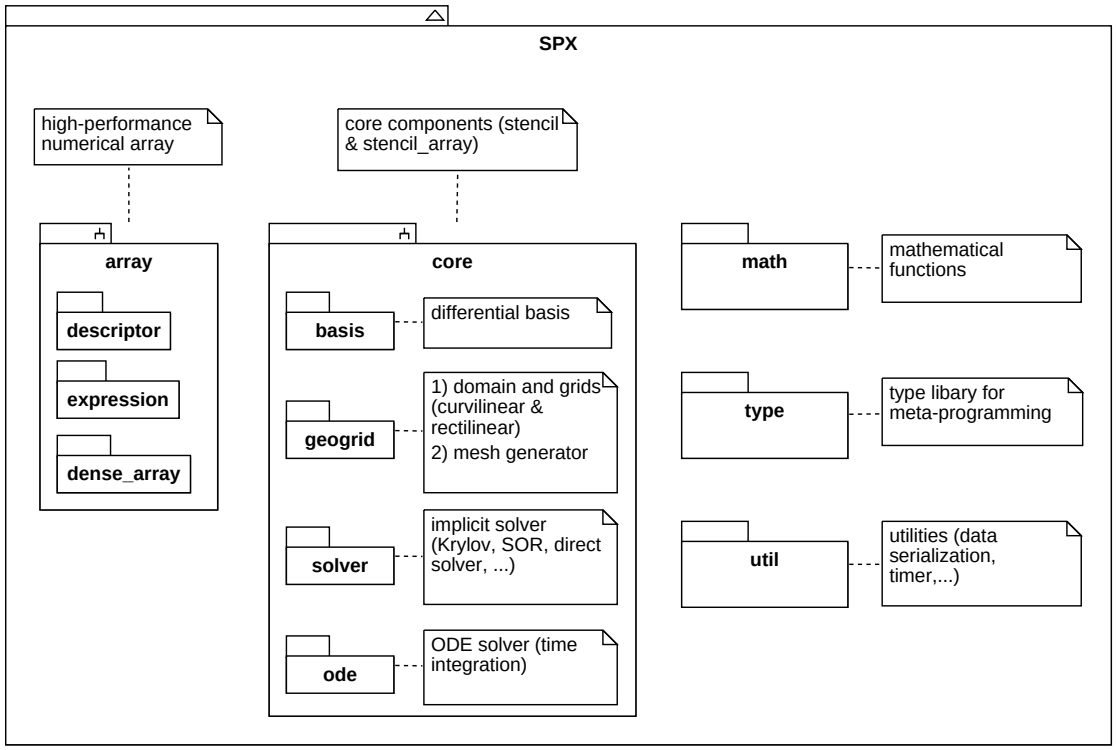


Figure 3.1: UML package diagram for the layout of SPX framework.

3.2 Generic programming and C++-Concepts in C++1y

Generic programming (GP) is a programming paradigm that can compensate for the side effects of from runtime type binding such as object-oriented (OO) programming. Instead of dynamic type binding, GP uses a static type system, which means every polymorphism type used in a program should be deduced by the compiler and determined at compile-time. The technique used in GP is usually referred to "template", so GP is sometimes called "template programming". Templates are realized by compilers differently depending on the compiler design for different programming languages, i.e., Java compiler substitutes template types in linking stage, and C# compiler substitutes template types at runtime stage, which all does not fully take the advantage of GP-static type deduction.

C++ [76], invented in 1983 by Dr. Bjarne Stroustrup, is the first language to introduce template to operate generic types in 1988 [74] based on the strong type system of C++. GP by using C++ templates guarantees all generic types can be deduced and determined at compile-time. That is, without the redundancy of runtime type binding, a well-designed GP software package can not only perform super high efficiency for runtime performance but also not sacrifice the type polymorphism. Standard Template Library (STL), first released in 1998, is the C++ standard library using GP to provide generic type containers and algorithms.

Designing GP software framework is usually more difficult than a pure OO framework, because there is no "standard methodology" to guide how to proceed with the GP design for large-scale software; GP-based software developments are mostly based on the practices. Boost [66] is a gigantic and comprehensive C++ GP framework to provide the components and algorithms required in any kind of developments of scientific programs, almost covering all the possibilities. Many components of Boost

have also gone on to affect C++ standards. An important technique, which can be seen as the only systematic methodology being used in GP, adopted in Boost is "meta-programming". Meta-programming is also a kind of template programming aimed at the operations on generic types. For example, type traits are a set of template classes to check the type information for any given generic type; `enable_if<..>` is a template class to select type statically in terms of given conditions at compile time.

Since in C++ generic types are designed to be "truly generic", any particular type can be substituted in any particular template without checking. For example, a generic sorting algorithm, which supposes only being used for "sortable" types, actually can be substituted by any non-sortable types, and of course results in many nonsense errors at compile time or runtime. How to constrain specific types that can be accepted by a given generic algorithm or container at compile time is a challenge and is still an ongoing research topic. The goals are clear: 1) constraint types according to the acceptance requirements; 2) generate human readable compile errors as long as any generic type is incorrectly substituted, and of course, all should be done statically. Unfortunately, there was no intrinsic solution. Typical solutions to coin static constraints for generic types could be done by the use of meta-programming and template partial specializations. This study will not discuss theses details since they were partial alternatives and essentially could not fulfill the goals.

The intrinsic solution is to introduce C++-Concepts [3]. Concepts can programmatically express the requirements for generic types, which implies the necessary of new syntax going to be introduced in C++. It had been discussed and proposed since 2005 [28, 68, 18, 27, 37]. However, since it was overly complex to be understood, ISO committee voted to remove "concepts" from C++0x in 2009, but allowed more years to have a simplified version [65, 75]. A simplified version (called "concept-lite")

was therefore proposed [77, 80, 81, 82] and opinion passed by vote in 2013. In addition, a completely new version of the C++ standard was released in 2011, with minor revisions in 2014 and major revision that will be delivered in 2017, denoted as C++11/14/17 or shortly C++1y for convenience. The use of suffix "1y" is due to many major changes making C++ resemble a new language, compared to its previous main standard in 1998. Some important features are directly associated with GP, e.g., variadic templates and static assertions. Developed by Dr. Andrew Sutton, origin [79] is the first C++1y package that uses concept-lite to redesign STL, as well as provides many useful tools for library development. In SPX we will employ origin as the very important foundation to build up upper level PDE tools.

3.3 High-efficiency numerical array using concept-based design

For any numerical framework, a user-friendly, robust, and high-efficiency numerical array is always required to start. It can be easily imagined how important it is for any operation of basic linear algebra subprograms (BLAS) [45, 17, 16]. Because of no runtime overhead, C++ generic programming offers both high-efficiency and flexibility via static polymorphism. Many successful C++ generic libraries for array or linear algebra have been proposed and have been proven successful, such as Blitz++[89], MTL (Matrix Template Library) [69, 70], MTL4 [26], and Eigen [1]. Thanks to C++1y new features, a better array design is now possible. For example, by using variadic template, element access in arbitrary rank can be treated as arbitrary length of function arguments and resolved at compile time [6], which was not possible previously. SPX array is the first generic array library that not only supports arbitrary rank but also employs concept-based design, particularly emphasizing the new features provided by C++1y.

Comparing to matrix-aimed design such as MTL, Eigen, origin's matrix [76], and

TC++PL4's matrix [76], SPX array emphasizes at the scope of element storage and access via subscription and subarray slicing using concept-based design, which is more similar to Blitz++. The main difference between SPX array and the other matrix-aimed libraries is that SPX array is just the design of *multi-dimensional array* but not really for *matrix*. SPX array is more like *a container of elements*, similar to the containers in STL. The design of SPX array is optimized for *indexing*, *slicing*, and *storage*. On the other hand, although a matrix can be declared as a two-dimensional array, but actually we can do more mathematical design for it since a *matrix* is of the mathematical semantic (but array is not). For example, in Eigen package, the expression template for "matrix (M)" and "vector (V)" are optimized for linear algebra calculation. For instance, in the case of $M \times M \times V$, instead of plain evaluation of $(M \times M) \times V$, by using expression template it can be optimized as $M \times (M \times V)$ so as to significantly reduce the cost of matrix-matrix multiplication. There is no such linear algebra-optimized design in SPX array, since SPX array is currently designed as a container for the elements in hyper-dimensional indexing domain. Due to the differences of fundamental concepts between array and matrix, in the future development, a SPX matrix would be expected to be designed separately in which the linear algebra-focused and -optimized design can be applied.

3.3.1 Dense descriptor

A basic idea for SPX array is to design a multidimensional container with arbitrary rank for any generic type. The underlying elements will be stored in a linear space. Therefore, a "dense descriptor" is an index placeholder and transformer that can map the index space from arbitrary rank into a linear memory offset starting from 0. Given rank N, there are some parameters to represent a general dense descriptor: 1) For index domain, extents $L[N]$ represents the length for each dimension

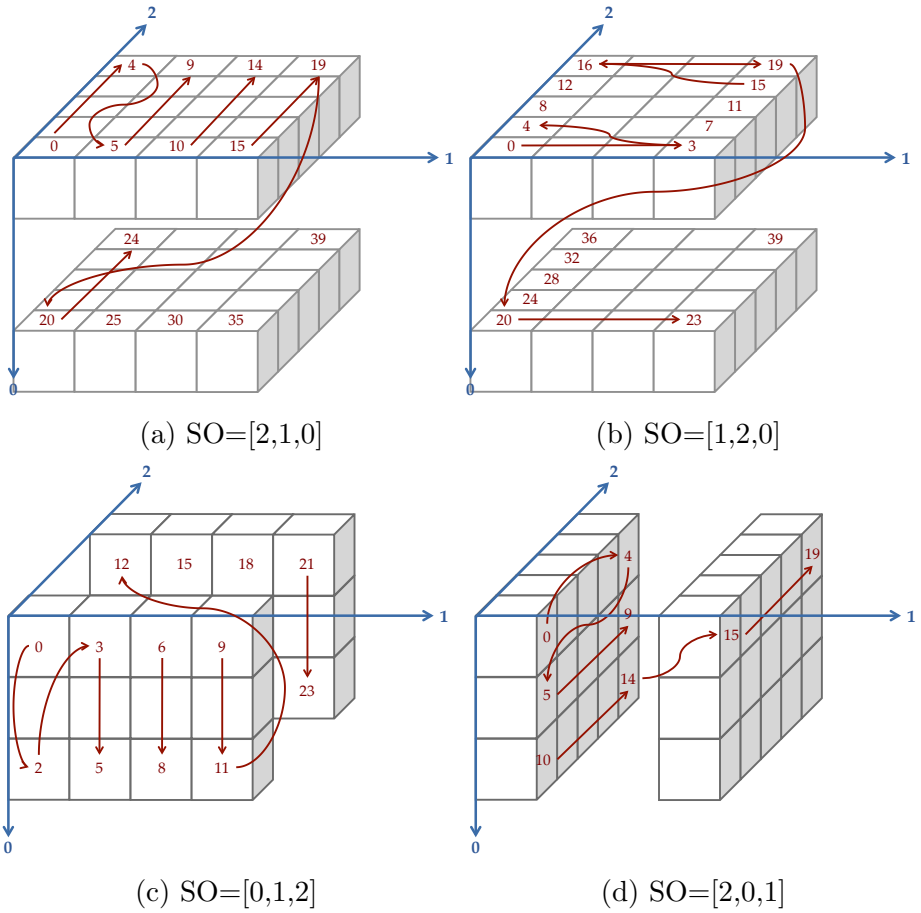
so that $L[0] \times L[1] \times \cdots L[N-1]$ will be the total size (number of elements); index base $B[N]$ represents the index lower bounds to allow the index domain to not necessarily be 0-based, i.e, 1-based for Fortran-like array. 2) For storage and mapping, storage order $SO[N]$ is for the ranking of storage dimension. For example, given a $3 \times 4 \times 5$ array, Figure 3.2 demonstrates the examples for four different storage order. For C-like array, $SO[N]$ is always from highest dimension to lowest one, and vice versa for Fortran-like array. Strides $TR[N]$ is the parameter that for each dimension how many elements should be skipped for a unit index increased at this dimension. p is the offset parameter to ensure the mapped memory offset starting at zero. Having subscript index $I[N]$ bounded between $B[d]$ and $B[d] + L[d] - 1$ for any dimension d, the linear memory offset M is given by

$$M = p + dot(I, TR) \tag{3.1}$$

where M always starts from 0. The relationship clearly shows that $TR[N]$ and p are the parameters actually used to map the subscript index to the linear space. TR and p are dependent parameters and can be pre-calculated. TR can be calculated from SO and L by $TR[SO[0]] = 1$ and $TR[SO[d-1]] = TR[SO[d-2]] * L[SO[d-2]]$. Also, p can be obtained from B and TR by $p = dot(-B, TR)$. That is, only SO, B, and L are independent parameters. As long as a new dense descriptor constructed with the three parameters, TR and p will be updated correspondingly once and repeatedly used many times, so as to minimize the calculation for every time requesting an index subscript.

3.3.2 Concepts for slice and subscript

As mentioned above, as long as the array has been sliced, a new subarray will be constructed. The new subarray will hold the reference of the original storage, but its



(a) SO=[2,1,0] (b) SO=[1,2,0]

(c) SO=[0,1,2] (d) SO=[2,0,1]

Figure 3.2: Four examples for array storage order (SO) for a $3 \times 4 \times 5$ array; (a) C-like storage, and (c) Fortran-like storage.

dense descriptor is updated from the old one so as to make it described by the new indexing domain. Figure 3.3 is the example that shows how to update the descriptor. Assuming that a uniform slice created by `slice(5,17,3)` (starting $s = 5$, ending $e = 17$, and stride $r = 3$) is applied to the dimension d, and the base index of this dimension is $B[d] = -2$, the new memory offset M is given by

$$M = (p + (s - B[d] * r) * TR[d]) + I[d] * (r * TR[d]) = p2 + I[d] * TR2[d] \quad (3.2)$$

Obviously, for the new descriptor the offset parameter $p2$ is updated by $p2 = p + (s - B[d] * r) * TR[d]$ and the stride is updated by $TR2[d] = r * TR[d]$. Also, the new length $L2[d] = (e - s)/r + 1 = (17 - 5)/3 + 1 = 5$. The similar procedure can be applied for any other dimension. The new descriptor also keeps the base index starting $B[d] = -2$ at the dimension d.

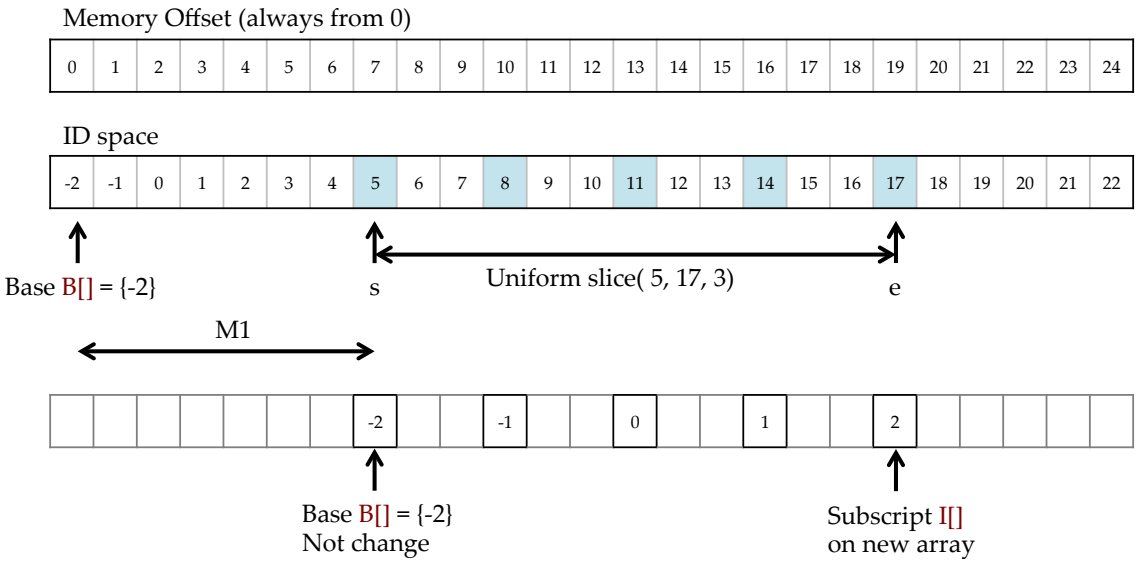


Figure 3.3: Illustration of uniform slicing on a specific dimension.

The example is only for the case of uniform slicing applied to a specific dimension. However, slice and subscript can be very general. The design goal is to provide any dense descriptor with a homogeneous interface looks like:

```
template <Indexable... Args>
  decltype(auto) operator()( Args&&... args ) const
  {
    /* returns any of
     * a) memory offset M if it is subscript of a single element
     * b) a new descriptor if it is slicing a subarray
```

```
    */
}
```

The GNU g++-concept compiler allows a shorthand convention to apply the concept `Indexable` to every argument listed by the variadic template. The return type is automatically deduced depending on the arguments. It could be a memory offset M for a single element access or a new descriptor for slicing a subarray. Therefore, the concept `Indexable` is the most abstract of those considered, since C++ Concept can be defined by either the description of itself or by relying on other existing ones, i.e., by the union of sub-concepts. In this case, `Indexable` is at the root of concept hierarchy, and Figure 3.4 shows the complete hierarchy of all concepts for the design of slice and subscript in SPX.

First of all, any `Indexable` can be only one argument of `Indexable_range` or multiple arguments of `Indexable_argument`, where `Indexable_range` is a range of `Indexable_argument` that can be checked by concept `Range`. For example, it could be a `std::vector<T>` or a `std::initializer_list<T>` whose element type `T` conforms `Indexable_argument`. For `Indexable_argument`, it can be a `Subscript` or a `Slice`. Since each `Indexable_argument` represents a dimension, `Subscript` means subscribing at a specific position at this dimension, while `Slice` means slicing a sub-index domain for this dimension. Accordingly, given multiple `Indexable_argument`, if all are `Subscript`, then return a specific memory offset M for a single element; otherwise, return a new descriptor for a subarray. If all arguments are `Slice`, the new descriptor is of the rank equal to the old one, while if some are `Subscript`, it is reduced slicing; the new descriptor is of the rank equal to the old rank minus the number of `Subscript`, i.e., for a 3D descriptor `d`, `d( slice_all(), slice_all(), 4 )` will return a 2D descriptor.

The concept `Subscript` can be a real integer represented by concept `Can_be_signed`

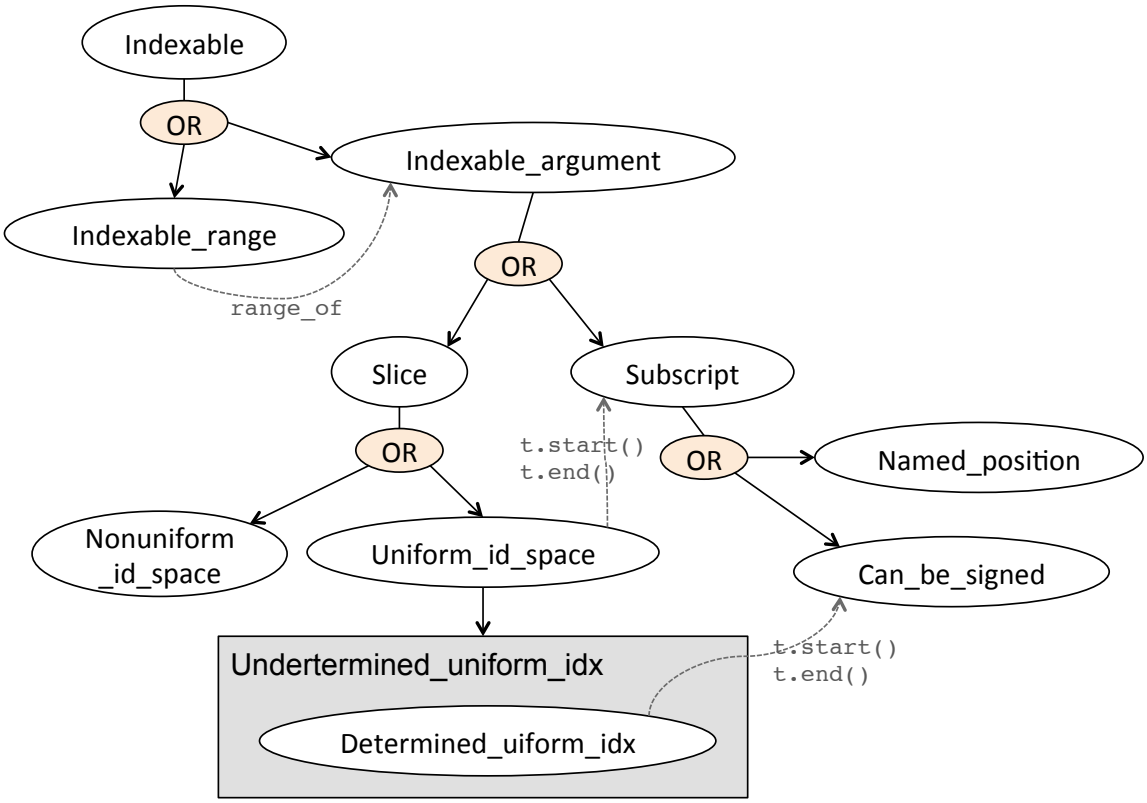


Figure 3.4: Concepts of slice and subscript

or a undetermined position `Named_position`. Since it is not possible to define a strictly mathematical concept for "integers" `Can_be_signed` is to check if the substituted integer type can be converted to a signed type in that all integers should be signed even though the substituted type is unsigned itself. `Named_position` is a design of lazy evaluation to determine the real index at the runtime. For example, SPX allows user to use first(), `last()`, or `half()` to represent the undetermined positions whose index will be calculated at the runtime. It is a very convenient design similar to the numerical array in Matlab or Python that we can subscribe the last element without a specific index number. Also, in SPX implementations, it supports expression, i.e., `half()+3`, `last()-4`, and `(first()+last())/4` all are

valid expressions.

The concept `Slice` can be a uniform index space `Uniform_id_space` or an nonuniform index space `Nonuniform_id_space`. `Uniform_id_space` describes the slice by a starting index, an ending index, and a stride for element stepping. Its starting and ending indices must conform the concept `Subscript`. For examples, slice(2, last(), 2) is valid and `all()` is always identical to slice(first(), last(), 1). `Uniform_id_space` consists of concept `Determined_uniform_idx` and its complement set `Undetermined_uniform_idx`, where `Determined_uniform_idx` clearly defines that integer types are used for both starting and ending indices. `Nonuniform_id_space` describes the index space by a list of real index that can be distributed randomly. As long as any dimension is sliced by `Nonuniform_id_space`, the new descriptor will be a nonuniform dense descriptor whose descriptor information is updated much complicated and can not trivially updated by the way similar to the example explained in Figure 3.3. We will not go through the details. The following examples all are valid:

```
a( 2, 4, 5 );
a( slice(2, 7, 2)), 2, 3 );
a( (first() + half())/2, 4, 2 );
a( all(), slice(4, last(), 2), half() );
```

3.3.3 Concepts for descriptors

According to section (3.3.1) and (3.3.2), we may define the concept `Descriptor` as

```
template <typename T>
  concept bool Descriptor()
  {
    return Dense_storage_major<T>()
        && requires( Main_type<T> t ) {
          typename Size_type<T>;
```

```
        typename Index_type<T>;
        { T::rank() } -> Size_type<T>;
        { t.extent( Size_type<T>() ) } -> Size_type<T>;
        { t.stride( Size_type<T>() ) } -> Size_type<T>;
        { t.lbound( Size_type<T>() ) } -> Index_type<T>;
      };
  }
```

where storage ranking $SO[]$ can be defined by the concept of `Dense_storage_major`:

```
template <typename T>
  concept bool Dense_storage_major()
  {
    return requires (T t) {
      { t.store_dim( std::size_t() ) } -> std::size_t;
    };
  }
```

For most of cases, the base index $B[]$ is unchangeable, i.e, 0-based or 1-based array. However, to keep all possibilities, SPX also defines a concept for the base changeable descriptors, called a flexible descriptor:

```
template <typename T>
  concept bool Flex_descriptor()
  {
    return Descriptor<T>()
        && requires( Main_type<T> t ) {
          // t.rebase( rank, new_base );
          t.rebase( Size_type<T>(), Index_type<T>() );
        };
  }
```

Accordingly, `spx::dense_array` can be defined by the combination of *a linear storage* and a *descriptor*. In the implementation, `spx::dense_array` is inherited from a general dense storage for linear element access. The descriptor for dense array is handled by the storage rather than `spx::dense_array` itself, because only the storage needs to know how to read / write the mapping elements from subscribing

and slicing. The subclass `spx::dense_array` itself remains only an interface adapter.

As a consequence, any class is `Describable` if it defines a member type of descriptor and has an interface to access the instance of its descriptor:

```
template <typename T>
  using Descritor_type = typename T::descriptor_type;

template <typename T>
  concept bool Describable()
  {
    return requires( T t ) {
             requires Descriptor<Descritor_type<T>>();
             { t.descriptor() } -> Descritor_type<T>;
           };
  }
```

3.3.4 Concepts for array storage

Besides the descriptor, the other significant part for the design of a general dense array is the linear storage. As mentioned above, the responsibility of descriptor is to convert the subscription and slicing from the hyper dimensional index space into a linear storage space. However, it does not define how the elements are actually stored and accessed in the linear space. This section will therefore explain the concepts and designs for SPX array storage.

Currently SPX array supports 3 types of linear storage: 1) *static storage*. For known extents, its shape is assigned via template arguments, so the size can be determined at compile-time and elements can be stored by conventional C-array. The advantage is that many loop-based calculations can be done at compile time by recursive unpacking of template arguments. 2) *dynamic storage*. This is a normal dynamic array. The simplest implementation to manage underlying elements can be realized by `std::vector`. However, for more flexibility, any other resizable ele-

ment container can also be applied, i.e., a sparse storage. 3) *sparse storage*. It is an important special case of dynamic storage since any sparse storage is a type of dynamic storage. The only difference is that only non-zeros (non-shared) elements are actually stored in memory rather than all elements. SPX general differential operator is also designed based on the idea of sparse storage. Due to its importance, we will explain more in section 3.3.5.

Figure 3.5 shows the concepts for SPX array storage, where the grey sections are the class design for sparse storage that to be explained later. The general concept `Dense_storage` could be a static storage (`Static_dense_storage`) or a dynamic storage (`Dynamic_dense_storage`). Either one must be a ranked storage:

```
template <typename T>
  concept bool Ranked_storage()
  {
    return Range<T>()        // T supports begin() and end()
        && Describable<T>()  // T must have a dense descriptor
        && requires( T t ) {

             // T must provide data() to its random access iterator
             { *(t.data()) } -> Value_type<T>;
             requires Random_access_iterator<decltype(t.data())>();

             // T must provide the information for rank() and size()
             { T::rank() } -> Size_type<T>;
             { t.size() } -> Size_type<T>;
           };
  }
```

The difference is that `Dynamic_dense_storage` is `Resizable` (by providing an `resize(...)` interface to re-allocate space) and returns size information at run-time, i.e., `{ t.size() } -> Size_type<T>`, whereas `Static_dense_storage` can not re-allocate space and returns size information statically, i.e., `{ T::size() } -> Size_type<T>`.

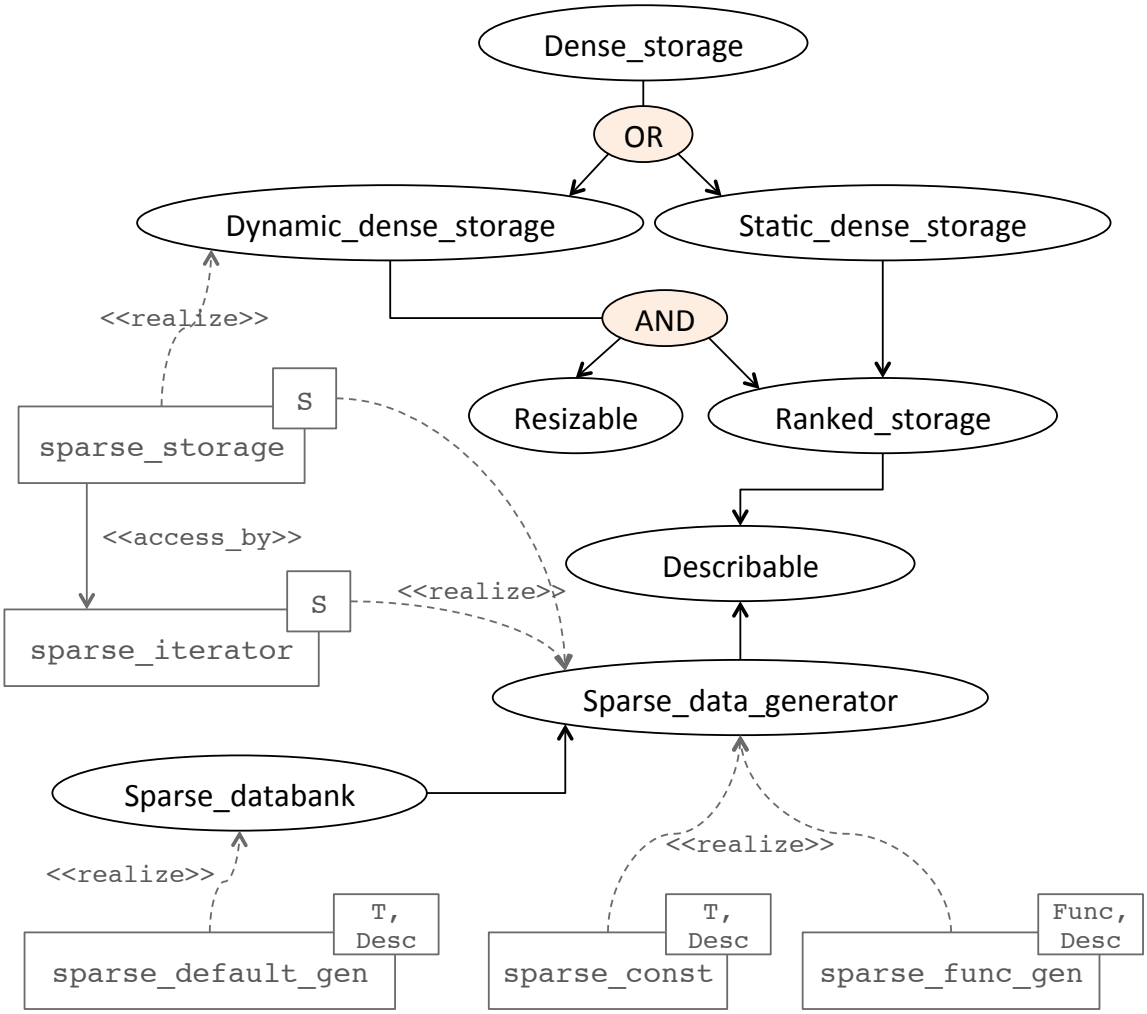


Figure 3.5: Concepts for array storage and designs of sparse array.

An obvious advantage for static storage is that the element-loop (for-each) calculation can be done by different implementations in terms of the known size of elements. For example, applying a function to each element can be done by either compile-time recursive loop-unrolling (for small number of elements), or by a regular runtime loop (for large number of elements). Thanks to the C++-Concept overloading, one of the implementation can be automatically selected at compile time based on the maximum recursive depth of C++ template, which can be assigned as

a compile option:

```
constexpr auto MAX_TEMPLATE_DEPTH = compile_options::max_template_depth();

template <Input_iterator I, typename F>
requires (static_storage<T, Desc, D...>::size() < MAX_TEMPLATE_DEPTH)
  void apply( I first, I last, F&& f )
  {
    // use compile-time recursive loop-unrolling
  }

template <Input_iterator I, typename F>
requires (static_storage<T, Desc, D...>::size() >= MAX_TEMPLATE_DEPTH)
  void apply( I first, I last, F&& f )
  {
    // use regular runtime loop
    //
    decltype(auto) iter = base_t::begin();
    for( ; first != last; ++first, ++iter )
      f( *iter, *first );
  }
```

3.3.5 Sparse array

As mentioned above, sparse storage is a special case of dynamic storage since elements in any sparse array are always dynamically allocatable. The advantage is that the elements are not required to be allocated in advance, but it still can behave like a dense array. The array can be therefore declared in a very large shape with most of empty (zero) elements. The design of sparse storage in SPX is shown as the gray part in Figure 3.5. Instead of "a real chunk of memory", in SPX the sparse storage is designed in a more general role. It is like an "element generator", for example, a callback to access elements. The callback mechanism can be also used for "element-sharing" storage in which only a few instances are initiated and shared by the others, i.e., a stencil array that most of domain nodes only requires a shared

instance (see section 3.4 for details). This is why the design idea is important since the idea will be borrowed and applied to the design of general differential operators, which is the core innovative part in this research.

```
template <typename T>
  concept bool Sparse_data_generator()
  {
    return Describable<T>()
        && requires( T t ) {
             { t.get( std::ptrdiff_t() ) } -> Value_type<T>;
           };
  }

template <typename T>
  concept bool Sparse_databank()
  {
    return Sparse_data_generator<T>()
        && requires( T t ) {
             { t.insert_on() };   // begin insertion of new elements
             { t.insert_off() };  // end insertion of new elements
             { t.elems() };       // access underlying real elements
           };
  }
```

Accordingly, `Sparse_data_generator` is the concept that defines the requirements of data generator. Any generator just provides with a simple callback interface `get( ptrdiff_t )` to return the corresponding element by giving the memory offset in linear space. `Sparse_databank` is therefore a special case for a traditional sparse array that deals with real memory chunks in which the interfaces for fetching and inserting real elements are provided.

Shown as the gray sections in Fig (3.5), `sparse_storage<S>` itself behaves as a regular dense array by realizing `Dynamic_dense_storage`, but it forwards the responsibility of element access to a sparse data generator. Its data iterator `sparse_iterator<S>` turns out to be a simple placeholder without real instance, i.e., a

plain integer `std::ptrdiff_t`. It will call back the real storage by invoking `get( ptrdiff_t )` if the element access is actually performed. SPX provides several sparse storages:

1. `spx::sparse_default_gen`: the traditional sparse array that realizes `Sparse_databank`. Owing to an arbitrary higher rank, instead of traditional two-dimensional sparse storage such as compressed row storage (CSR), the elements are stored as a hash map `std::unordered_map<std::ptrdiff_t, T>` [7].

2. `spx::sparse_const`: designed for constant array. Only one element is initiated and kept. Therefore, any element access requested via `get( ptrdiff_t )` will return the same instance.

3. `spx::sparse_func_gen`: designed as an adapter class with an external function in which all element access via interface `get( ptrdiff_t )` will be forwarded to the external function. This design will be very useful to create a "callback array" and provides with the maximum flexibility in that all element access will be lazily evaluated, determined at runtime, and all defined by user specific implementation. The callback mechanism is used in the design for general stencil operator in that any stencil requested at an index is lazily evaluated at runtime in terms of the differential expressions.

3.3.6 Expression template

Expression template (ET) [88] is a commonly used technique to customize the arithmetic expression by using C++ operator overloading. For example, in linear algebra given a matrix M and a vector v, without ET, the default calculation of the expression `M * M * v` is `(M * M) * v`, which involves expensive matrix-matrix

multiplication. By using ET, Eigen [1] can catch the expression and evaluate the results by `M * (M * v)`, which involves matrix-vector multiplication only [34, 35].

In contrast to linear algebra, SPX array is designed for a multi-dimensional container. The ET used in SPX is more similar to Blitz++. Any calculation between arrays is element-wised. Our idea is similar to ET in Blitz++, but to improve the design by using C++1y features and Concepts. An expression is designed as a random access iterator that can sequentially traverse all elements or randomly subscribe a single element. Accordingly, the concept `Expression_iterator` is defined by

```
template <typename T>
  concept bool Expression_iterator()
  {
    return Random_access_iterator<T>()
        && requires( T t ) {
             { t.suggest_stride( std::size_t() ) } -> std::size_t;
             { t.is_stride( std::size_t(), std::size_t() ) } -> bool;
             { t.can_collapse( std::size_t(), std::size_t() ) } -> bool;
             { t.advance_data( std::ptrdiff_t() ) };
             { t.advance_data() };
             { t.advance_stride( std::size_t(), std::ptrdiff_t() ) };
             { t.advance_stride( std::size_t() ) };
             requires Expression_rank<decltype(T::rank())>();
             { t.extent( std::size_t() ) } -> decltype(T::rank());
           };
  }
```

where `can_collapse( i, j )` is to detect whether elements along contiguous ranks $SO[i]$ and rank $SO[j]$ are distributed continuously, i.e., sharing the same stride. If so, then the element traversing can be expedited by looping them as the same dimension. The optimistic case is to traverse the newly created array—all elements are contiguous and can be traversed linearly without stride jumping across ranks. According to the ideas, the concept `Expressible` can be defined as

```
template <typename T>
  concept bool Expressible()
  {
    return requires( T t ) {
            requires Expression_iterator<decltype( as_expr(t) )>();
          };
  }
```

That is, for any type T substituted into a generic function `as_expr()`, this function can promote it to a corresponding expression iterator. SPX provides 3 types of expression:

1. For any type of concrete array, `as_expr()` will turn it to `expr_dense`. It will act like a normal array iterator.

2. For any type of constant applicable to each element in array, `as_expr()` will turn it to `expr_const`, which is a wrapper iterator that always traverses at the same element for whatever iterator operations.

3. For any function operations, including operator overloading, `as_expr()` will turn it to `expr_func`. This is an important design for lazy evaluation. For long expression involving a couple of large arrays such as `a * b * c *...`, without ET the binary operator `*` will be applied to all elements of two operands immediately before evaluating next `*` operator, which results in slow efficiency. `expr_func` is to wrap both the operator `*` (`spx::multiply`) and all operands into a expression, and will evaluate the value later by traversing each element and applying operator element-wisely, i.e., $a[0] * b[0] * c[0] * ...$ for the first element of the resulting array, and so on.

4. For the type that already conforms `Expression_iterator`, `as_expr()` will not further promote, and just simply return the instance itself.

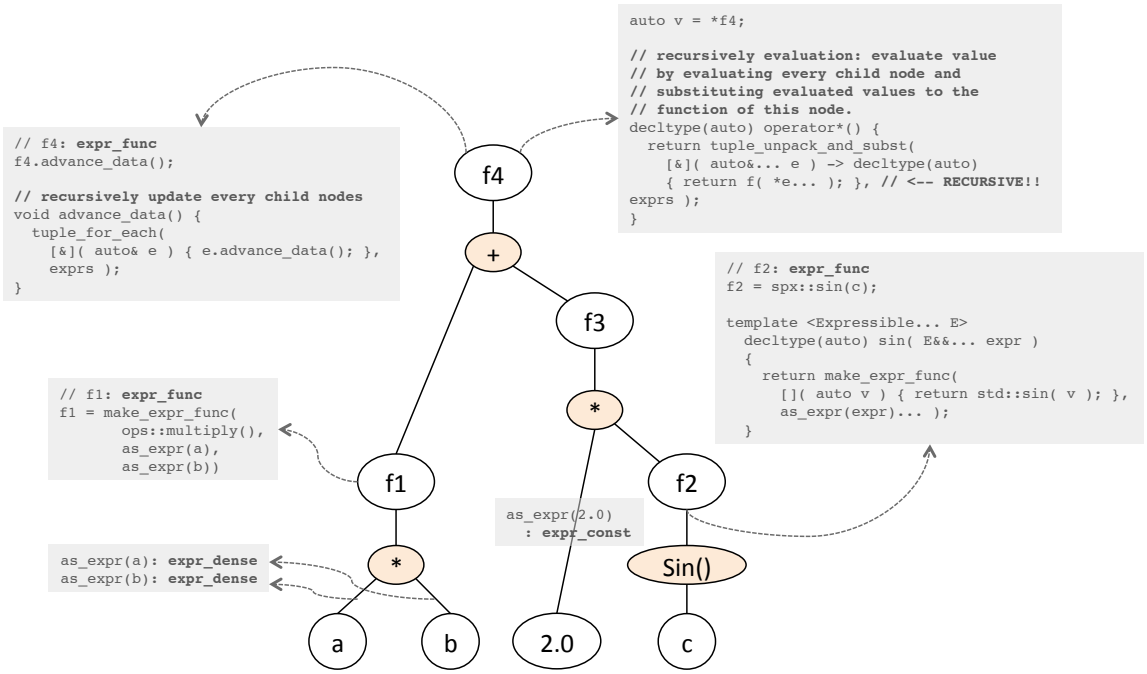


Figure 3.6: Expression tree for `a*b+2.0*sin(c)`; `a`, `b`, and `c` are the instances of `Dense_expressible`

Figure 3.6 is an example to illustrate how ET works in SPX. The expression tree will be automatically built at compile time for the expression `a*b+2.0*sin(c)`. `a`, `b`, and `c` are arrays so `as_expr` will return `expr_dense`. `2.0` is constant and will be converted into a `expr_const`, which behaves like a normal iterator and can be operated with other expression iterators, but will always refer to a single value internally. For any operations such as multiplying or numerical function `sin()`, `make_expr_func(...)` will firstly promote all Expressible operands to expression iterator by invoking `as_expr()` and then return the function expression iterator `expr_func` that wraps one operator and many operands of expressions. For binary arithmetic operators, the expressions can be created by C++ operator overloading, i.e., node `f1`, `f3`, and `f4`, while for callable functions, it can be created in the way listed as the note for node f2. Thanks to C++1y variadic template, the arbitrary

number of expression operands are saved as `std::tuple<E...>`. SPX implements useful function `tuple_for_each()` that can unpack and loop each tuple argument (expression operands) by substituting them into a function. The left note of node `f4` shows how to use the function to forward the invocation of "`advance_data()`" to every its child node, which is possibly another function expression such as `f1` or `f3` in this case. Therefore, as long as the root node `f4` for the expression `a*b+2.0*sin(c)` is moved forward, all the sub nodes in the trees will be recursively updated and moved forward as well. Similarly, the function `tuple_unpack_and_subst()` is to unpack all tuple elements and forward all unpacked elements into a function as its function arguments. It is used for the evaluation shown as the right note of node `f4`. When the iterator is invoked by de-reference, the value at current position is being evaluated by recursively evaluating every child node and substituting evaluated values to the function of this node.

As for the out class operator overloading, considering the traditional C++ design for out class binary operators, it requires many combinations for just one single operator. For example, `std::complex<T>` requires three combinations for a operator "+" overloading to do the same thing: `complex<T> + complex<T>`, `complex<T> + T`, and `T + complex<T>`. Similarly, here we have three expressible types: dense array (A), expression it self (E), and constant value type (T). Therefore, for any operator there will need 8 overloading versions doing same thing: `A-A`, `A-E`, `A-T`, `E-E`, `E-A`, `E-T`, `T-A`, and `T-E`. Thanks to the C++-Concept, instead of exhaustively listing the combinations and repeating the same implementations, the design of out-class binary operators is possible in a more general way to make all possible combinations collapse into a single interface for one operator: by introducing a concept `Binary_expressible`. For example, all possible combinations for operator "+" overloading can be included in a single interface by checking this concept:

```
template <Expressible T, Expressible U>
requires Binary_expressible<T, U, ops::plus>()
  constexpr decltype(auto) operator + ( T&& t, U&& u )
  {
    return make_expr_func( ops::plus(),
                           std::forward<T>(t), std::forward<U>(u) );
  }
```

Before designing binary expressible, we can categorize `Expressible` into two sub concepts, constant or non-constant expressible. A non-constant expressible type (`Nonconst_expressible`) is defined as a dense array (`Dense_array`) or an `Expression_iterator`, since classes for the two concepts are specifically introduced in SPX design and will be correspondingly promoted to particular expressions. Any other types, on the other hand, remain in an array constant value that will be promoted as a constant expression (`expr_const`) and regarded as an integral object operating with each of array elements. Therefore, a `Const_expressible` can be simply defined as any `Expressible` type except a `Nonconst_expressible` one:

```
template <typename T>
  concept bool Nonconst_expressible()
  {
    return Dense_array<T>() || Expression_iterator<T>();
  }

template <typename T>
  concept bool Const_expressible()
  {
    return Expressible<T>() && not Nonconst_expressible<T>();
  }
```

Accordingly, the concept of binary expressible can be very elegant by checking types `T` and `U` in three cases: 1) both are `Nonconst_expressible`, 2) `T` is `Nonconst_expressible` and its value type can be binary operated with `U` through checking the function `F`, and 3) the counterpart of case 2:

```
template <typename E, typename C, typename F>
  concept bool Binary_expressible_check()
  {
    return requires( Value_type<E> e, C c, F f ) {
            { f( e, c ) };
          };
  }

template <typename T, typename U, typename F>
  concept bool Binary_expressible()
  {
    return ( Nonconst_expressible<T>()
           && Nonconst_expressible<U>() )
        || ( Nonconst_expressible<T>()
           && Binary_expressible_check<T, U, F>() )
        || ( Nonconst_expressible<U>()
           && Binary_expressible_check<U, T, F>() );
  }
```

Here we have shown how C++-Concept can be applied in the general design of out-class binary operator overloading for expression templates to significantly reduce duplications by checking the behavior of expressible types rather than exhaustedly listing all combinations.

3.4 General design for linear stencil operator

A linear equation built on a node (called collocation node) consists of a stencil linear operator $L[\cdot]$ and its right hand side value. The stencil operator on the node $\mathbf{x}_m$ can be generally represented by

$$L[\phi(\mathbf{x}_m)] = \sum_{n \in S(\mathbf{x}_m)} w_{mn} \phi(\mathbf{x}_n) \tag{3.3}$$

where $S(\mathbf{x}_m)$ is the set of neighbor nodes of $\mathbf{x}_m$, and w_{mn} is the weight for the stencil operator at node $\mathbf{x}_m$ for its neighbor node $\mathbf{x}_n$. Considering a N-dimensional regular grid, $\mathbf{x}_m$ is at the node index $\mathbf{I}_m = [I^0, I^1, \cdots, I^N]_m \quad \forall \mathbf{I} \in \mathbb{Z}$, or denoted as $\mathbf{x}(\mathbf{I}_m)$,

so its neighbor node $\mathbf{x}_n$ can be represented as $\mathbf{x}(\mathbf{I}_n) = \mathbf{x}(\mathbf{I}_m + \Delta\mathbf{I}_n)$ in which $\Delta\mathbf{I}_n$ is node index offset from the applying node $\mathbf{I}_m$. Because of all collocation coordinates fixed on the node grids, stencil can be expressed as a set of key-value pairs in which key is $\Delta\mathbf{I}_n$ and value is its corresponding weight w_{mn}, denoted as $\mathbf{I}_n \rightarrow w_{mn}$. For example, the stencil of a 2D Laplace operator looks like Table 3.1.

$\Delta\mathbf{I}_n$	w_{mn}
$[0, 0]$	$-2\left(\frac{1}{(\Delta x^0)^2} + \frac{1}{(\Delta x^1)^2}\right)$
$[1, 0]$	$\frac{1}{(\Delta x^0)^2}$
$[-1, 0]$	$\frac{1}{(\Delta x^0)^2}$
$[0, 1]$	$\frac{1}{(\Delta x^1)^2}$
$[0, -1]$	$\frac{1}{(\Delta x^1)^2}$

Table 3.1: Illustration of stencil binary operator overloading.

Accordingly, stencil in SPX is designed like a "map container", which can be implemented by using `std::map<K, V>` or hash table `std::unordered_map<K, V>`. C++-Concept for stencil can be defined as

```
template <typename T>
  using Index_type = typename T::index_type;

template <typename T>
  using Weight_type = typename T::weight_type;

template <typename T>
  using Index_vector_type = typename T::idx_vec_t;

template <typename T>
  concept bool Stencil()
  {
    return requires( T t ) {
```

```
        { T::rank() } -> Size_type<T>;
        typename Index_type<T>();
        typename Weight_type<T>();
        requires Range_of_type<Index_vector_type<T>, Index_type<T>>();
        requires Default_constructible<Index_vector_type<T>>();
        // map-like query interface
        { t[ Index_vector_type{} ] } -> Weight_type<T>;
        // iterator key-value pairs like std::map
        requires Range<T>();
      };
}
```

Since stencil operator is a linear operator, it implies that linear arithmetic operation "axpy" is applicable to stencils. For example, any stencil operator can be multiplied with a constant or a scalar field not depending on unknowns. Also, given two linear terms, adding or subtracting a linear term to the other one, or applying a linear term to the other one such as $\frac{\partial}{\partial x^0}\left(\frac{\partial}{\partial x^1}\right)$, all result in another linear term. By whichever linear operations applied, the resultant stencil is called "composite stencil", still representing a linear term, acting like a normal stencil, and can be linearly joined by another linear term. It can be formulated like below in which a can be a constant or a known scalar field.

$$\texttt{linear\_term}(\mathbf{x}_m) = \begin{cases} \text{normal } \texttt{stencil} \text{ (leaf) at } \mathbf{x}_m \\ \texttt{composite\_stencil}(\mathbf{x}_m) \\ a(\mathbf{x}_m) \times \texttt{linear\_term}(\mathbf{x}_m) \end{cases}$$

$$\texttt{composite\_stencil}(\mathbf{x}_m) = \begin{cases} \texttt{linear\_term\_1}(\mathbf{x}_m) \pm \texttt{linear\_term\_2}(\mathbf{x}_m) \\ \texttt{linear\_term\_1}[\texttt{linear\_term\_2}(\mathbf{x}_m)] \end{cases}$$

As stencil is designed like a map container, these operations can be done by directly manipulating on the entries of key-value pairs. For example, the multipli-

cation of constant coefficient can be trivially implemented by multiplying all values (weights) within the stencil by this coefficient, illustrating as Figure 3.7.

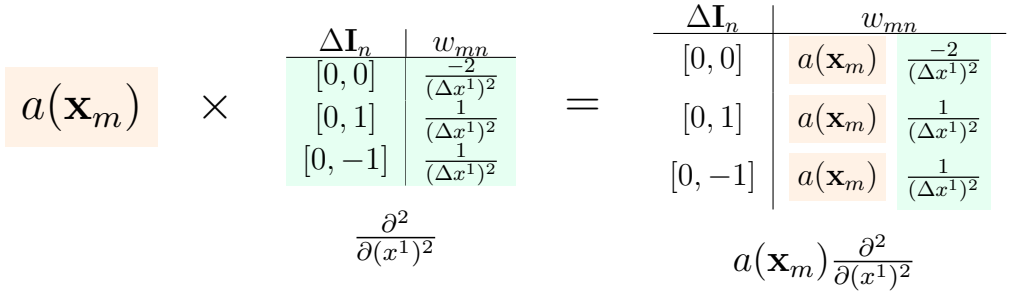


Figure 3.7: Illustration of stencil operator overloading for multiplying constant coefficient.

For adding two linear terms, we can merge two stencils by set union with the operation of plus or minus. Figure 3.8 demonstrates the example how SPX can automatically generate a 2D Laplace stencil out of adding two unidirectional differential stencil operators, which could be automatically generated by the differential basis (see section 3.5.1 for details).

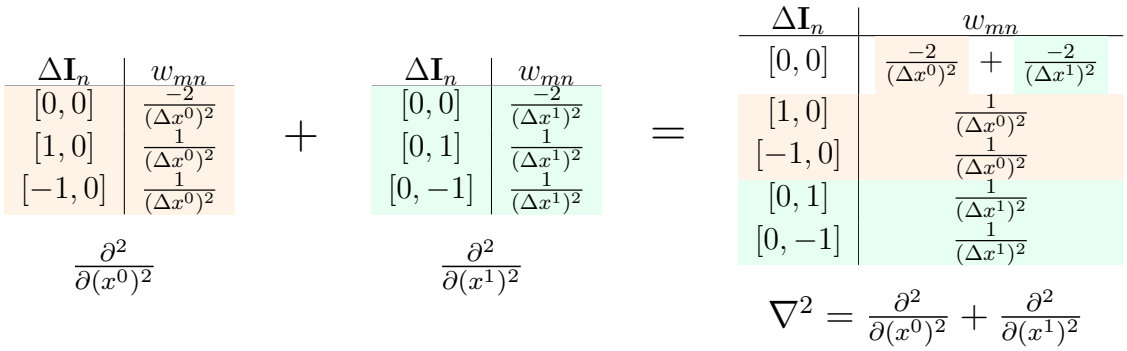


Figure 3.8: Illustration of stencil operator overloading for binary plus.

Therefore, we may have a basic design of `spx::stencil` class. The implementation of the stencil operations can be done by C++ operator overloading, listing as below. The out-class binary operator overloading can be implemented accordingly. Moreover, unary operation such as $-\frac{\partial}{\partial x^0}$ is also supported, implemented by unary minus operator overloading that forwards it to the binary multiplication of -1 and $\frac{\partial}{\partial x^0}$.

```
template <typename T, std::size_t D, typename X = std::ptrdiff_t>
  class stencil
  {
  private:
    std::unordered_map<static_vector<X, D>, T, idx_hash> data;

  public:
    /*
      ... necessary implementations to fullfil concept Stencil()
    */

    // operator overloading for stencil & stencil
    //
    // plus another stencil
    template <typename U>
    requires Has_plus_assign<T, U>()
      stencil& operator += ( const stencil<U, D, X>& x )
      {
        for( auto kv : x )
          data[ kv.first ] += kv.second;
        return *this;
      }

    // operator overloading: minus another stencil
    template <typename U>
    requires Has_minus_assign<T, U>()
      stencil& operator -= ( const stencil<U, D, X>& x )
      {
        // similar to +=
      }

    // operator overloading for stencil & scalar
    //
```

```
  // multiply coefficient
  template <typename U>
  requires Has_multiplies_assign<T, U>()
    stencil& operator *= ( const U& c )
    {
      for( auto& kv : data )
        kv.second *= c;
      return *this;
    }

  // divide by coefficient
  template <typename U>
  requires Has_divides_assign<T, U>()
    stencil& operator /= ( const U& c )
    {
      // similar to *=
    }
};
```

In addition, a more important design is that applying a linear term to the other linear term results in another linear term. Its corresponding stencil also can be deduced automatically. Take an example such as $\frac{\partial}{\partial x^0}\left[\frac{\partial \phi}{\partial x^1}\right]$:

$$\left.\begin{aligned}\left.\frac{\partial \phi}{\partial x^0}\right|_{0,0} &= \frac{\phi_{1,0}-\phi_{0,0}}{\Delta x^0}\\ \left.\frac{\partial \phi}{\partial x^1}\right|_{0,0} &= \frac{\phi_{0,1}-\phi_{0,0}}{\Delta x^1}\end{aligned}\right\} \Rightarrow \frac{\partial}{\partial x^0}\left[\frac{\partial \phi}{\partial x^1}\right] = \frac{\frac{\phi_{1,1}-\phi_{1,0}}{\Delta x^1}-\frac{\phi_{0,1}-\phi_{0,0}}{\Delta x^1}}{\Delta x^0} = \frac{\phi_{1,1}-\phi_{1,0}-\phi_{0,1}+\phi_{0,0}}{\Delta x^0 \Delta x^1}$$

To deduce the resulting stencil, we may loop the entry of the first stencil. Each entry i in the first stencil $\mathbf{\Delta I}_m^{(i)} \rightarrow w_{mn}^{(i)}$ is therefore further applied to every entry j, by a second loop, of the second stencil and to deduce the resulting entry $\mathbf{\Delta I}_m^{(i)} + \mathbf{\Delta I}_m^{(j)} \rightarrow w_{mn}^{(i)} w_{mn}^{(j)}$. Figure 3.9 illustrates the example we used here.

This example only shows applying a stencil to another stencil. To expand to a more general application, a stencil can be applied to any composite stencil–an expression of linear equation. That is, the inner stencil is not necessary to be pre-

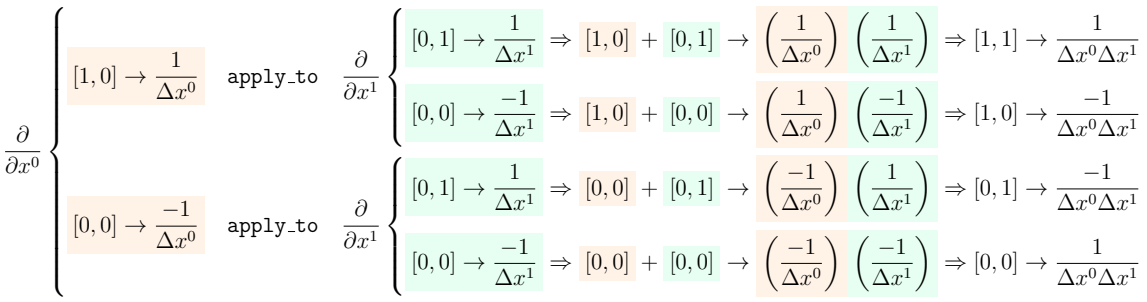


Figure 3.9: Illustration of stencil operator overloading for applying to another linear stencil.

determined. Instead, it can be *an expression with value type of stencil* that can be lazily evaluated on site at the node that the outer stencil is applying to. Therefore, the inner part is no more a fixed one, but dynamically generates the corresponding stencil at the request location. In summary, the procedure looks like: 1) give an entry i of outer stencil $\mathbf{\Delta I}_m^{(i)} \rightarrow w_{mn}^{(i)}$, 2) shift the current index by this offset to a new index: $\mathbf{I}_n = \mathbf{I}_m + \mathbf{\Delta I}_m^{(i)}$, 3) generate another (inner) stencil at $\mathbf{I}_n$: $\mathbf{\Delta I}_n \rightarrow w_{nk}$, 4) similar to last example, apply the outer stencil to inner stencil by looping each entry j of inner stencil $\mathbf{\Delta I}_m^{(i)} + \mathbf{\Delta I}_n^{(j)} \rightarrow w_{mn}^{(i)} w_{nk}^{(j)}$, and 5) loop next i and repeat steps 1 to 4. Obviously, `Expression_iterator` with its value type of `Stencil` is the best candidate to be adopted in the design since it perfectly fulfills every requirement in that every expression iterator is also a `Random_access_iterator` that can be randomly shifted to anywhere, and then stencil generation can be done by de-reference at the shifted location. Accordingly, the `operator ()` overloading can be appended into `spx::stencil` class as below.

```
template <Expression_iterator E>
requires Stencil<Value_type<E>>()
  decltype(auto) operator () ( E e ) const
```

```
{
  using S = Value_type<E>; // S = spx::stencil
  using U = Weight_type<S>;
  using R = function_result<ops::multiplies, T, U>;
  stencil<R, D, X> s;
  for( auto kv_i : *this )
  {
    // shift iterator e by the shifting index kv_i.first
    // and de-reference to return stencil at this location
    decltype(auto) s_j = *e.shift( kv_i.first );
    for( auto kv_j : s_j )
    {
      idx_vec_t idx = kv_i.first + kv_j.first;
      s[ idx ] += kv_i.second * kv_j.second;
    }
  }
  return s;
}
```

However, in addition to form and solve linear problem, stencil can be applied to a known field and evaluate the operated values. It is not required to be linear operator. For example, multiplying stencil $\frac{\partial}{\partial x^0}$ with the other stencil $\frac{\partial}{\partial x^1}$ results in a nonlinear operator $\left(\frac{\partial}{\partial x^0}\right)\left(\frac{\partial}{\partial x^1}\right)$, which is not possible to deduce its resulting stencil. It is no longer linear, but still applicable to evaluate a unknown scalar fields, i.e., by term-by-term evaluating a single value separately using inner product and then multiplying the two values together. More specifically, if the inner expression iterator is not type of `Stencil`, but its value type is "linearly combinable" with the stencil weight type, it implies the stencil is applying on a known scalar field and the resultant value can be obtained by linear combination. Therefore, we need to introduce a concept `Linearly_combinable` to check if two given types `T` and `U` are defined for `(T*U) + (T*U) + ...`

```
template <typename T, typename U>
  concept bool Linearly_combinable()
  {
```

```
    return Has_multiplies<T, U>()
        && requires( T t, U u ) {
             requires Has_plus<decltype(t * u)>();
             requires Has_plus_assign<decltype(t * u)>();
             requires Default_constructible<decltype(t * u)>();
           };
  }
```

Therefore, we can add an overload version of `operator ()` to `spx::stencil`. It comes with the same declaration with the previous version of `operator ()`. Thanks to the C++-Concept overloading, by checking the concepts for value type of the operatee expression, C++-Compiler will automatically dispatch the implementation version at compile time. If stencil is applying to an expression of another, then choose the first to generate another stencil; if it is applying to an expression of a pure scalar field, then choose the second (as below) to compute and return the operated single value.

```
template <Expression_iterator E, typename T = weight_type, typename U =
    Value_type<E>>
requires not Stencil<U>()
      && Linear_combinable<T, U>()
  decltype(auto) operator () ( E e ) const
  {
    using S = function_result<ops::multiplies, T, U>;
    S s{0};
    // shift iterator e by the shifting index kv_i.first
    // and de-reference to return value (of type U) at this location
    for( auto kv_i : *this )
      s += kv_i.second * (*e( kv_i.first ));
    return s;
  }
```

Stencil can be integrated with `spx::array`, as the type of array elements. Physically speaking, a dense array represents a regular grid, so an array of stencil represents the operational field in which stencil at each node stands for the linear operator

applying at this location. Therefore, two advantages are obvious: 1) expression template will automatically supports for the inter-operation of a stencil array and a scalar field (a constant coefficient array) or another stencil field; 2) the designs of subscription and slicing addressed in section 3.3.2 are automatically supported for stencil array. As mentioned in section 3.3.5, sparse array is the best candidate applied to the design of stencil array. Provided a sparse data generator with the value type of `Stencil`, the stencil array can generate stencils dynamically by the callback mechanism whenever element access is performed. In summary, `spx::stencil_array` can be implemented by inheriting `spx::dense_array`:

```
template <Sparse_data_generator SG>
  class stencil_array
    : public dense_array<Value_type<SG>, sparse_storage<SG>>
  {
  public:
    using base_t = dense_array<Value_type<SG>, sparse_storage<SG>>;
    using base_t::base_t;

    // case 1: operate on another stencil field
    //        --> return the expression of stencil (a new stencil field)
    //
    template <Stencil_generator T>
      decltype(auto) operator()( T&& sg )
      {
        return as_expr(*this)( as_expr( std::forward<T>(sg) ) );
      }

    // case 2: operate on a sclar field (linearly combinable)
    //        --> return the expression of operated field (a sclar field)
    //
    template <Expressible T>
    requires Linearly_combinable<Value_type<T>, typename
        Value_type<SG>::weight_type>()
      decltype(auto) operator()( const T& t )
      {
        return as_expr(*this)( as_expr(t) );
      }
```

```
    // for subscription and slicing, just forward to base class of
       spx::dense_array
    //
    template <Indexable... Args>
      decltype(auto) operator()( Args&&... args )
      {
        return base_t::operator()( std::forward<Args>(args)... );
      }
  };
```

where the concept of `Stencil_generator` can be defined as an `Expressible` with the value type of `Stencil`, i.e., a `spx::dense_array` itself or an `Expression_iterator` with value type of `Stencil`:

```
template <typename T>
  concept bool Stencil_generator()
  {
    return Expressible<T>() && Stencil<Value_type<T>();
  }
```

Similar to sparse array (see section 3.3.5 and Figure 3.5), to generate stencils, SPX provides three useful classes fulfilling `Sparse_data_generator` used as stencil data generators:

1. `spx::stencil_1d_to_nd`. For regular grid problems, the numerical schemes for differential operators are usually represented as a set of basis along a particular axis. Therefore, this class is an adapter to adopt `Basis_1d` (see section 3.5.1) from 1-dimension to N-dimension domain. For example, a forward difference scheme for $\frac{\partial}{\partial x_0}$ is defined along axis x_0, and by using `stencil_1d_to_nd` as the stencil generator, we can create a 2D or 3D stencil array to represent $\frac{\partial}{\partial x_0}$ for whole domain. As representing the differential operator directly associating with numerical scheme, this class usually turns out to be the leaf nodes of the expression tree.

2. `spx::stencil_func_gen`. Similar to `spx::sparse_func_gen`, it provides a callback function to forward `get( ptrdiff_t)` to the external user function to generate the stencil at the request index. It is very useful for two purposes: a) *shared stencil*. Similar to `spx::sparse_const` and the "Flyweight" pattern [23], if many locations share the same stencil, only one instance is needed to be created and kept and through the callback it is used for those locations, i.e., a Dirichlet operator represented by an identity stencil and used everywhere can form an identity stencil array (an identity operational field). b) *expression*. Considering an `Expression_iterator` with value type of `Stencil` representing an expression of linear composite operator, instead of explicit evaluating, it can be used as a callback stencil array to generate stencil by lazily evaluating at the request index since any expression iterator can be randomly shifted to any location. The code below lists the factory functions for the two applications:

```
//  factory method for callback stencil array
//
template <Descriptor Desc, typename F>
  decltype(auto) make_stencil_array( const Desc& desc, F&& f )
  {
    using SG = stencil_func_gen<Desc, F>;
    return stencil_array<SG>( SG( desc, std::forward<F>(f) ) );
  }

//  application 1: make_on_node
//
//    Given a descriptor, a stencil array can be formed with a shared
//    Dirichlet operator applied everywhere, named a "on_node"
//    operator, or an identity stencil ( zero offset pairing
//    with weight=1 ).
//
template <typename T, Descriptor Desc>
  decltype(auto) make_on_node( const Desc& desc )
  {
    using S = stencil<T, Desc::rank()>;
```

```
    auto f  = []( auto& desc, auto id ){ return S::on_node(); };
    using F = decltype(f);
    auto gen = stencil_func_gen<Desc, F>( desc, std::forward<F>(f) );

    return make_stencil_array( std::move(gen) );
  }

// application 2:
//
//   Given an expression iterator that is also a stencil generator
//   (value type of Stencil), the stencil array can be formed with
//   a callback function that just shifts the expression to the
//   request index, de-reference to get the stencil and returns it.
//
template <Stencil_generator SG>
requires Expression_iterator<SG>()
  decltype(auto) make_stencil_array( SG expr )
  {
    return make_stencil_array( expr.descriptor(),
                               [e = expr]( auto&& desc, auto idx )
                                 -> decltype(auto)
                               {
                                return *e( desc.index_of( idx ) );
                               } );
  }
```

3. `spx::stencil_selector`. Given multiple stencil arrays that each array represents a type of differential operator applying to the entire domain, `stencil_selector` can integrate them to form a selective stencil array in which each array only provides stencils for partial domain and the selection function is defined by users. It is extremely useful for boundary value problems. For example, if we have a stencil array representing ∇^2 operator for entire domain and also have the other one representing $\frac{\partial}{\partial x_0}$, by using `stencil_selector` we can form a stencil array to represent the boundary value problem in which the stencils for ∇^2 are selected for domain nodes, while $\frac{\partial}{\partial x_0}$ are selected for boundary nodes for Neumann boundary conditions.

3.4.1 Example

To put all together, the following example shows the complete code to represent the expression of $b(\mathbf{x})\nabla^2\left(a(\mathbf{x})\frac{\partial}{\partial x_1}\right) + \frac{\partial}{\partial x_0}\left(b(\mathbf{x})\nabla^2\right)$ at a 5-by-5 2D domain.

```
using T = double;

int main()
{
  // 5 x 5 domain for [0, 1] x [0, 1]
  //
  size_t N[2] = { 5, 5 };
  auto x0 = linspace( 0, 1, N[0] );
  auto x1 = linspace( 0, 1, N[1] );

  // 1D basis of 1st order difference
  //
  auto b0 = make_fd_basis( x0, 1 ); // along x0
  auto b1 = make_fd_basis( x1, 1 ); // along x1

  // through stencil_1d_to_nd, the stencil array for d0() and d1()
  // can be respectively formed for the 5x5 entire domain
  //
  auto d0 = make_stencil_array<2>( N, b0, 0 ); // wrap axis-0
  auto d1 = make_stencil_array<2>( N, b1, 1 ); // wrap axis-1

  // laplace operator:
  //
  // 1) d0( d0 ) and d1( d1 ) will be dispatched via case 1 of
  //    spx::stencil_array and return expressions of stencil for
  //    d^2/d(x0)^2 and d^2/d(x1)^2, respectively.
  //
  // 2) the "+" is automatically supported by regular expression
  //    template since both sides are Expression_iterator
  //
  // 3) through application 2 of stencil_func_gen, a stencil array for
  //    laplace operator can be established through a callback to
  //    the expression
  //
  auto lap = make_stencil_array<2>( d0( d0 ) + d1( d1 ) );

  // prepare coefficients (dynamic arrays)
```

```
  //
  d_arr<T, 2> a( N );
  a = {1, 2, 3, 4, 5,
       //... //,
       21, 22, 23, 24, 25};

  d_arr<T, 2> b( N );
  b = {1, 1, 1, 1, 1,
       2, 2, 2, 2, 2,
       //... //,
       25, 25, 25, 25, 25};

  // expressions can be very flexible
  //
  auto op = make_stencil_array<2>( b * lap( a * d1 ) + d0( b * lap ) );

  return 0;
}
```

To extend to a boundary value problem, we may define `op` as the domain operator, while `d0` as the boundary operator to represent Neumann boundary conditions at top and bottom. Through `stencil_selector`, the whole problem can be defined as a selective stencil array `lhs` to represent the left hand side for the linear equation.

```
// define the left hand side of BVP
//
auto lhs = make_stencil_array(
    []( auto& desc, auto idx, auto it_dom, auto it_d0 )
    {
      if( idx[0] == desc.ubound(0) // top
       || idx[0] == desc.lbound(0) ) // bottom
      {
        return *it_d0;
      }
      else
      {
        return *it_dom;
      }
    },
    op,   // domain operator
```

```
    d0 ); // boundary operator

// applying lhs to a scalar field will be dispatched via case 2
// of spx::stencil_array to evaluate the operated results of this
// BVP (must be identical to the right hand side).
//
d_arr<T, 2> rhs = lhs( a );
```

where the last line is to apply the resulting operator to a scalar field to obtain the operated results. Since the stencil array `lhs` itself is also a `spx::dense_array`, the powerful design for subscription and slicing (section 3.3.2) can be all applied here. For example, if we query at a particular index, `lhs` will deduce and return the stencil at this node. Also we can slice `lhs` to obtain the sub stencil array:

```
// deduce and return stencil at a node
//
lhs( 2, 2 ); // domain node --> stencil from op
lhs( last(), 2 ); // boundary node at top --> stencil from d0

// a sub stencil array of lhs
//
lhs( half(), slice( 2, last() ) ); // part of the row 2 (the half row)
lhs( all(), 3 ); // the column 3
```

As the left hand side of a well defined boundary value problem, `lhs` is ready to be submitted to an implicit solver with a right hand side of scalar field (loading term) to solve the results. If a stencil is used for relaxation stationary solver, `spx::stencil` provides a set of interface to return different iterators that can traverse the specific entries, i.e., traversing key-value pairs for a line of nodes aligning the collocation node along a specified dimension (see section 3.5.3 for details). On the other hand, if a linear system is solved by an actual matrix, a stencil can form a row of the matrix, representing an equation at the collocation node. Consequently, through the design of expression template and stencil array, any expression of linear operator can be

compiled into composite stencil equation automatically.

3.5 Significant components

3.5.1 Differential basis

Any scalar field ϕ operated by a linear operator $L[\cdot]$ at collocation node i, denoted as $L[\phi(\mathbf{x}_i)]$, can be generally represented by the inner product of the weight and the basis from neighbor nodes. On structure grids, it can be seen as stencil operator, which is also generally supported by SPX mentioned in section 3.4. However, for structure grids, most of differential operators are associated to unidirectional differentiation. For example, considering a 2D example, Laplace operator at node i is actually the combination of two 1D 2nd-order differentiations along $0-$ and $1-$ directions respectively:

$$L[\phi(\mathbf{x}_i)] = \nabla^2\phi(\mathbf{x}_i) = \left.\frac{\partial^2\phi}{\partial(x^0)^2}\right|_{\mathbf{x}=\mathbf{x}_i} + \left.\frac{\partial^2\phi}{\partial(x^1)^2}\right|_{\mathbf{x}=\mathbf{x}_i} \tag{3.4}$$

The abstract class for unidirectional differential basis in SPX is therefore generally defined as a set of 1D nodes that can generate differential stencils at q-order to represent $\frac{\partial^q}{\partial(x)^q}$. Since for N-dimensional structure domain the PDE problem is solved on a rectilinear grid, which is spanned by N 1D axes. SPX allows user to choose any basis attaching on a specific axis along dimension n. As long as the example such as Eq. (3.4) is queried at node i, the basis for axis 0 and axis 1 will generate the stencils of 2nd order differentiation along axes 0 and 1, respectively, and then the stencil of Laplace operator at this node will be generated by the union of two stencils (so the stencil generated by the basis is the "leaf" of the composite tree of joint stencils). That is, a user can easily mix different differential schemes. Also, for the domain with periodic boundary condition along direction n, a user may simply

select periodic basis , i.e., Fourier basis and period finite difference basis, without any additional considerations.

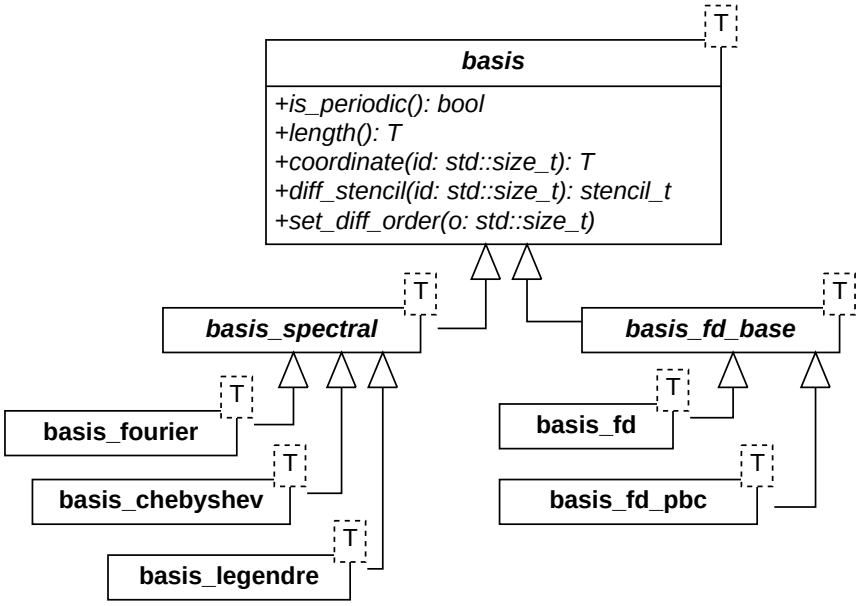


Figure 3.10: Conceptual UML class diagram for differential basis.

Figure 3.10 shows the conceptual UML class diagram of the hierarchy of differential basis classes. It is only used to demonstrate that the differential basis can be easily designed in a hierarchical structure. However, instead of object-oriented approach, in SPX the basis is designed by using C++-Concept and generic programming. Any concrete basis can be specialized according to its properties:

1. *period / non-period:* Fourier and period finite difference are periodic basis, while Chebyshev, Legendre and regular finite difference basis are non-period.
2. *fixed / flexible griding:* grids for all finite difference basis can be arbitrarily defined, i.e., non-uniform grid, while all spectral basis must be fixed griding (Fourier grids are uniformly spanned, and Chebyshev and Legendre grids are Gauss or Gauss-Lobatto quadrature points).

3. *uniform / non-uniform griding:* grids for Fourier basis must be uniform, finite difference basis can be optionally uniform, but Chebyshev and Legendre basis must be non-uniform.

4. *length:* for non-period basis the length is the distance between the first and the last node, while for period basis the length is longer than it. Period length is bounded by domain size. For example, Fourier basis with N grids actually represents the length of $N\Delta x$. SPX supports the calculation of periodic distance.

To design the concept of differential basis, some requirements are therefore listed: 1) generate differential stencils, i.e., a 1D array of `Stencil`; 2) define axial coordinates (1D array) at which those stencils are defined; 3) define periodicity; 4) define bounded basis length; and 5) user can assign the order of differentiation:

```
template <typename T>
  concept bool Basis_1d()
  {
    return Default_constructible<T>()
        && Movable<T>()
        && Copyable<T>()
        && requires( T t ) {
             requires Stencil<typename T::stencil_t>();
             { t.diff_stencil()[ std::size_t() ] } -> typename T::stencil_t;
             { t.coords()[ std::size_t() ] } -> Value_type<T>;
             { T::periodic() } -> bool;
             { t.length() } -> Value_type<T>;
             { t.set_diff_order( std::size_t() ) };
           };
  }
```

It can be observed that the C++-Concept definition is very similar to the base class in object-oriented approach. However, the basis classes in SPX are implemented in pure generic programming in which no class hierarchy is needed and the flexibility can be easily done by type alias of a general version of a generic base class. For example, the implementations for period and non-periodic finite difference are only different at the calculation of stencils, while the rest parts re-

main identical to fulfill `Basis_1d`. Therefore, in SPX the class `spx::basis_1d_fd` is the general version of finite difference basis. Its private function `update_stencil` used to update the member data of differential stencils will have two different versions. Thanks to C++-Concept overloading, by checking `periodic()` at compile time only one version will be adopted correspondingly. Therefore, the classes for periodic (`spx::basis_fd_pbc<T>`) and regular (non-periodic) finite difference basis (`spx::basis_fd<T>`) are just simply defined by type alias:

```
template <typename T, pbc_type P>
  class basis_1d_fd
  {
  public:
    constexpr static bool periodic() { return P == pbc_type::periodic; }

    // ...implementations for "Basis_1d" concept

  private:
    template <bool dummy = true>
    requires basis_1d_fd<T, P>::periodic()
      void update_stencil()
      {
        // update stencils for PERIODIC finite difference basis
      }

    template <bool dummy = true>
    requires not basis_1d_fd<T, P>::periodic()
      void update_stencil();
      {
        // update stencils for NON-PERIODIC finite difference basis
      }
  };

// type alias
//
template <typename T>
  using basis_fd = basis_1d_fd<T, pbc_type::non_periodic>;

template <typename T>
  using basis_fd_pbc = basis_1d_fd<T, pbc_type::periodic>;
```

3.5.2 Geometry and grids

As shown in Figure 3.11, SPX supports both regular and curvilinear domains. Coordinates for any domain are represented by a position vector field. For N-dimensional regular domain, the position vector field is spanned by N coordinate axes, and formed as a rectilinear grid. If this regular domain is used as a computational domain, each axis along with a specific dimension might be associated with a differential basis (section 3.5.1). On the other hand, the position vector field for curvilinear domain has individual coordinate components and covariant basis for each domain node. Position vector will be defined by a standard vector field to ensure

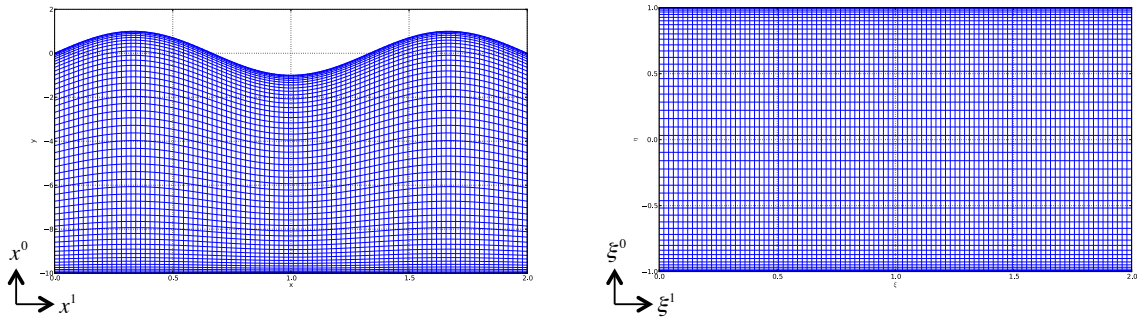


Figure 3.11: Illustration of curvilinear (left) and rectilinear (right) domain.

the vector invariance. For the example of Figure 3.11, given a node index (I_0, I_1) we have the position vector invariance:

$$\begin{aligned} \mathbf{r}(I_0, I_1) &= x^0(I_0, I_1)\mathbf{e}_0(I_0, I_1) + x^1(I_0, I_1)\mathbf{e}_1(I_0, I_1) \\ &= \xi^0(I_0, I_1)\mathbf{a}_0(I_0, I_1) + \xi^1(I_0, I_1)\mathbf{a}_1(I_0, I_1) \end{aligned} \tag{3.5}$$

Clearly, the requirement of grid class is to provide the $N \times N$ basis and its associated N coordinate components for every node. For every node, both rectilinear (`spx::rectlin_grd`) and curvilinear (`spx::curvlin_grd`) grids provide the following geometric information:

1. *coordinates*. Since curvilinear domain is formed upon an underlying rectilinear grid spanned by ξ^i axes, which is the reference domain used for the computations, invoking `coords()` can get the coordinates of $\boldsymbol{\xi}$. Note that, as a rectilinear grid, $\boldsymbol{\xi}$ need not to be uniform, while it depends on what numerical scheme is used for reference domain, i.e., nonuniform regular grid for Chebyshev basis. For rectilinear domain, on the other hand, `coords()` just returns the Cartesian coordinates in for real geometry.

2. *coordinate basis*: invoking `b_covrnt()` to get covariant basis $\mathbf{a}_j = \frac{\partial \mathbf{x}}{\partial \xi^j}$. In SPX it is stored as $\frac{\partial x^i}{\partial \xi^j}$ for each node, i.e., a 3×3 matrix for 3D domain and each column j represents $\mathbf{a}_j$. Similarly, `b_contra()` is for contravariant basis $\mathbf{a}^j = \nabla \xi^j$.

3. *identity matrix*: invoking `g_covrnt()` to get covariant metric tensor $g_{ij} = \mathbf{a}_i \cdot \mathbf{a}_j$, while invoking `g_contra()` to get contravariant metric tensor $g^{ij} = \mathbf{a}^i \cdot \mathbf{a}^j$.

4. *Christoffel symbol*: invoking `cristoffel()` to get Γ^i_{jk}.

5. *Jacobian*: invoking `jacobian()` to get Jacobian $J = \det(\mathbf{J}) = \det\left(\frac{\partial x^i}{\partial \xi^j}\right)$

Curvilinear domain requires to compute and store all geometric information for all nodes. Those information listed as above can be updated by calling `update( const A& x )`, where `x` is a dense array of Cartesian coordinates mapped in real geometry.

For rectilinear grid, on the other hand, most of nodes share the same geometry information, so SPX will not really initiate instances for every node. For example,

for Cartesian grid we only need to initiate an instance of basis that can be shared by all nodes. Thanks to the general design of sparse array (see section 3.3.5), the famous `meshgrid` function in Matlab to generate Cartesian grid can be implemented via the callback mechanism. The value of coordinate vector is only lazily determined when element access is performed, while most of elements are sparsely kept only for axial coordinates:

```
// return a dense_array with a callback sparse storage
//
template <Vector V, Vector... Args>
  decltype(auto) meshgrid( const V& x0, const Args&... x )
  {
    using T = Value_type<V>;
    constexpr auto D = 1 + sizeof...(Args);

    return make_callback_array<D>(
             // extents
             { x0.size(), x.size()... },
             // callback function to return element
             [&]( auto& desc, auto id )
             {
               using S = static_vector<T, D>;
               auto idx = desc.index_of(id);
               return varargs_trans(
                        [&]( auto&&... u ) { return S{ u... }; },
                        [&]( auto&& y, auto d ) { return y[ idx[d] ]; },
                        x0, x... );
             } );
  }
```

where `varargs_trans(...)` is an auxiliary function supported by SPX type library for the transformation of variadic arguments. It takes input arguments of `x0, x1, x2, ...`, as well as the "reduce" function `g` (the first argument) and the "transform" function `f` (the second argument), and eventually returns `g( f(x0), f(x1), f(x2), ... )`. In this example, the transform function returns the corresponding

axial coordinate in terms of request index, and then the reduce function will combine them into a vector of coordinate.

Moreover, SPX also supports dynamic curvilinear grid, named `curvilin_dyn_grid`, which can be used for moving grid problems. It is based on two assumptions: 1) time steps for real domain (t) and reference domain (τ) are identical, $t = \tau$; and 2) the coordinates $\boldsymbol{\xi}$ are stationary, not changed wit respect to τ. The design therefore can be simplified to just keeps a few time points of static grids (3×3 for 3D domain rather than 4×4) used for the calculation of temporal differentiation. For example, if 3-point finite difference is applied, given the latest time step t_c only the grids t_c, $t_c - \Delta t$, $t_c - 2\Delta t$ are kept to be used for the evaluation of $\frac{\partial}{\partial t}$ at current time t_c. By calling `advance( const A& x )` for the latest mapping coordinate, `curvilin_dyn_grid` will roll back the grids and update the spatial and temporal geometric information. Besides the information provided by the static curvilinear grid, `curvilin_dyn_grid` gives some more temporal information for moving grid:

1. *grid velocity*: invoking `grid_vel_contra()` to get contravariant components of grid velocity (along covariant basis) $\bar{w}^i = w^j \frac{\partial \xi^i}{\partial x^j}$, where the grid velocity $w^j = \frac{\partial x^j}{\partial \tau} = \frac{\partial x^j}{\partial t}$ can be obtained by `grid_vel()`.

2. *grid history*: invoking `grid_history()` to get recent history of grids kept currently.

3.5.3 Implicit solver

Implicit solver is used for boundary value problem or implicit initial value problem. Given any scalar field $\phi(\mathbf{x}, t)$, one can be generally written in

$$L[\phi(\mathbf{x}, t)] = 0 \tag{3.6}$$

where $L[\cdot]$ can be any linear or nonlinear operator. In practice, instead of being directly solved, many nonlinear problems can not be decomposed and solved in linear problems. Therefore, linear solver is even more important. Any linear problem can be written in the form

$$L\phi = b \tag{3.7}$$

The most common form of linear operator L here is, but not limited to, a matrix. L can be designed generally. For example, it could be a stencil matrix-free operator that directly evaluates the operated field directly on the grid rather than forming a matrix. Or reversely, any linear stencil operator can be transfered into a global matrix form. However, we rarely do so unless we would like to doubly confirm the solution for a small problem size. For example, evaluating a typical Laplace operator on a $N \times M$ grid only needs $N \times M$ operations of inner product of 5 numbers, but it requires solving a $(N \times M)^2$ sparse matrix if it is converted to the matrix form.

According to the numerical format of the linear operators, SPX supports 3 types of solver: 1) matrix-free Krylov methods for linear and nonlinear solvers, 2) direct solver, and 3) successive-over-relaxation (SOR) (stationary iterative) methods.

1. *Matrix-free Krylov methods (linear).* Krylov subspace methods are used to solve large linear system in the form of Eq. (3.7) by a iteration solution that only involves in matrix-vector product operation without knowing the individual element in the matrix. SPX currently supports conjugate gradient (CG), bi-conjugate gradient (biCG), and bi-conjugate gradient stable method (biCG-Stab). By this approach, the matrix operator can be designed as a general one that can operate on a scalar field, and also the vector ϕ can be in any form of scalar field. In SPX, the operator is designed as a callback that can operate on

a scalar type assigned as a generic type in the template argument, which means forming matrix is not necessary and it is based on a matrix-free workaround. Therefore, the C++-Concept for a Krylov operator is simply defined as the `Krylov_operatable`:

```
template <typename O, typename V>
  concept bool Krylov_operatable()
  {
    return requires( O op, V phi ) {
             { op( phi ) } -> V;
           };
  }
```

The UML class diagram shown as Figure 3.12 demonstrates the design for Krylov iterative solvers. The iterative solving procedures are designed in derived classes with the interface of `operator()(...)` overloading. The linear solver `iter_solver<T, S>` takes the policy class `S` for the Krylov methods, i.e., `cg`, `bicg`, or `bicgstab` (default value). The solution can be obtained by invoking `operator()(op, b, f)` with the (matrix-free) operator `op`, the right hand side of Eq. (3.7) `b`, and the (optional) constraint function `f` for adjusting the solution at each iteration, which can be optionally used for boundary conditions are applied.

2. *Matrix-free non-linear solver.* It is easily to extend Krylov solvers to a non-linear solver to solve Eq. (3.6). The tradition non-linear solver such as Newton's method for large system will inevitably involve the burden calculation of Jacobian $\frac{\partial L(\mathbf{\Phi})}{\partial \Phi}$ for $\mathbf{\Phi} = [\phi(\mathbf{x}_1), \phi(\mathbf{x}_2), \phi(\mathbf{x}_3), \cdots]$, which is not always easily to be formed. In SPX, we design and implement a simple nonlinear solver using Jacobian-free Newton-Krylov (JFNK) method [42]. By employing any linear solver, increment of correction field $\delta\Phi^k$ at iteration k can be solved, and the

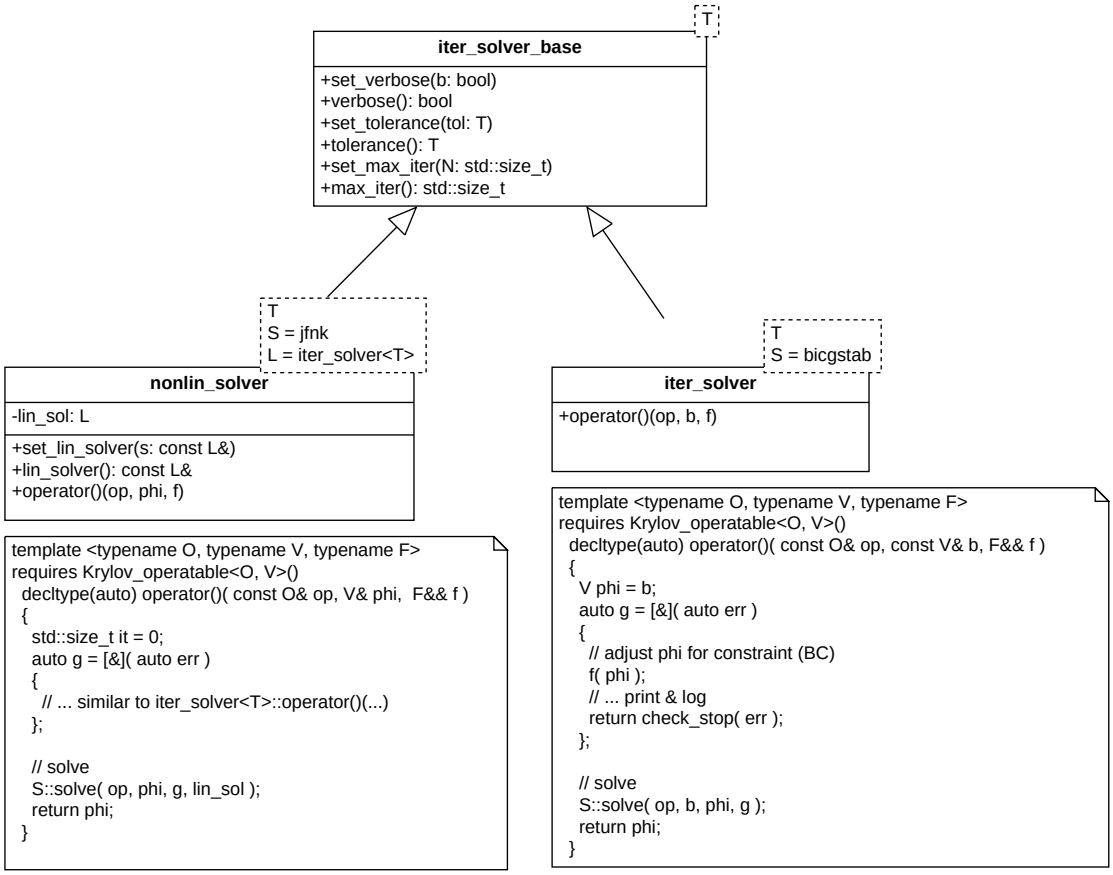


Figure 3.12: UML class diagram for Krylov iterative solver.

solution Φ^{k+1} can be updated iteratively until converged, written as

$$\mathbf{J}^k \delta \mathbf{\Phi}^k = -L(\Phi^k), \quad \Phi^{k+1} = \Phi^k + \delta\Phi^k, \quad k = 0, 1, \cdots \tag{3.8}$$

In JFNK, the matrix-free Jacobian can be simply evaluated numerically, i.e., in first order approximation

$$\mathbf{J} \approx \frac{L(\mathbf{\Phi} + \epsilon\delta\Phi) - L(\mathbf{\Phi})}{\epsilon} \tag{3.9}$$

where ϵ is a small perturbation. It can be clearly observed that for JFNK all

we need are merely a matrix-free operator $L(\Phi)$, and a embedded matrix-free linear solver.

As shown in Figure 3.12 it is designed as the derived class `nonlin_solver<T, S, L>`, where the default value of policy class `S` is `jfnk`, and takes an external linear solver `L` whose default value is, by reuse, the linear Krylov iterative solver `iter_solver`, which is used to solve the increment of correction field $\delta\Phi^k$ at each iteration k. It implies, as commonly applied nonlinear iterative solver, two nested loops are required to obtain the nonlinear solution. Similar to the linear version, in this class `operator()(op, phi, f)` takes `op` for the nonlinear operator, `phi` for the guess solution for the initial point of iteration, and `f` for the constraint function.

3. *Direct solver*. Form a concrete matrix and solve it by iterator solvers addressed above or by direct method such as Gaussian elimination. The first option is usually used for large scale problem, which can be done by wrapping the matrix-vector product as the operator used in the iterator solvers:

```
template <Matrix A, Vector V,
          typename T = Value_type<V>, typename S = bicgstab>
  decltype(auto) iter_solve( const A& a,
                             const V& b,
                             T tol = 1.e-16,
                             std::size_t max_it = 10000,
                             bool verbose = true,
                             S krv = S() )
  {
    auto op = [&]( auto& x )
    {
      return MxV( a, x );
    };
    auto f = make_iter_solver( tol, max_it, verbose, krv );
    auto x = f( op, b );
    return x;
```

```
}
```

where `MxV` is just the function to perform matrix-vector product. The second method, on the other hand, is to directly solve the linear matrix without residue error, usually used for small problem since it takes time complexity $O(N^3)$. In SPX, several options can be used to directly solve a matrix: simple LU decomposition, LAPACK LU solver, or matrix inversion. In addition, tridiagonal matrix algorithm (TDMA, or Thomas algorithm) is also implemented for solving tridiagonal matrix:

```
// directly solve ax = b

// method 1: simple LU
// b will be modified and turn out to be the solution
lu_solve( a, b );

// method 2: LAPACK LU
// b will be modified and turn out to be the solution
lapack_solve( a, b );

// method 3: matrix inversion
d_arr<T, 2> a_inv = inv( a );
d_vec<T> x = MxV( a_inv, b );

// TDMA: solve tridiagonal matrix
// b will be modified and turn out to be the solution
tdma_solve( a, b );
```

4. *Successive-over-Relaxation (SOR) (stationary iterative) methods.* This type of solver is quite different from all the solvers mentioned above. Comparing to Krylov-subspace solver that uses a general matrix-free operator, or to a standard matrix operator, a relaxation method will solve a linear problem by directly iterating on the grids. For any linear operator $L[\cdot]$ on structure grid, the

stencil operator usually exists. The design details are mentioned in section 3.4 and not repeated here. The `spx::stencil` class provides with a few different filtered iterators (provided by origin library) that allow user can iterate on specific nodes. Figure 3.13 demonstrates a 2D example for a pseudo-spectral problem in which the uniform finite difference is used as the differential basis along ξ^1, while a non-uniform Chebyshev or Legendre spectral basis is used for axis ξ^0. The detail of problem configuration procedure can be checked in section 3.4.1. According to the configuration, the colored nodes represent the stencil generated at node $x_{I,J}$ for the operator $L\left[\phi(\mathbf{x})\right] = \frac{\partial}{\partial\xi^1} + \frac{\partial}{\partial\xi^0} + \frac{\partial^2}{\partial\xi^0\partial\xi^1}$, where the term $\frac{\partial}{\partial\xi^1}$ involves in the horizontal nodes aligning $i = I$, the term $\frac{\partial}{\partial\xi^2}$ involves in the vertical nodes aligning $j = J$, and the term $\frac{\partial^2}{\partial\xi^0\partial\xi^1}$ involves in all colored nodes due to the auto-deduction design for coupled stencils (section 3.4). Note that the stencil composition may be different site by site, i.e., different operator on boundaries if a boundary value problem is being solved.

Regardless what nodes involved for which sub term, the lumped operator is all represented by the colored nodes. Different filtered iterators that provides different sweeping strategies for each entry of a stencil. For example, given a stencil `s` at node $x_{I,J}$, `s.range_line(0)` returns an iterator for the key $(\Delta x_{i,j})$-value pairs along $i = I$ (orange and blue nodes). That is, a "1D" line iterator. The other cases are similar:

- `s.range_line(0)` $\rightarrow$ iterator of nodes for key=$\Delta x_{i,j}$, $\quad i = I$
- `s.range_line(1)` $\rightarrow$ iterator of nodes for key=$\Delta x_{i,j}$, $\quad j = J$
- `s.range_line_else(0)` $\rightarrow$ iterator of nodes for key=$\Delta x_{i,j}$, $\quad i \neq I$
- `s.range_line_else(1)` $\rightarrow$ iterator of nodes for key=$\Delta x_{i,j}$, $\quad j \neq J$
- `s.range_off()` $\rightarrow$ iterator of all nodes but self, key=$\Delta x_{i,j} \neq \Delta x_{0,0}$
- `s.range()` $\rightarrow$ iterator of all nodes (normal iterator)
- `s.self()` $\rightarrow$ weight of self node, key=$\Delta x_{0,0}$

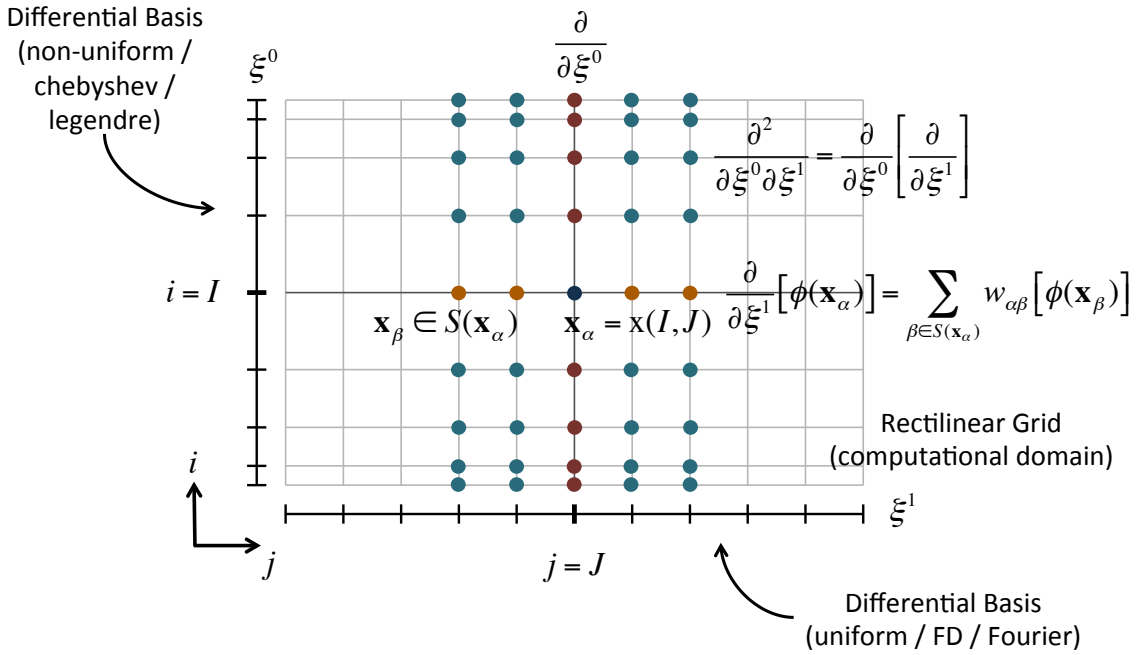


Figure 3.13: Illustration of a pseudo-spectral stencil of $L[\phi(\mathbf{x})] = \frac{\partial}{\partial \xi^1} + \frac{\partial}{\partial \xi^0} + \frac{\partial^2}{\partial \xi^0 \partial \xi^1}$ distinguished by different iterators.

based on these iterators, many SOR methods can be implemented generally. Some popular schemes supported by SPX are Jacobi iteration method, point Gauss-Seidel iteration method (PGS), line Gauss-Seidel iteration method (LGS), point successive over-relaxation method (PSOR), line successive over-relaxation method (LSOR), and alternating direction implicit method (ADI).

Generally all SOR methods take a `Stencil_array` and a `Dense_array` as inputs, respectively standing for the field of linear operator and the right hand side scalar field, as well as take the relaxation factor for PSOR, LSOR, and ADI methods. The following example code shows how PGS method can be solved by using `s.self()` and `s.range_off()`:

```
template <Stencil_array S, Dense_array A, typename T = Value_type<A>>
  void point_gauss_seidel( const S& stnarr, const A& rhs, A& u )
```

```
{
  // node index type (such as std::size_t[])
  using X = typename Value_type<S>::idx_vec_t;

  auto it = stnarr.dense_begin();
  auto end = stnarr.dense_end();

  // loop all nodes
  for( ; it != end; it++ )
  {
    // index at the current node
    X idx = it.index();

    // pump the stencil (auto-deduced if necessary)
    auto s = *it;

    // loop the stencil entries except "self"
    // and calculate the applied result
    T c{0};
    for( auto& kv : s.range_self_else() )
    {
      X id = idx + kv.first; // shift index
      c += kv.second * u( id ); // apply weight on neighbor node
    }

    // update solution
    u( idx ) = (1.0 / s.self()) * (rhs( idx ) - c);
  }
}
```

Note that for PGS the solution is updated on the old field so as to achieve converged faster than Jacobi method. From a matrix view, `s.range_self_else()` loops the nodes identical to the off-diagonal terms, while `s.self()` is the diagonal term, and `(rhs( idx ) - c)` is therefore the residue for the results applied by the off-diagonal terms. The other SOR methods can be implemented similarly by using the different categories of stencil entry iterators.

3.5.4 Time-integration (ODE solver)

Mathematically speaking, time integration is the solver for the first order ODE, which is needed by many transient (non-stationary) problems. SPX will support two types of ODE solver: 1) *multiple step schemes*, and 2) *fractional step schemes*, such as Runge-Kutta (RK) methods. Considering a scalar field of dependent variable $u = u(\mathbf{x}, t)$, it can be updated by implicit part and explicit part, either one or both.

$$\frac{\partial Ju}{\partial t} = f(u) + g(\mathbf{x}, t) = f_I + f_E + g \tag{3.10}$$

where $J = J(\mathbf{x}, t)$ is the coefficient for the dependent variable u, g is a loading term not changed with the dependent variable u, f_I is the implicit operator while f_E is the explicit operator. Note that the representation of f could be comprised of multiple f_I and f_E. For multiple step schemes, the scalar field is constantly updated with a time step Δt by using the solutions in previous steps $n, n-1, n-2, \cdots$. Therefore, the design idea in SPX is to support this general numerical form:

$$\tfrac{1}{\Delta t}\left(\alpha_0 (Ju)^{n+1} + \alpha_1 (Ju)^n + \alpha_2 (Ju)^{n-1} + \cdots\right) = f_I + f_E + g \tag{3.11}$$

$$f_I = \beta_0 f_I^{n+1} + \beta_1 f_I^n + \beta_2 f_I^{n-1} + \cdots \tag{3.12}$$

$$f_E = \gamma_0 f_E^n + \gamma_1 f_E^{n-1} + \gamma_2 f_E^{n-2} + \cdots \tag{3.13}$$

By rearranging we have

$$\begin{aligned} \left(\frac{\alpha_0}{\Delta t} J^{n+1}\right) u^{n+1} - \beta_0 f_I^{n+1} &= \left(\beta_1 f_I^n + \beta_2 f_I^{n-1} + \cdots\right) \\ &+ \left(\gamma_0 f_E^n + \gamma_1 f_E^{n-1} + \gamma_2 f_E^{n-2} + \cdots\right) \\ &- \left(\frac{\alpha_1}{\Delta t}(Ju)^n + \frac{\alpha_2}{\Delta t}(Ju)^{n-1} + ...\right) + g \end{aligned} \tag{3.14}$$

where $\alpha_n, \beta_n, \gamma_n$ are the coefficients depending on selected numerical schemes. We may write the C++-Concept for multi-step schemes accordingly:

```
template <typename T>
  concept bool Multistep_scheme()
  {
    return requires( T t ) {
            // implicit scheme?
            { T::implicit } -> bool;
            // coefficient of transient term
            requires Static_vector<decltype(t.coef_u())>();
            // number of transient term
            requires (decltype(t.coef_u())::size() == T::num_u);
            // coefficient of f term
            requires Static_vector<decltype(t.coef_f())>();
            // number of f term
            requires (decltype(t.coef_f())::size() == T::num_f);
          };
  }
```

Accordingly, the C++ Concepts for explicit and implicit multiple step schemes are easily defined with respect to `T::implicit`:

```
template <typename T>
  concept bool Implicit_multistep_scheme()
  {
    return Multistep_scheme<T>()
        && requires( T t ) {
             requires (T::implicit);
           };
  }

template <typename T>
  concept bool Explicit_multistep_scheme()
  {
    return Multistep_scheme<T>()
        && not Implicit_multistep_scheme<T>();
  }
```

The multi-step schemes supported by SPX include: 1) Leap-frog explicit method; 2) Adams-Bashforth (AB) explicit methods: forward Euler (AB1), AB2, AB3, and AB4; 3) Adams-Moulton (AM) implicit ethods: backward Euler (AM1), Crank-Nicolson (CN) or θ-method (AM2), AM3, and AM4; and 4) Backward-Dierence Formulas (BDF, implicit): BDF2, BDF3, and BDF4. Users can easily assign any scheme by substituting them as the class template arguments constrained by the `Multistep_scheme` concept.

The design of multi-step solver in SPX is pretty general. From Eq. (3.11) to Eq. (3.14) it is found that it is not necessary to just have operators f_I and f_E on the right hand side. Instead, it is possible to embed arbitrary number of operators and choose the corresponding multi-step scheme for each term, implicit or explicit, i.e., picking up AM2 and AM3 for two implicit operators respectively, as well as AB2 and AB3 for two other explicit operators. In addition, the loading term g also could be many. To achieve this goal, firstly the class `spx::multi_diff_term` is introduced to represent a single operator term:

```
// The implementation class for differential term: explicit or implicit
//
template <Dense_array U, Multistep_scheme T, Diff_operator<U> O>
  class multi_diff_term;
```

where type `U` is a `Dense_array` to represent the dependent variable u, and type O represents the operator (f_I or f_E) that fulfills the concept of differential field operator `Diff_operator`, defined in a simple behavior that can operate on a dense array to result in an operated scalar field:

```
template <typename O, typename U>
  concept bool Diff_operator()
  {
    return Dense_array<U>()
```

```
      && requires( O op, U u ) {
           { op( u ) } -> U;
         };
}
```

Two main classes are introduced subsequently: `spx::multi_stepper_transcoef` and its subclass `spx::multi_stepper`. The first one is the most general class to solve Eq. (3.10) and the second one is the specialized subclass without the consideration of the transient coefficient J, i.e., $J = 1$.

```
// main class - WITH transient coefficient
//
template <typename V, Multistep_scheme T, Dense_array U,
          typename J, typename... Args>
requires Not_empty<Args...>()
  class multi_stepper_transcoef : public multi_stepper_term<U, Args...>

// main class - WITHOUT transient coefficient
//
template <typename V, Multistep_scheme T, Dense_array U,
          typename... Args>
requires Not_empty<Args...>()
  class multi_stepper : public multi_stepper_transcoef<V, T, U, V, Args...>
```

where `V` is the value type, `T` is the multi-step scheme for transient term, `U` is the dense array of dependent variable u, and `J` is the type of transient coefficient J. The main class will deal with transient term by itself, since we always only have one transient term, while forward the operators terms and loading terms (type of `Args...`) to the base class `spx::multi_stepper_term<U, Args...>`. The design of `spx::multi_stepper_term` is a recursive inherited hierarchy in which each right hand side term, an operator or a loading term, will map to a base class of `spx::multi_stepper_term`:

```
// base class
```

```
//
template <typename... Args>
  class multi_stepper_term;

// operator term
//
template <Dense_array U, Multistep_scheme T, Diff_operator<U> O>
  class multi_stepper_term<U, T, O>;

// loading term
//
template <Dense_array U, Dense_array S>
  class multi_stepper_term<U, S>;

// recursive class - operator term
//
template <Dense_array U, Multistep_scheme T, Diff_operator<U> O,
    typename... Args>
requires Not_empty<Args...>()
  class multi_stepper_term<U, T, O, Args...>
    : public multi_stepper_term<U, Args...>;

// recursive class - loading term
//
template <Dense_array U, Dense_array S, typename... Args>
requires Not_empty<Args...>()
  class multi_stepper_term<U, S, Args...>
    : public multi_stepper_term<U, Args...>;
```

This technique is very common in generic or metaprogramming. The hierarchy of class inheritance depends on the expansion of type list `Args...` If C++ compiler encounters a pair of `Multistep_scheme` and `Diff_operator<U>`, it implies an operator term with its corresponding multi-step scheme, while if encounters just a `Dense_array`, it implies a loading term (source term). The responsibility of `spx::multi_stepper_term` is to calculate the right hand side of Eq. (3.14), and the calculation of operator term will be carried out by forwarding to the class `spx::multi_diff_term` explained above.

This design serves the maximum flexibility for client code. Users can assign the terms for right hand side as many as they want, in arbitrary combination of the sequence of 1) a pair of operator and multi-scheme or 2) a loading term. For example, a convection-diffusion equation can be declared as:

```
using T = double;

// define diffusion operator as a stencil array,
// i.e., a stencil array of Laplace operator
//
auto dfus_op = make_stencil_array( ... );

// define convection operator (evaluate explicitly)
//
auto conv_op = []( auto& u )
{
  d_arr<T, 3> cnv = //... calculate convection of u
  return cnv;
};

// initial condition for u
//
d_arr<T, 3> u = initial_u();

auto eq = make_multi_stepper( dt, bdf1<T>(), u, // BDF1 for transient term
                              am2<T>(), dfus_op,  // AM2 for diffusion term
                              ab2<T>(), conv_op ); // AB2 for convection term
```

where `dfus_op` is the diffusion operator in the type of `Stencil_array` while `conv_op` is the lambda function for convection operator that evaluates convection explicity. Note that both `Stencil_array` and the C++ lambda function fulfill `Diff_operator<U>` and can be used as the operators in `spx::multi_stepper`. Users can embed the operators or loading terms as many as they want after the third term.

Note that if the problem is defined including any implicit term, we need to further employ implicit solver to solve it, addressed in section 3.5.3. Otherwise, for explicit-only problem, u^{n+1} can be updated directly without any more iteration. Therefore,

for this example, due to the implicit diffusion term, users need to solve it by an external implicit solver at each time step:

```
auto solver = [&]( auto& rhs, auto coef_a, auto& ut,
                   auto beta_0, auto&& op )
{
  d_arr<T, 3> u =
  /*
      ... use an implicit solver to solve:
          coef_a * u - beta_0 * op(u) = rhs
          at time (n+1)
  */
  return u;
};

// advance the equation and get updated solution
//
d_arr<T, 3> new_u = eq.advance( solver );
```

Comparing to Eq. (3.14), `rhs` is the lumped array at right hand side, `coef_a` is the coefficient for transient term $\left(\frac{\alpha_0}{\Delta t} J^{n+1}\right)$ where $J = 1$ in this example, `ut` is the current (un-updated) solution, `beta_0` is β_0, and `op` is the corresponding implicit operator, which is `dfus_op` in this example. It is expected that an implicit solver would be employed here to solve u^{n+1} and return. By calling `advance( solver )`, the equation class will roll back the old solutions, update and return the latest solution for u^{n+1} by invoking `solver`. If there is no implicit operator, then the solver is not required.

For fractional step method, SPX supports explicit Runge-Kutta method, which is given by

$$u^{n+1} = u^n + \Delta t \sum_{i=1}^{s} b_i k_i \tag{3.15}$$

where s is the number of stages, and

$$
\begin{aligned}
k_1 &= f(t^n, u^n), \\
k_2 &= f(t^n + c_2\Delta t, u^n + \Delta t(a_{21}k_1)), \\
k_3 &= f(t^n + c_3\Delta t, u^n + \Delta t(a_{31}k_1 + a_{32}k_2)), \\
&\vdots \\
k_s &= f(t^n + c_s\Delta t, u^n + \Delta t(a_{s1}k_1 + a_{s2}k_2 + \cdots + a_{s,s-1}k_{s-1}))
\end{aligned}
$$

Obviously, the coefficients s, a_{ij}, b_i, and c_i can be listed as a *Butcher tableau* and form a Runge-Kutta scheme. The C++-Concept can be easily defined accordingly:

```
template <typename T>
  concept bool Runge_kutta_scheme()
  {
    return requires( T t ) {
            // number of stages
            { T::N } -> std::size_t;
            // coefficients
            { t.a( std::size_t(), std::size_t() ) } -> Value_type<T>;
            { t.b( std::size_t() ) } -> Value_type<T>;
            { t.c( std::size_t() ) } -> Value_type<T>;
           };
  }
```

Although users can easily expand the new Runge-Kutta scheme by following the definition of this concept, there are already a few commonly used schemes that have been implemented and supported in SPX: 1) The first order scheme (RK1): `rk1<T>()` as known as the forward Euler method; 2) The second order methods with two stages (RK2): `rk2<T>(p)` where $p = 1/2$ is the midpoint method (default value), $p = 2/3$ is the Ralston method, and $p = 1$ is the Heun's method; 3) The third order method (RK3, `rk3<T>()`); 4) The 4th order methods (RK4): `rk4<T>()` for the conventional

RK4 method, while `rk4_38<T>()` for the 4th order method with 3/8-rule.

Considering that the ODE problem can be solved for either a single dependent variable or multiple dependent variables, as known as the ODE system, by using partial specialization of variadic templates the design of Runge-Kutta classes look like:

```
// base class
//
template <typename... Args>
  class runge_kutta;

// Single-variable
//
template <typename V, Runge_kutta_scheme T, Dense_array U>
  class runge_kutta<V, T, U>;

// Multi-variables
//
template <typename V, Runge_kutta_scheme T, Dense_array... U>
requires ( sizeof...(U) > 1 )
  class runge_kutta<V, T, U...>;
```

where `V` is the value type and `U...` are types of `Dense_array` for dependent variables. Similar to multi-step solvers, `spx::runge_kutta` provides with an interface `advance( f )` to roll back old solutions, update and return the new solution:

```
// Single-variable
//
template <typename F>
requires RK_operator<F, V, U>()
  const U& advance( F&& f )
  {
    for( std::size_t s = 1; s < T::N; ++s )
    {
      // ...

      // call back f to get right hand side for k[s]
      k[s] = f( s, c[s]*dt, u[s] );
```

```
      // ... update solution
    }
    // return new solution
  }

// Multi-variable
//
template <typename F>
requires RK_operator<F, V, U...>()
  const std::tuple<U...>& advance( F&& f );
```

where `f` is a user specified callback function to determine what to return for the operated field on the right hand side, which is defined as a "Runge-Kutta" operator `RK_operator`:

```
template <typename F, typename V, typename... U>
  concept bool RK_operator()
  {
    return All( Dense_array<U>()... )
        && requires( F f, V v, U... u ) {
             { f( std::size_t(), v, u... ) };
           };
  }
```

The first argument is s representing the current stage, the second argument is the current fractional time point that equals to $c_s \Delta t$, and the rest of arguments are the dense arrays for the current solution at stage s. If multi-variable version is employed, `std::tuple` can be used to return the right hand side for $\frac{\partial u}{\partial t}$ at time $t + c_s \Delta t$:

```
using T = double;

t = 0;  // initial time
dt = 0.1; // time step

// create 3 arrays with arbitrary types,
//  i.e., different dimmensions
d_arr<T, 2> u0( 5, 5 ); // 5 x 5 2D array
d_arr<T, 2> u1( 8, 8 ); // 8 x 8 2D array
```

```
d_arr<T, 3> u2( 6, 6, 6); // 6 x 6 x 6 3D array

// initial conditions (IC)
//
u0 = // ... IC for u0 (at t = 0)
u1 = // ... IC for u1 (at t = 0)
u2 = // ... IC for u2 (at t = 0)

// RK operator: right-hand-side evaluation function
//
// s: current stage
// cs_dt: time increment from time t to current stage
// u0_s, u1_s, u2_s: temporary solutions at current stage
//
auto f = []( std::size_t s, T cs_dt, auto& u0_s, auto& u1_s, auto& u2_s )
{
  T t_s = t + cs_dt; // fractional time at current stage
  d_arr<T, 2> u0_dt = //... evaluate d(u0)/dt at time t_s
  d_arr<T, 2> u1_dt = //... evaluate d(u1)/dt at time t_s
  d_arr<T, 3> u2_dt = //... evaluate d(u2)/dt at time t_s

  // wrap up to a tuple and return
  return std::make_tuple( u0_dt, u1_dt, u2_dt );
};

// create RK4
//
rk = make_runge_kutta( dt, rk4<T>(), u0, u1, u2 );

// advance to obtain updated solutions (in std::tuple)
auto sol_tup = rk.advance( f );

// assign back to update variables
u0 = std::get<0>( sol_tup );
u1 = std::get<1>( sol_tup );
u2 = std::get<2>( sol_tup );

// advance time
t += dt;
```

In this example, every time `advance( f )` is performed, `f` will be called four times, at `cs_dt = 0`, `cs_dt = 0.5*dt`, `cs_dt = 0.5*dt`, and `cs_dt = dt`, respectively, accompanying with different values of `u0_s`, `u1_s`, and `u2_s`, respectively standing for

u0, u0, and u2 at that time stage. The corresponding values of $\frac{\partial u_0}{\partial t}$, $\frac{\partial u_1}{\partial t}$, and $\frac{\partial u_2}{\partial t}$ at time t_s need to be evaluated here and returned by wrapping up in a std::tuple. On the other hand, if only a single variable is being solved here, i.e., only u0, then none of std::tuple needs to be used.

3.5.5 Math functions and infrastructure

Many basic math functions will be employed, whether for library code or client code. SPX provides all necessary math functions and global functions with generic types, and will be used like Matlab or Python. Currently they are mostly for FFT functions and the functions required for spectral methods, i.e., the node and weight generation functions for Chebyshev and Legendre basis.

According to section 3.3.4, some useful C++ concepts can be defined for mathematical calculations. Note that Static_matrix_NxN shows that requirement checking of C++ concepts can be applied for checking a square static matrix.

```
// vector
template <typename T>
  concept bool Vector()
  {
    return Range<T>()
        && requires( T t ) {
             { t.size() } -> Size_type<T>;
             { t[ std::ptrdiff_t() ] } -> Value_type<T>;
             { *(t.data()) } -> Value_type<T>;
             requires Random_access_iterator<decltype(t.data())>();
             requires not Ranked_storage<T>()
                  || (Ranked_storage<T>() && T::rank() == 1);
           };
  }

// matrix
template <typename T>
  concept bool Matrix()
  {
    return Dense_array<T>()
```

```
        && requires( T t ) {
             requires (T::rank() == 2);
           };
  }

// static vector
template <typename T>
  concept bool Static_vector()
  {
    return Static_dense_storage<T>()
        && Vector<T>();
  }

// static matrix
template <typename T>
  concept bool Static_matrix()
  {
    return Static_dense_storage<T>()
        && Matrix<T>();
  }

// N-by-N static matrix
template <typename T, std::size_t N>
  concept bool Static_matrix_NxN()
  {
    return Static_matrix<T>()
        && requires() {
             { type_impl::check_nxn( size_constant<N>(), Main_type<T>() ) };
           };
  }

namespace type_impl
{
  // check matrix NxN
  template <std::size_t N, typename T, typename Desc,
            std::size_t N0, std::size_t N1>
  requires (N == N0) && (N == N1)
    constexpr defined_t check_nxn( size_constant<N>,
                                   g_static_array<T, Desc, N0, N1>&& );
}
```

Some external libraries are also linked for fundamental mathematical calculations, i.e. FFTW and LAPACK. C++-Concept helps for the optimization by using overloading.

For example, similar to the example explained in section 3.3.4, given a `Static_matrix` the implementation of matrix inversion can be optimized at compile time in terms of its extents: direct inverse for small static matrix ($N < 5$) while LAPACK inverse used for large matrix or any dynamic matrix.

```
// Matrix inverse
//
// det: determination of ORIGINAL matrix x
template <Matrix A, typename T>
requires Static_matrix_NxN<A, 1>()
  decltype(auto) inv( const A& x, T& det ) { /* direct inverse */ }

// ... similar for Static_matrix_NxN<A, 2>()
// ... similar for Static_matrix_NxN<A, 3>()
// ... similar for Static_matrix_NxN<A, 4>()

// a wrapper without determination (just inverse)
template <Matrix A>
requires Static_matrix_NxN<A, 1>()
      || Static_matrix_NxN<A, 2>()
      || Static_matrix_NxN<A, 3>()
      || Static_matrix_NxN<A, 4>()
  decltype(auto) inv( const A& x )
  {
    using T = Value_type<A>;
    T det(0);
    return inv( x, det );
  }

// use LAPACK for N >= 5 static matrix or any non-static Matrix
template <Matrix A, typename... Args>
requires not Static_matrix_NxN<A, 1>()
      && not Static_matrix_NxN<A, 2>()
      && not Static_matrix_NxN<A, 3>()
      && not Static_matrix_NxN<A, 4>()
  decltype(auto) inv( const A& x, Args&&... args )
  {
    A r = x;
    lapack_inv( r, std::forward<Args>(args)... );
    return r;
  }
```

In addition, in the context of a comprehensive framework, the infrastructure is also very important. The data serialization in SPX is also supported for VTK and HDF5 formats. `spx::vtk_writer` and `spx::vtk_writer_bin` are respectively designed for ASCII and binary VTK file output. `spx::hdf_writer` and `spx::hdf_reader` are designed for `Dense_array` output and input serialization in HDF5 format. Given any writer `w`, any `Dense_array` can be written out with a name by using `w.name( "p" ) << p`. Moreover, SPX also provides with a general timer `spx::task_timer` that can be used for simple benchmark for a piece of code (but not for profiling).

3.5.6 Parallelization

Simple parallelization has been also designed and implemented in SPX. The approach follows the C++ concurrency specifications, available since C++11 standard, which implies the shared-data algorithms on a single machine with multiple processors. There are two places designed with parallelization: 1) the evaluation of expression template, and 2) SOR solvers. The idea is simple that as long as the number of available threads sported by the current machine is known, all workload is therefore equally distributed to the working threads. First of all, `std::thread::hardware_concurrency()` is invoked to get the suggested number of concurrent threads for current environment. Note that it is just a hint number, usually (but not always, depending on the C++ compiler implementation and running machine) equal to the number of processors. As long as the number is available, each working thread will execute a dispatched task, running together with the others concurrently:

```
// number of working threads currently available
std::size_t np = std::thread::hardware_concurrency();
```

```
// prepare task
//
// th_id: the thread ID of current thread running this task
auto task = [&]( std::size_t th_id )
{
  // do sub-problem with part of workload
  // according to the current th_id
};

// launch threads
std::vector<std::thread> thds;
for( std::size_t n = 0; n < np; ++n )
  thds.push_back( std::thread( task, n ) );

// join trheads
for( std::size_t n = 0; n < np; ++n )
if( thds[ n ].joinable() )
  thds[ n ].join();
```

Considering a composite expression with multiple expressions with common stride, evaluating the result of this expression can be simply done by traversing each element without dealing with dimensional stride. Therefore, in the evaluation procedure, it is easily to equally distribute the total elements to the working threads. Since `Expression_iterator` can be copied and randomly shifted to any location, each working thread executes the task by duplicating the original expression at the starting position of local workload for itself, which can be obtained in terms of `th_id`.

SOR algorithms can be easily parallelized by this approach as well. For example, LSOR method is to construct and solve 1D matrices along a specific axis in high dimensional domain. Therefore, each task can solve part of matrices assigned by equal domain subdivision, and can be executed by working threads concurrently.

3.6 Overall performance

As for the overall efficiency, since SPX is a pure C++ generic library, there is no runtime overhead due to dynamic type information. All of abstraction, flexibility

and generality are carried out at compile-time stage due to static type system of C++ templates. In addition, runtime performance can be optimized by using the techniques of generic programming such as expression template. Another example is to utilize C++-Concept overloading such as the last example in section 3.3.4 and the matrix inversion example in section 3.5.5. Thanks to concept overloading, the performance can be optimized for small static size, automatically determined by compiler, in which for-loop calculation can be unrolled and compiled in plain and inlined assembly code. It is a common technique in generic programming to prevent runtime for-loop overhead.

This research work aims at the design of SPX, particularly for the application of C++-Concepts on the large numerical framework. Therefore, the rigorous performance benchmark waits for the future evaluation. Also the enhancement of efficiency requires future developments. However, most of cases in the dissertation work can be done pretty quickly. The performance of two larger cases can be roughly given as below:

- *Stokes' wave (section 4.8.3).* Less than one day by using Mac Pro (2 x 2.4GHz Quad core Intel Xeon E5620) with 8 GB RAM.

- *Air-water DNS (section 4.7).* About two weeks after turbulence initialization by using 2 x 3.0GHz Quad core Intel Xeon E5450 with 8GB RAM.

4. PART 3—A SMALL SCALE STUDY: DETAILED WAVE DEVELOPMENT USING CFD

4.1 Literature review on the generation of wind-waves

As mentioned earlier, the underlying mechanisms governing wind wave growth are still unclear. The earliest physical explanation was proposed by Jefferey in 1925 [38, 39]. In his "sheltering" mechanism, the waves can be grown by the pressure difference between the upwind side and the lee side of wave. However, it had been proved to be insufficient in that observations show that the rate of growth is an order of magnitude larger than prediction due to the underestimation of the pressure difference. The full physical models for wind-wave generation were first proposed by Phillips [57] and Miles [48]. Both are based on the resonance mechanism for inviscid flow. Also, both theories make *a priori* assumptions for the instantaneous presence of initial conditions, irrespective of the previous wave or fluid conditions. In Phillips' theory, a stationary random distribution of turbulent pressure fluctuations is assumed as the *a priori* condition; in Miles' theory, a perfect mean air flow is assumed. Both theories then consider the effect of perturbations caused by surface waves as a basis for development. Phillips' theory is founded on the resonant forcing on the free surface due to turbulent pressure fluctuation, while Miles' is based on the interaction between wave-induced pressure fluctuation and surface waves. One result is that growth if the wave spectrum under Phillips' theory is linear with time, where the growth rate is proportional to the variance spectrum of the turbulent pressure fluctuation, and of the order of $O(\rho_{air}/\rho_{water})$. On the other hand, Miles' theory gives the exponential growth in which growth rate is proportional to the spectrum itself and of the order of $O(\rho_{air}^2/\rho_{water}^2)$. Miles' theory is generally considered the most

promising physical model, but it is still limited due to over-simplified assumptions: 1) inviscid airflow makes air turbulence not playing a role to maintain shear flow; 2) non-linear effect such as wave-mean flow interaction is neglected, which might be critical for in-phase winds and waves. The field experiments show that in Miles' theory the rate of energy transfer from winds to waves is underestimated in an order of magnitude [15].

Considering turbulent effects, mixing length model, for wind-wave generation have been proposed and investigated by [24, 61, 4, 36, 13]. The wave growth resulting from mixing length models are similar to that from quasi-laminar theory such as Miles' model. However, the direct effect of small-scale eddies and finite wave steepness on wave growth is small. Additionally, the prediction contrasts with Miles' theory when: 1) mean flow U is in the opposite direction of wave propagation with phase speed C; 2) $C > U$ if both toward same direction. For both cases, these models give considerable wave damping. In addition, a mixing length model fails for low-frequency waves since the phase speeds of these waves allow their residence time in the generation area to be less than the eddy-turnover time, which is supposed to be the fastest process in mixing length model [87]. This leads to insufficient time to allow the momentum transfer from eddies to waves. A similar problem was also founded by [9] when there is remote air turbulence passing over slowly propagating waves, resulting in a severe truncation of the mixing length in the outer layer. Belcher and Hunt also proposed a "non-separated sheltering" mechanism [9]. Similar to Jeffrey's hypothesis, the mechanism details how Reynolds stress near surface thickens the boundary layer on the lee side of waves, allowing for flow separation if wave slope is large. According to rapid distortion theory, in Belcher's model the critical layer mechanism is only relevant for very fast moving waves [14].

In short, so far there is no perfect physical model to theoretically explain the

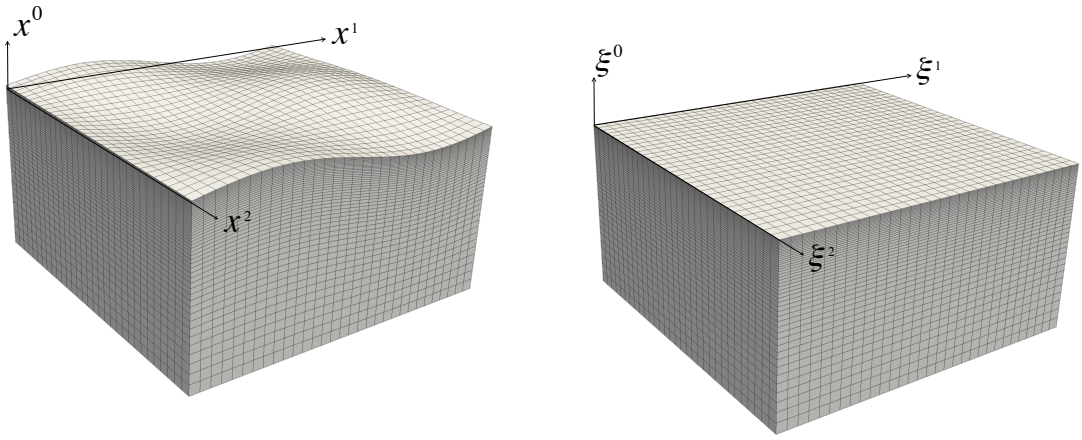
wave growth by winds. Thanks to growing computing power, it is possible to perform a numerical simulation by using direct numerical simulation (DNS) to inspect the process of wind-wave generation, *ab initio*: applying a shear wind on the top of air domain, driving air turbulence, and generating waves by the coupled interface conditions. Since there is no ensemble and averaging, and all eddies are resolved both in time and space, DNS can be seen as a means for providing small-scale information, given that the experimental measurements for wave growth by turbulent air flow can not be easily achieved. By using three-dimensional DNS, [78] simulated the turbulent air flow over an idealized wave surface, while [84] simulated the water and waves generated by specified wind stress. The first truly air-water coupled DNS simulation was proposed by [46] in which a 3D air domain and a 3D water domain are solved separately and coupled by kinematic and dynamic surface boundary conditions that considers surface tensions and the continuity of tangential stress. The work assumes linearized surface conditions and because of the space-fixing Eulerian grids, the fully nonlinear interface condition is not possible as it is only valid when it is evaluated at $z = \eta$ rather than by Taylor's expansion from $z = 0$. Accordingly, [90, 91] proposed the improved works in which curvilinear grid is employed and moving with surface waves.

In this part, we first propose a numerical approach for wave modeling using a surface-fitted moving grid, so as to evaluate fully non-linear conditions at the free surface. It is similar to [84, 90] but allows for more flexibility since it is implemented by using our well-designed SPX framework, which implies every piece of the procedure can be arbitrarily replaced according to researcher's requirements. Our approach also has several differences from previous works in more general derivations, coordinate mappings, Navier-Stokes projection methods, surface condition treatments, etc. Two main study cases are carried out subsequently. The first study case is a

DNS result for air-water coupled wave generation in which all formulations are simplified to Cartesian coordinates and solved on the fixed grid. We improve the work proposed in [46] by using nonlinear normal stress surface conditions, and make the comparisons and discussions. The second study case is to run several examples using the proposed curvilinear moving grid for water, to verify that the proposed approach is feasible.

4.2 Definition of domain and grids

A surface-fitted moving grid is employed in the numerical approach, so as to make the nonlinear properties, i.e., nonlinear surface stress, applicable directly at the free surface. To achieve the goal, a surface-fitted curvilinear grid is therefore used in which, as shown in Figure 4.1, x-domain represents the actual surface in Cartesian coordinates, while ξ-domain represents the mapping reference domain for computation, which must be a rectilinear grid.



(a) x-domain. (b) ξ-domain.

Figure 4.1: Illustration of 3D surface-fitted curvilinear and reference grids.

For horizontal axes, x^1 and x^2 are exactly coincident with or linearly scaled to ξ^1 and ξ^2, respectively. In addition, the domain along both horizontal axes are periodic. For the vertical axis, on the other hand, the definition of coordinates in Cartesian domain is shown as Figure 4.2. The surface elevation is defined as $\eta = x^0 = f(x^1, x^2, t)$, while the possible uneven bottom is defined as $h' = f(x^1, x^2, t)$. Figure 4.3 shows the mapping of vertical axis. In our approach, ξ^0 is defined as the unperturbed coordinate in the fixed range of $\xi^0 : [z_2, z_1]$, which is mapped to an arbitrary range of $x^0 : [z_2 + h', z_1 + \eta]$ in Cartesian domain.

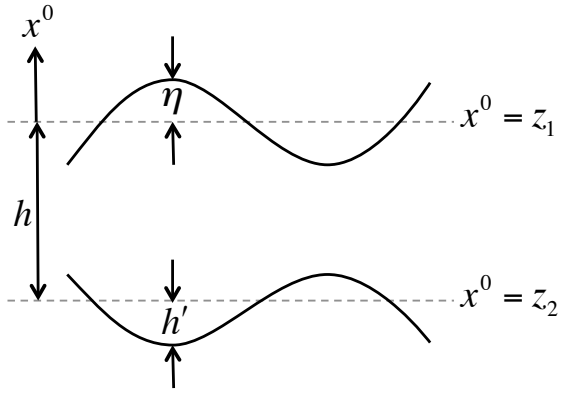


Figure 4.2: Configuration of surface coordinates in Cartesian domain.

Having the transformation,

$$S_j^i = \frac{\partial \xi^i}{\partial x^j} \tag{4.1}$$

based on the fact of this grid configurations, the important properties can be seen for simplification in many formulation transformations:

$$S_0^1 = S_0^2 = S_1^2 = S_2^1 = 0 \tag{4.2}$$

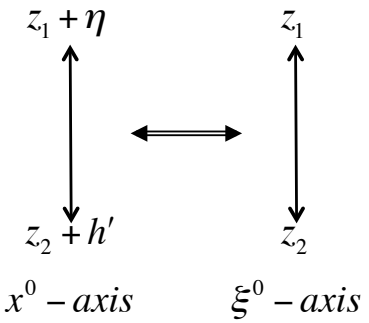


Figure 4.3: Illustration of coordinate mapping between x^0 and ξ^0.

In addition, the differential operator can be transformed from x^i to ξ^i by chain rule:

$$\frac{\partial}{\partial x^0} = S_0^0 \frac{\partial}{\partial \xi^0} \tag{4.3}$$

$$\frac{\partial}{\partial x^1} = S_1^0 \frac{\partial}{\partial \xi^0} + S_1^1 \frac{\partial}{\partial \xi^1} \tag{4.4}$$

$$\frac{\partial}{\partial x^2} = S_2^0 \frac{\partial}{\partial \xi^0} + S_2^2 \frac{\partial}{\partial \xi^2} \tag{4.5}$$

Also, based on the mapping of vertical axis, we have two significant relationships between S_j^i and η, which can be used in the derivation of surface conditions throughout:

$$S_1^0 = -\eta_{x1} S_0^0 \tag{4.6}$$

$$S_2^0 = -\eta_{x2} S_0^0 \tag{4.7}$$

where $()_{x1}$ indicates $\frac{\partial}{\partial x^1}$ and similar for $()_{x2}$. The detail of derivations in this section can be found in appendix section A.3 and section A.4.

4.3 Solving Navier-Stokes equations on curvilinear coordinates

The conservation form of Navier-Stokes equations for constant density ρ, constant viscosity ν, and incompressible Newtonian fluids reads

$$\nabla \cdot \mathbf{u} = 0 \tag{4.8}$$

$$\frac{\partial \mathbf{u}}{\partial t} = -\nabla \cdot (\mathbf{u}\mathbf{u}) - \nabla P + \nu \nabla^2 \mathbf{u} \tag{4.9}$$

where $P = p/\rho$ and p is the pressure. By using chain rule and Eq. (4.3) to (4.5), converting continuity equation from Cartesian coordinates to curvilinear coordinates results in

$$S_0^0 \frac{\partial u^0}{\partial \xi^0} + S_1^0 \frac{\partial u^1}{\partial \xi^0} + S_1^1 \frac{\partial u^1}{\partial \xi^1} + S_2^0 \frac{\partial u^2}{\partial \xi^0} + S_2^2 \frac{\partial u^2}{\partial \xi^2} = 0 \tag{4.10}$$

Similarly, the momentum equations become

$$\frac{\partial u^i}{\partial \tau} = -C[u^i] - G_i[P] + \nu \nabla^2 u^i \tag{4.11}$$

$$G_i = \frac{\partial}{\partial x^i} = S_i^j \frac{\partial}{\partial \xi^j} \tag{4.12}$$

Assuming $t = \tau$, we have

$$\frac{\partial}{\partial t} = \frac{\partial}{\partial \tau} - \bar{w}^i \frac{\partial}{\partial \xi^i} \tag{4.13}$$

where the contravariant grid velocity $\bar{w}^i = -\frac{\partial \xi^i}{\partial t}$. Obviously, $\bar{w}^1 = \bar{w}^2 = 0$ and the term is actually of the form of convection velocity, so numerically it is commonly merged with the convection term. By Eq. (4.3) to (4.5) the convection operator $C[\cdot]$

therefore becomes

$$C[u^i] = S_0^0 \frac{\partial(u^0 u^i)}{\partial \xi^0} + S_1^0 \frac{\partial(u^1 u^i)}{\partial \xi^0} + S_1^1 \frac{\partial(u^1 u^i)}{\partial \xi^1} + S_2^0 \frac{\partial(u^2 u^i)}{\partial \xi^0} + S_2^2 \frac{\partial(u^2 u^i)}{\partial \xi^2} - \bar{w}^0 \frac{\partial u^i}{\partial \xi^0} \tag{4.14}$$

For the Laplace operator, it can be decomposed into two terms as below (see appendix A.7 for derivation in detail).

$$\nabla^2 = \nabla_{\mathrm{d}}^2 + \nabla_{\mathrm{off}}^2 \tag{4.15}$$

$$\nabla_{\mathrm{d}}^2 = H_c \frac{\partial^2}{\partial \xi^0 \partial \xi^0} + g^{11} \frac{\partial^2}{\partial \xi^1 \partial \xi^1} + g^{22} \frac{\partial^2}{\partial \xi^2 \partial \xi^2} \tag{4.16}$$

$$\begin{aligned} \nabla_{\mathrm{off}}^2 = {} & (g^{00} - H_c) \frac{\partial^2}{\partial \xi^0 \partial \xi^0} + 2 g^{01} \frac{\partial^2}{\partial \xi^0 \partial \xi^1} + 2 g^{02} \frac{\partial^2}{\partial \xi^0 \partial \xi^2} \\ & + \left[S_0^0 \frac{\partial S_0^0}{\partial \xi^0} + S_1^0 \frac{\partial S_1^0}{\partial \xi^0} + S_1^1 \frac{\partial S_1^0}{\partial \xi^1} + S_2^0 \frac{\partial S_2^0}{\partial \xi^0} + S_2^2 \frac{\partial S_2^0}{\partial \xi^2} \right] \frac{\partial}{\partial \xi^0} \end{aligned} \tag{4.17}$$

where g^{ij} is the contravariant metric tensor and H_c is a constant relative to the order of g^{00}, i.e., $H_c = 1$ for the mapping illustrated in Figure 4.3.

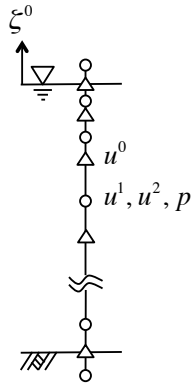


Figure 4.4: Vertical griding along ξ^0-axis. $\circ$: collocation points for u^1, u^2, and p; $\triangle$: staggered points for u^0.

Pseudospectral method is employed as the numerical scheme in which finite difference method is applied for ξ^0, while Fourier basis is applied for ξ^1 and ξ^2. Owing to the mixed scheme, a staggered grid is used for ξ^0 axis only, as shown in Figure 4.4, where u^0 is solved at staggered points while the others are solved at collocation points. Two-sided Vinokur method (`spx::two_sided_vinokur`) is used to generate grid points. By defining the ratios of the first grid spacing $\Delta\xi_0$ and the last grid spacing $\Delta\xi_{N_0}$ to the uniform grid space $\Delta\xi_u$:

$$S_0 = \Delta\xi_0/\Delta\xi_u \tag{4.18}$$

$$S_1 = \Delta\xi_{N_0}/\Delta\xi_u \tag{4.19}$$

choosing $S_0 > 1$ and $S_1 < 1$ will generate denser grid points close to the surface (compression) while coarser close to the bottom (expansion). For the quantity evaluated at the collocation points above the surface (ϕ_{surf}) or below the bottom (ϕ_{btm}), the finite difference extrapolation can be used

$$\phi_{surf} = \phi_{N_0} + \Delta\xi_{N_0} \left.\frac{\partial\phi}{\partial\xi^0}\right|_{\xi^0=z_1} \tag{4.20}$$

$$\phi_{btm} = \phi_0 - \Delta\xi_{N_0} \left.\frac{\partial\phi}{\partial\xi^0}\right|_{\xi^0=z_2} \tag{4.21}$$

where $\frac{\partial\phi}{\partial\xi^0}$ is evaluated by using higher-order finite difference (`spx::basis_1d_fd` supports any order of differentiation and any order of accuracy).

The numerical scheme also accounts for the reason for the decomposition from Eq. (4.15) to (4.17) and why an additional constant H_c is introduced here. It is because of the restriction of Fourier basis in which applying Fourier transformation to $g^{00}\frac{\partial^2}{\partial\xi^0\partial\xi^0}$ will result in a nonlinear convolution since $g^{00} = f(\xi^1, \xi^2)$, even though this term itself is a linear term.

The operator ∇^2_{d} can be therefore solved implicitly using linear stencil operator, while ∇^2_{off} can be combined with explicit terms. Accordingly, we may define an implicit operator $L_I[\cdot]$ and an explicit operator $L_E[\cdot]$, given by

$$L_I[u^i] = \nu \nabla^2_{\mathrm{d}} u^i \tag{4.22}$$

$$L_E[u^i] = \nu \nabla^2_{\mathrm{off}} u^i - C[u^i] \tag{4.23}$$

Therefore, Eq. (4.11) becomes

$$\frac{\partial u^i}{\partial t} + G_i[P] \quad = L_I[u^i] + L_E[u^i] \tag{4.24}$$

By applying numerical schemes, the general multiple step and single stage operators can be respectively represented as

$$\frac{\partial u^i}{\partial t} = \frac{1}{\Delta t}\left(\alpha_0 \left[u^i\right]^{n+1} + \alpha_1 \left[u^i\right]^{n} + \alpha_2 \left[u^i\right]^{n-1} + \alpha_3 \left[u^i\right]^{n-2} + \cdots\right) \tag{4.25}$$

$$L_I[u^i] = \beta_0 L_I[u^i]^{n+1} + \beta_1 L_I[u^i]^{n} + \beta_2 L_I[u^i]^{n-1} + \beta_3 L_I[u^i]^{n-2} + \cdots \tag{4.26}$$

$$L_E[u^i] = \gamma_0 L_E[u^i]^{n} + \gamma_1 L_E[u^i]^{n-1} + \gamma_2 L_E[u^i]^{n-2} + \gamma_3 L_E[u^i]^{n-3} + \cdots \tag{4.27}$$

where α_n, β_n, and γ_n are the coefficients depending on the selected time schemes, and n is the index of time iterator with constant time step Δt. Accordingly, Eq. (4.24) can be rewritten in the form below, similar to the general form of advection-diffusion scalar transport equation.

$$\frac{\alpha_0}{\Delta t}\left[u^i\right]^{n+1} - \beta_0 L_I[u^i]^{n+1} + G_i[P]^{n+1} = RHS \tag{4.28}$$

$$\begin{aligned} RHS = {} & \beta_1 L_I[u^i]^n + \beta_2 L_I[u^i]^{n-1} + \beta_3 L_I[u^i]^{n-2} + \cdots \\ & + \gamma_0 L_E[u^i]^n + \gamma_1 L_E[u^i]^{n-1} + \gamma_2 L_E[u^i]^{n-2} + \cdots \\ & - \frac{1}{\Delta t}\left(\alpha_1\left[u^i\right]^n + \alpha_2\left[u^i\right]^{n-1} + \cdots\right) \end{aligned}$$

where the unknown is on the left hand side while all unknowns remain on the right hand side, denoted as RHS.

Based on this formulations, Navier-Stokes equations can be solved by using fractional step method. By introducing an intermediate velocity $[u^i]^*$ and an artificial correction scalar field ϕ, according to [11], a two-stage approach named "*increment-pressure projection method*" can be written as

1. *prediction step.* Solve for intermediate velocity $\left[u^i\right]^*$

$$\frac{\alpha_0}{\Delta t}\left[u^i\right]^* - \beta_0 L_I[u^i]^* = -G_i[P]^{n-1/2} + RHS \tag{4.29}$$

2. *projection step.* Project velocity by

$$\left[u^i\right]^{n+1} = \left[u^i\right]^* - \frac{\Delta t}{\alpha_0} G_i[\phi]^{n+1} \tag{4.30}$$

To obtain ϕ^{n+1}, considering continuity equation for time $n+1$

$$\nabla \cdot \left[u^i\right]^{n+1} = 0 \tag{4.31}$$

the boundary value problem for ϕ^{n+1} is therefore defined as

$$\nabla^2\phi^{n+1} = \frac{\alpha_0}{\Delta t}\left(\nabla\cdot\left[u^i\right]^*\right) \qquad \text{on} \qquad \Omega \tag{4.32}$$

$$G_i[\phi]^{n+1} = \frac{\alpha_0}{\Delta t}\left(\left[u^i\right]^* - \left[u^i\right]^{n+1}\right) \qquad \text{on} \qquad \partial\Omega \tag{4.33}$$

In addition, the relationship between ϕ and P can be found as below, which can be used to update pressure.

$$P^{n+1/2} = P^{n-1/2} + \phi^{n+1} - \frac{\Delta t\beta_0}{\alpha_0}\left(\nu\nabla^2\phi^{n+1}\right) \tag{4.34}$$

4.4 Dynamic surface conditions

Surface and bottom boundary conditions are required for solving Eq. (4.29) and Eq. (4.32). Given impermeable air-water interface with constant surface tension, the total stress balance equations can be decomposed into 1) normal stress balance, and 2) tangential stress balance. Appendix A.1 shows the details of derivations for surface dynamics.

From the normal stress balance, the relationships of air and water for the surface pressure in Cartesian coordinate can be found as

$$-p^w + M^w + \rho^w g\eta = -p^a + M^a + \rho^a g\eta - \gamma\kappa \tag{4.35}$$

where γ is the coefficient of surface tension (force per unit length), κ is mean curvature given by

$$\kappa = \frac{-\eta_{x2x2}\left(1+\eta_{x1}^2\right) - \eta_{x1x1}\left(1+\eta_{x2}^2\right) + 2\eta_{x1}\eta_{x2}\eta_{x1x2}}{\left(1+\eta_{x1}^2+\eta_{x2}^2\right)^{\frac{3}{2}}} \tag{4.36}$$

and M^α is given by

$$M^\alpha = \frac{2\mu^\alpha}{(1+\eta_{x1}^2+\eta_{x2}^2)}\left\{\left(\frac{\partial u^0}{\partial x^0}\right) - \eta_{x1}\left(\frac{\partial u^0}{\partial x^1}+\frac{\partial u^1}{\partial x^0}\right) - \eta_{x2}\left(\frac{\partial u^0}{\partial x^2}+\frac{\partial u^2}{\partial x^0}\right) \right.$$
$$\left. +\eta_{x1}^2\left(\frac{\partial u^1}{\partial x^1}\right) + \eta_{x1}\eta_{x2}\left(\frac{\partial u^1}{\partial x^2}+\frac{\partial u^2}{\partial x^1}\right) + \eta_{x2}^2\left(\frac{\partial u^2}{\partial x^2}\right)\right\} \quad (4.37)$$

The superscript α could be a or w to indicate air or water phase. For example, if $\alpha = a$ then velocities u^i are evaluated in air phase, and similar for water. With the absence of air, the relationship can be simplified and the surface pressure for water is given by

$$p^w = \rho^w g\eta + M^w + \gamma\kappa \quad (4.38)$$

The details of derivation can be seen in appendix A.2.1. As the derivations addressed in appendix A.5, Converting M^α to curvilinear coordinates results in

$$M^\alpha = \frac{2\mu^\alpha}{1+\eta_{x1}^2+\eta_{x2}^2}\left\{-(\eta_{x2}^2+1)\left(S_1^1\frac{\partial u^1}{\partial \xi^1}\right) - (\eta_{x1}^2+1)\left(S_2^2\frac{\partial u^2}{\partial \xi^2}\right)\right.$$
$$\left. -\eta_{x1}\left(S_1^1\frac{\partial u^0}{\partial \xi^1}\right) - \eta_{x2}\left(S_2^2\frac{\partial u^0}{\partial \xi^2}\right) + \eta_{x1}\eta_{x2}\left(S_2^2\frac{\partial u^1}{\partial \xi^2}+S_1^1\frac{\partial u^2}{\partial \xi^1}\right)\right\} \quad (4.39)$$

On the other hand, according to continuity of tangential stress across the interface of air and water, given that σ_1 and σ_2 are the surface traction forces along the tangential direction x_1 and x_2, respectively, we have

$$\sigma_1^w = \sigma_1^a \quad (4.40)$$

$$\sigma_2^w = \sigma_2^a \quad (4.41)$$

With the absence of air, the tangential stress for water surface simply meets

$$\sigma_1^w = 0 \tag{4.42}$$

$$\sigma_2^w = 0 \tag{4.43}$$

The detail of the full form of σ^α and its derivations can be seen in appendix A.2.2. More importantly, the surface velocity conditions $\frac{\partial u^1}{\partial \xi^0}$ and $\frac{\partial u^2}{\partial \xi^0}$ can be respectively derived from σ_1 and σ_2. As the derivations shown in appendix A.6, the final forms read

$$\frac{\partial u^1}{\partial \xi^0} = C_0 \left\{ C_1 \frac{\partial u^0}{\partial \xi^1} + C_2 \frac{\partial u^0}{\partial \xi^2} + C_3 \frac{\partial u^1}{\partial \xi^1} + C_4 \frac{\partial u^1}{\partial \xi^2} + C_5 \frac{\partial u^2}{\partial \xi^1} + C_6 \frac{\partial u^2}{\partial \xi^2} + C_7 \frac{\sigma_{t1}^\alpha}{\mu^\alpha} + C_8 \frac{\sigma_{t2}^\alpha}{\mu^\alpha} \right\} \tag{4.44}$$

$$\frac{\partial u^2}{\partial \xi^0} = D_0 \left\{ D_1 \frac{\partial u^0}{\partial \xi^1} + D_2 \frac{\partial u^0}{\partial \xi^2} + D_3 \frac{\partial u^1}{\partial \xi^1} + D_4 \frac{\partial u^1}{\partial \xi^2} + D_5 \frac{\partial u^2}{\partial \xi^1} + D_6 \frac{\partial u^2}{\partial \xi^2} + D_7 \frac{\sigma_{t1}^\alpha}{\mu^\alpha} + D_8 \frac{\sigma_{t2}^\alpha}{\mu^\alpha} \right\} \tag{4.45}$$

where

$$C_0 = [S_0^0 (G_0)^2]^{-1} \qquad C_4 = S_2^2 A \qquad C_7 = (1 + \eta_{x2}^2) G_1$$

$$C_1 = S_1^1 (\eta_{x1}^2 - \eta_{x2}^2 - 1) \qquad C_5 = S_1^1 A \qquad C_8 = -\eta_{x1} \eta_{x2} G_2$$

$$C_2 = 2\eta_{x1} \eta_{x2} S_2^2 \qquad C_6 = S_2^2 \eta_{x1} (1 + \eta_{x1}^2 - \eta_{x2}^2) \qquad A = \eta_{x2} (1 + \eta_{x2}^2 - \eta_{x1}^2)$$

$$C_3 = S_1^1 \eta_{x1} (3 + \eta_{x1}^2 + 3\eta_{x2}^2)$$

$$D_0 = [S_0^0 (G_0)^2]^{-1} = C_0 \qquad D_4 = S_2^2 B \qquad D_7 = -\eta_{x1} \eta_{x2} G_1$$

$$D_1 = 2\eta_{x1} \eta_{x2} S_1^1 \qquad D_5 = S_1^1 B \qquad D_8 = (1 + \eta_{x1}^2) G_2$$

$$D_2 = S_2^2 (\eta_{x2}^2 - \eta_{x1}^2 - 1) \qquad D_6 = S_2^2 \eta_{x2} (3 + 3\eta_{x1}^2 + \eta_{x2}^2) \qquad B = \eta_{x1} (1 + \eta_{x1}^2 - \eta_{x2}^2)$$

$$D_3 = S_1^1 \eta_{x2} (1 + \eta_{x2}^2 - \eta_{x1}^2)$$

and

$$G_0 = (1 + \eta_{x1}^2 + \eta_{x2}^2)$$
$$G_1 = \sqrt{(1 + \eta_{x1}^2) G_0}$$
$$G_2 = \sqrt{(1 + \eta_{x2}^2) G_0}$$

As for $\frac{\partial u^0}{\partial \xi^0}$ at the surface, it can be directly given from the continuity equation Eq. (4.10), leading in

$$\frac{\partial u^0}{\partial \xi^0} = -\frac{1}{S_0^0} \left\{ S_1^0 \frac{\partial u^1}{\partial \xi^0} + S_1^1 \frac{\partial u^1}{\partial \xi^1} + S_2^0 \frac{\partial u^2}{\partial \xi^0} + S_2^2 \frac{\partial u^2}{\partial \xi^2} \right\} \tag{4.46}$$

Owing to the restrictions of Fourier transform applied to horizontal axes in which a linear differential term with non-constant coefficient will turn to be a nonlinear convolution term in frequency domain, similar to the reason accounting for the decomposition of Laplace operator, keeping only the simple forms of $\frac{\partial u^1}{\partial \xi^0}$ and $\frac{\partial u^2}{\partial \xi^0}$ (without coordinate-dependent coefficients) as the surface conditions respectively used in solving velocity u^1 and u^2 can avoid non-linear convolutions. In addition, keeping all the terms on the right hand side of Eq. (4.44) and Eq. (4.45) only with respect to $\frac{\partial}{\partial \xi^1}$ and $\frac{\partial}{\partial \xi^2}$, without finite difference calculations for vertical terms, ensures that only the spectral differentiations are involved, which improves the accuracy for evaluating surface conditions.

4.5 Kinematic surface conditions

An ordinary kinematic free surface boundary condition can be given by

$$\frac{\partial \eta}{\partial t} = f(u^i, \eta) = u^0 - u^1 \eta_{x1} - u^2 \eta_{x2} \tag{4.47}$$

Since the independent variables in Eq. (4.47) only associate with non-moving horizontal x^1 and x^2 axes and in our approach dependent variables u^i are all expressed using Cartesian coordinates, Eq. (4.47) can be solved as an independent 2D initial problem by applying Fourier basis for both axes, and need not be further converted to curvilinear coordinates.

4.6 Consolidation

First of all, η can be updated by Eq. (4.47) using current u^i. As long as η is updated, it can be further used to update the curvilinear grids by perturbing the surface grid from rectilinear initial grid. Thus, all geometry properties are accordingly updated, and the momentum equations for u^i can be solved in terms of updated curvilinear grid, so as to feed in Eq. (4.47) to update η at next time step.

Eq. (4.47) can be solved by using a second order Runge-Kutta method. As addressed in section 3.5.4, by setting $p = 1$ for `rk_2<T>` we have Heun's method for the second order Runge-Kutta written in

$$\hat{\eta}^{n+1} = \eta^n + \Delta t f([u^i]^n, \eta^n) \tag{4.48}$$

$$\eta^{n+1} = \eta^n + \frac{\Delta t}{2}\left[f([u^i]^n, \eta^n) + f([\hat{u^i}]^{n+1}, \hat{\eta}^{n+1})\right] \tag{4.49}$$

where $[\hat{u^i}]^{n+1}$ indicates u^i evaluated at time $n+1$ using $\hat{\eta}^{n+1}$. Note that the selection of Heun's method rather than the other second order schemes is to avoid u^i evaluated at fractional time step, due to u^i solved by a multiple step solver with constant Δt.

Withing each internal Runge-Kutta step, a full procedure for solving momentum equation is performed as following steps:

1. *update geometry.* Use given η to update curvilinear grid and all geometry properties.

2. *extrapolate ϕ^n to ϕ_E^{n+1}.* To obtain the boundary conditions for velocity and pressure at time $t+1$, we need a guess field of ϕ_E^{n+1} extrapolated from ϕ^n by one of the following extrapolation schemes. Numerical experiments show that $r=1$ or $r=3$ give much more stable results.

$$\phi_E^{n+1} = \begin{cases} 0 & r=0 \\ \phi^n & r=1 \\ 2\phi^n - \phi^{n-1} & r=2 \\ 3\phi^n - 3\phi^{n-1} + \phi^{n-2} & r=3 \end{cases}$$

3. *prepare Neumann boundary conditions at surface for $[u^i]^*$.* Extrapolate $[u^i]_E^{n+1}$ by using Eq. (4.30):

$$\left[u^i\right]_E^{n+1} = \left(\left[u^i\right]^*\right)^k - R_i \tag{4.50}$$

$$R_i = \frac{\Delta t}{\alpha_0} G_i[\phi_E]^{n+1} \tag{4.51}$$

where $\left([u^i]^*\right)^k$ is a guess field and k denotes the internal iteration, given by the previous internal iteration of $[u^i]^*$, which will be converged after several iterations of solving $[u^i]^*$. Eq. (4.44), Eq. (4.45), and Eq. (4.46) can be therefore evaluated as

$$\frac{\partial}{\partial \xi^0}\left[u^1\right]^* = \frac{\partial R_1}{\partial \xi^0} + f\left(\left[u^i\right]_E^{n+1}, C_0, \cdots, C_8\right) \tag{4.52}$$

$$\frac{\partial}{\partial \xi^0}\left[u^2\right]^* = \frac{\partial R_2}{\partial \xi^0} + f\left(\left[u^i\right]_E^{n+1}, D_0, \cdots, D_8\right) \tag{4.53}$$

$$\frac{\partial}{\partial \xi^0}\left[u^0\right]^* = \frac{\partial R_0}{\partial \xi^0} + f\left(\left[u^i\right]_E^{n+1}, S_j^i\right) \tag{4.54}$$

which are respectively used for the surface Neumann boundary conditions for

solving $[u^1]^*$, $[u^2]^*$, and $[u^0]^*$.

4. *prepare bottom boundary conditions for* $[u^i]^*$. According to [11], for increment-pressure project method, if prescribed velocities are imposed at the bottom, Dirichlet boundary conditions can be directly applied by the known $[u^i]^{n+1}$:

$$\left[u^i\right]^* = \left[u^i\right]^{n+1} \qquad \text{at} \qquad x^0 = -h + h' \tag{4.55}$$

On the other hand, if slip bottom is considered, i.e., emulating deep sea, the Neumann boundary conditions are imposed at the bottom as

$$\frac{\partial u^i}{\partial x^0} = S_0^0 \frac{\partial u^i}{\partial \xi^0} = \left[\mathrm{d}U^i\right]^{n+1}\Big|_{\partial\Omega} \tag{4.56}$$

where the prescribed boundary values are usually $[\mathrm{d}U^1]^{n+1}\Big|_{\partial\Omega} = [\mathrm{d}U^2]^{n+1}\Big|_{\partial\Omega} = 0$ for free slip bottom. Similar to the surface Neumann boundary conditions, we have

$$\frac{\partial}{\partial \xi^0}\left[u^i\right]^* = \frac{\partial R_i}{\partial \xi^0} + \frac{1}{S_0^0}\left[\mathrm{d}U^i\right]^{n+1} \qquad \text{at} \qquad x^0 = -h + h' \tag{4.57}$$

5. *solve* $[u^i]^*$. With the surface and bottom boundary conditions, solve Eq. (4.29) using the implicit solver supported by SPX. Considering the resultant stencil for the left hand side of Eq. (4.29) at any given node n, the stencil entries only distribute along ξ^0-axis due to the stencil of o-th order Fourier differentiation with respect to ξ^1 and ξ^2 only regarding "self" node as $\left(ik_n^1 \frac{2\pi}{L_1}\right)^o$ and $\left(ik_n^2 \frac{2\pi}{L_2}\right)^o$ respectively, where k_n^p indicates the mode number along p-axis for node n. Accordingly, LSOR is the best chose to solve this problem, since by setting sweeping direction along 0-axis, it takes only one iteration to reach converged.

6. *converge* $[u^i]^*$. Repeat step 3 to 5 to converge $[u^i]^*$, i.e., L^2-norm at the surface smaller than a tolerance:

$$\left\| \left([u^i]^* \right)^{k+1} - \left([u^i]^* \right)^k \right\|_2$$

7. *prepare Dirichlet boundary conditions at surface for* ϕ^{n+1} On the basis of converged solution of $[u^i]^*$, extrapolate $[u^i]_E^{n+1}$ again using Eq. (4.50), and evaluate pressure p^{n+1} at surface using Eq. (4.38). Hence, P at time $t+1$ is given by $P^{n+1} = p^{n+1}/\rho$, and $P^{n+1/2}$ can be obtained by quadratic interpolation:

$$P^{n+1/2} = \frac{1}{15}\left(8P^{n+1} + 10P^{n-1/2} - 3P^{n-3/2}\right) \tag{4.58}$$

According to Eq. (4.34), the Dirichlet boundary condition at the surface for ϕ^{n+1} can be given by

$$\phi^{n+1} = P^{n+1/2} - P^{n-1/2} + \frac{\Delta t \beta_0}{\alpha_0}\left(\nu \nabla^2 \phi_E^{n+1}\right) \tag{4.59}$$

8. *prepare Neumann boundary conditions at the bottom for solving* ϕ^{n+1}. According to [11], for increment-pressure projection method, the boundary conditions for solid walls is given by $\mathbf{n} \cdot \nabla \phi^{n+1}|_{\partial\Omega} = 0$. Therefore, the Neumann bottom boundary condition for Eq. (4.32) is simply given by

$$\frac{\partial \phi^{n+1}}{\partial \xi^0} = 0 \qquad \text{at} \qquad x^0 = -h + h' \tag{4.60}$$

9. *solve* ϕ^{n+1}. Solve Eq. (4.32) for ϕ^{n+1} with the surface and bottom boundary conditions resulted from step 7 and 8, as well as the loading term evaluated

based on $[u^i]^*$ resulted from step 6. Similar to solving momentum equations, the Laplace operation on the curvilinear coordinates is firstly decomposed by using Eq. (4.15) to Eq. (4.17). The operator ∇^2_{d} involves no nonlinear convolution so it is kept on the left hand side as the implicit operator, while the remaining terms in ∇^2_{off} are lumped with the loading terms on the right hand side. By introducing an internal iteration index k, ϕ^{n+1} can be therefore solved as

$$\nabla^2_{\mathrm{d}}\left(\phi^{n+1}\right)^{k+1} = -\nabla^2_{\mathrm{off}}\left(\phi^{n+1}\right)^{k} + \frac{\alpha_0}{\Delta t}\left(\nabla\cdot\left[u^i\right]^*\right) \tag{4.61}$$

Similar to step 5 and 6, LSOR is chosen as the solver to sweep along 0-axis. After each iteration, the residue can be calculated by

$$R^{k+1} = \nabla^2\left(\phi^{n+1}\right)^{k+1} - \frac{\alpha_0}{\Delta t}\left(\nabla\cdot\left[u^i\right]^*\right) \tag{4.62}$$

The convergence therefore can be checked with the maximum residue $\left\|R^{k+1}\right\|_\infty$ being smaller than a tolerance. Numerical experiments shows that the number of iterations depends on the mapping: $x^0 : [-h+h', \eta] \to \xi^0 : [0, 1]$ requiring fewer steps while $x^0 : [-h+h', \eta] \to \xi^0 : [-h, 0]$ requiring more steps. The tolerance is suggested to be 10^{-10}.

10. *update velocity and pressure.* As long as ϕ^{n+1} is solved, $[u^i]^{n+1}$ can be updated by Eq. (4.30), while pressure P^{n+1} can be updated by Eq. (4.34).

11. *update η and advance.* As long as $[u^i]^{n+1}$ is updated, η can be updated by Eq. (4.48) or Eq. (4.49), depending on the stage of Runge-Kutta. If it has not reached the final stage, then repeat step 1 to 10. Otherwise, advancing a time step by rolling back u^i and P. A full time step is complete and we then move forward to the next time step.

4.7 Case study–wind generation on Cartesian coordinates

In this case, we study an air-water coupled wind generation process by following [46] with the identical configurations using Cartesian formulations on fixed grid. The work proposed in [46] is only based on linearized formulations. To improve this, we study the same case by using nonlinear normal stress surface condition. First, Eq. 4.35 can be expanded and written in the non-dimensionalized form as

$$\left[p^w - \frac{\eta}{\mathrm{Fr}} + \frac{2}{\mathrm{Re}^w} G^w\right] - \frac{\rho^a}{\rho^w}\left[p^a - \frac{\eta}{\mathrm{Fr}} + \frac{2}{\mathrm{Re}^a} G^a\right] = \frac{\kappa}{\mathrm{We}} \tag{4.63}$$

where $\mathrm{Fr} = \frac{U}{\sqrt{gL}}$ is Froude number; $\mathrm{Re}^a = \frac{\rho^a UL}{\mu^a}$ and $\mathrm{Re}^w = \frac{\rho^w UL}{\mu^w}$ are respectively air and water Reynolds numbers; $\mathrm{We} = \frac{\rho^w U^2 L}{\gamma}$ is the Weber number. $G^\alpha = \frac{M^\alpha}{2\mu^\alpha}$, M^α given by Eq. (4.37), and curvature κ given by Eq. (4.36) may differ between non-linear and linearized forms. Numerical results in this study will examine the differences.

1. Linearized forms [46]:

$$G^\alpha = \left[\frac{\partial u^0}{\partial x^0} + \frac{\partial v^1}{\partial y^1}\right] \tag{4.64}$$

$$\kappa = -\eta_{x1x1} - \eta_{x2x2} \tag{4.65}$$

2. Nonlinear forms, $G^\alpha = \frac{M^\alpha}{2\mu^\alpha}$, and κ remains the same as Eq. (4.36):

$$\begin{aligned} G^\alpha = \frac{1}{1+\eta_{x1}^2+\eta_{x2}^2}\Bigg[&\left(\frac{\partial u^0}{\partial x^0}\right) - \eta_{x1}\left(\frac{\partial u^0}{\partial x^1} + \frac{\partial u^1}{\partial x^0}\right) - \eta_{x2}\left(\frac{\partial u^0}{\partial x^2} + \frac{\partial u^2}{\partial x^0}\right) \\ &+\eta_{x1}^2\left(\frac{\partial u^1}{\partial x^1}\right) + \eta_{x1}\eta_{x2}\left(\frac{\partial u^1}{\partial x^2} + \frac{\partial u^2}{\partial x^1}\right) + \eta_{x2}^2\left(\frac{\partial u^2}{\partial x^2}\right)\Bigg] \end{aligned} \tag{4.66}$$

$$\kappa = \frac{-\eta_{x2x2}\left(1+\eta_{x1}^2\right) - \eta_{x1x1}\left(1+\eta_{x2}^2\right) + 2\eta_{x1}\eta_{x2}\eta_{x1x2}}{\left(1+\eta_{x1}^2+\eta_{x2}^2\right)^{\frac{3}{2}}} \tag{4.67}$$

4.7.1 Model configuration

As addressed in previous sections, pseudospectral method is employed as the numerical scheme to solve the model equations. Fourier basis is used to solve derivatives along x^2-axis (streamwise) and x^1-axis (spanwise), which implies that periodic boundary conditions (BCs) are applied along two horizontal directions. The x^0 direction is non-periodic and we use second order finite difference schemes to solve all vertical derivatives. Additionally, a non-uniform staggered grid is also used along vertical direction, shown as Figure 4.3. Since in this case all formulations are based on Cartesian coordinates and fixed grid, choosing the computational ξ-domain identical to the real x-domain will degenerate all transformed formulations degenerated to Cartesian ones, so that no curvilinear mapping is required.

Because the air and water domains are solved separately, inevitable "ghost grids" are placed above water domain and below air domain. Since Navier-Stokes equations are solved entirely in each domain, BCs are required on the ghost points, which are the boundary points for each domain. Thus, the BCs derived in the previous section are used to provide the boundary values for the ghost points.

As for the model configuration to perform the numerical cases, non-uniform computational grids are used along vertical direction. Finer grids are collocated near air-water interface for both domains, while coarser grids are collocated far from interface. The number of grids is $(N_2, N_1, N_0) = (64, 64, 65)$. The dimension of computational domain (L_2, L_1, L_0) equals to $(6h, 6h, h)$ in which the reference length scale $L = h$ is equal to $4cm$. Reference velocity is set to be $U = U_0 = 300(cm/s)$, so the reference time is $0.01333s$. The time step is 0.005 in non-dimensional units, which equals to 6.6667^{-5}s in dimensional units. A fractional step method, one of the project methods, is employed to solve Navier-Stokes equations, and low storage

second Runge-Kutta method is used for the time-marching scheme. The proposed model is also parallelized and the numerical case is performed using 8 processors.

According to our numerical experience, a successful simulation case should proceed in prescribed steps rather than naively triggered *ab initio* by only the shear wind. A complete simulation is sequentially performed in three stages, which are explained below.

1. *Stage I.* The first step is to assign the mean velocity profile analytically, and set a constraint for free surface to retain flat interface. Therefore, we spin up the turbulence by adding a buoyancy force in the x^0-momentum equation for 80 turnover time units (physically $1.0664s$).

2. *Stage II.* At the second stage the buoyancy force in the x^0-momentum equation is turned off, but we still continue the spin-up simulation for another 2400 large-eddy turnover time units to reach a pure shear-driven state (physically $32s$).

3. *Stage III.* In turn, we release the constraint to allow the flat interface to become freely deformable. Waves are then generated according to the prescribed air and water flow conditions, as well as the coupled BCs. Therefore, based on the fully developed shear-driven turbulent provided by the previous Stage II, we officially start our simulation when the waves reach fully developed state.

4.7.2 Flow snapshots

Figure 4.5 and 4.6 respectively show the water elevation η and streamwise velocity u at interface at 2.6s, 15.37s, 26.44s, and 66.8s. It can be observed that at 26.44s waves are in the transition state in which random waves become uniform. Before the transition region, wind waves occupy the initial linear growth stage, which is exemplified by highly random waves with shorter crests. After the transition region

waves are developed and enter the exponential growth stage in which gathered waves propagate toward uniform direction with long crest.

4.7.3 Wave growth

The rate of wave growth can be defined as $\langle \eta^2 \rangle$–mean-squre-of-η, which is the most significant indicator for the state of wind waves. The wave growth results from linearized normal stress BC and nonlinear normal stress BC are compared and shown in Figure 4.7. It is seen that at the linear growth stage ($t < 35s$) waves generated using the nonlinear normal stress BC amplify at a faster rate than those from linearized normal stress BC. However, the opposite occurs after the transition region: waves generated from the linearized normal stress BC grow at a faster rate at exponential growth stage. Therefore, there is a crossover point found at the transition stage at around $t = 37s$.

At the linear growth stage, we can compare the numerical results with the theoretical prediction using the formulation proposed by Phillips (1957):

$$\left\langle \eta^2 \right\rangle \approx \frac{\left\langle {p'_a}^2 \right\rangle}{2\sqrt{2}\rho_w^2 g \left(18 u_a^*\right)} t \tag{4.68}$$

where p'_a is the air pressure fluctuation near interface, and the friction velocity u_a^* is related to wind shear stress τ_s, calculated by

$$u_a^* = \sqrt{\frac{\tau_s}{\rho_a}} \tag{4.69}$$

The results are shown in Figure 4.8. It can be observed that at linear growth stage, the analytical result is underestimated comparing to the both DNS results.

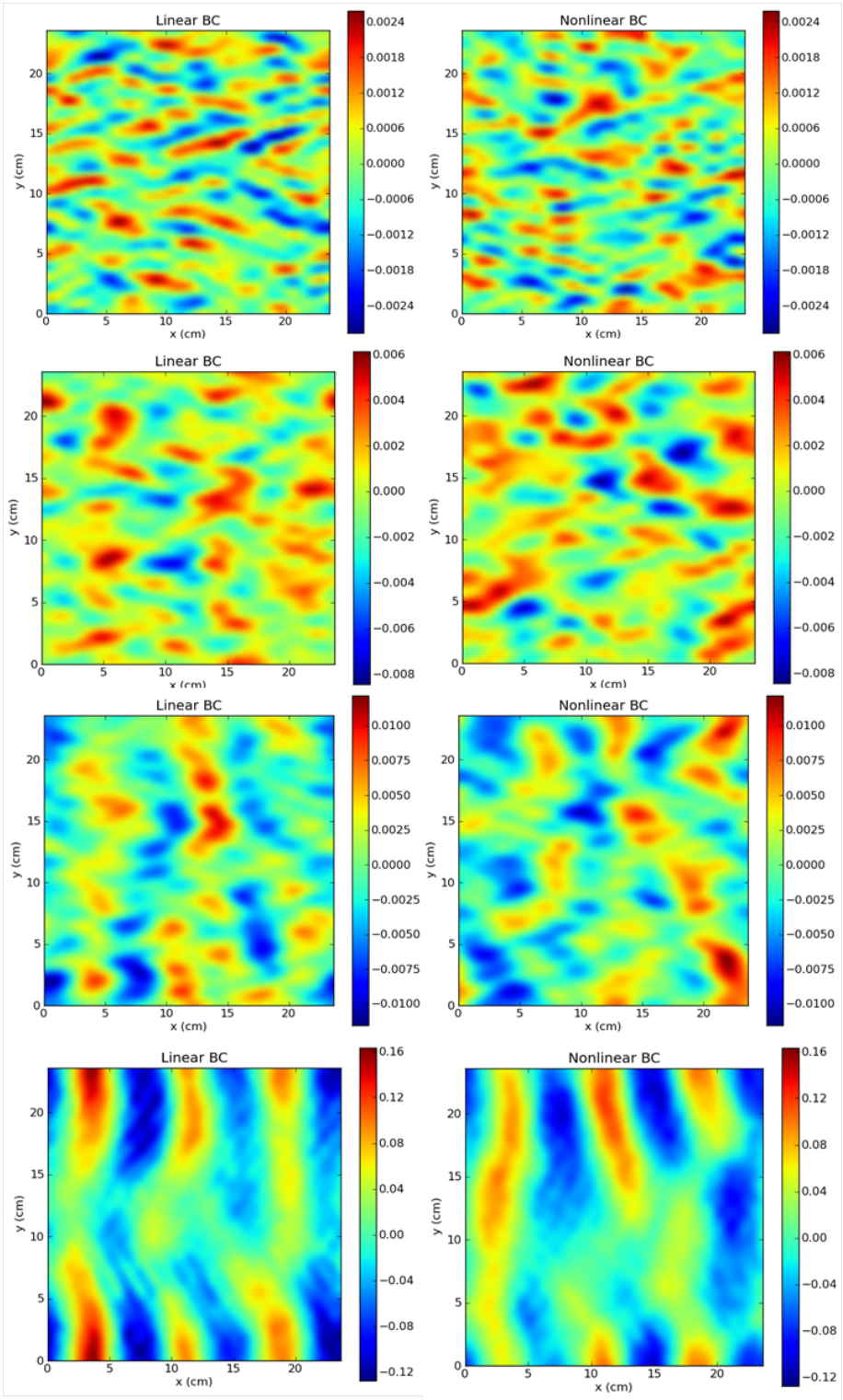


Figure 4.5: Water surface elevation η at time $t = 2.6$s, 15.37s, 26.44s, and 66.8s (top to bottom). Left column: results from linearized normal stress BC. Right column: results from nonlinear normal stress BC

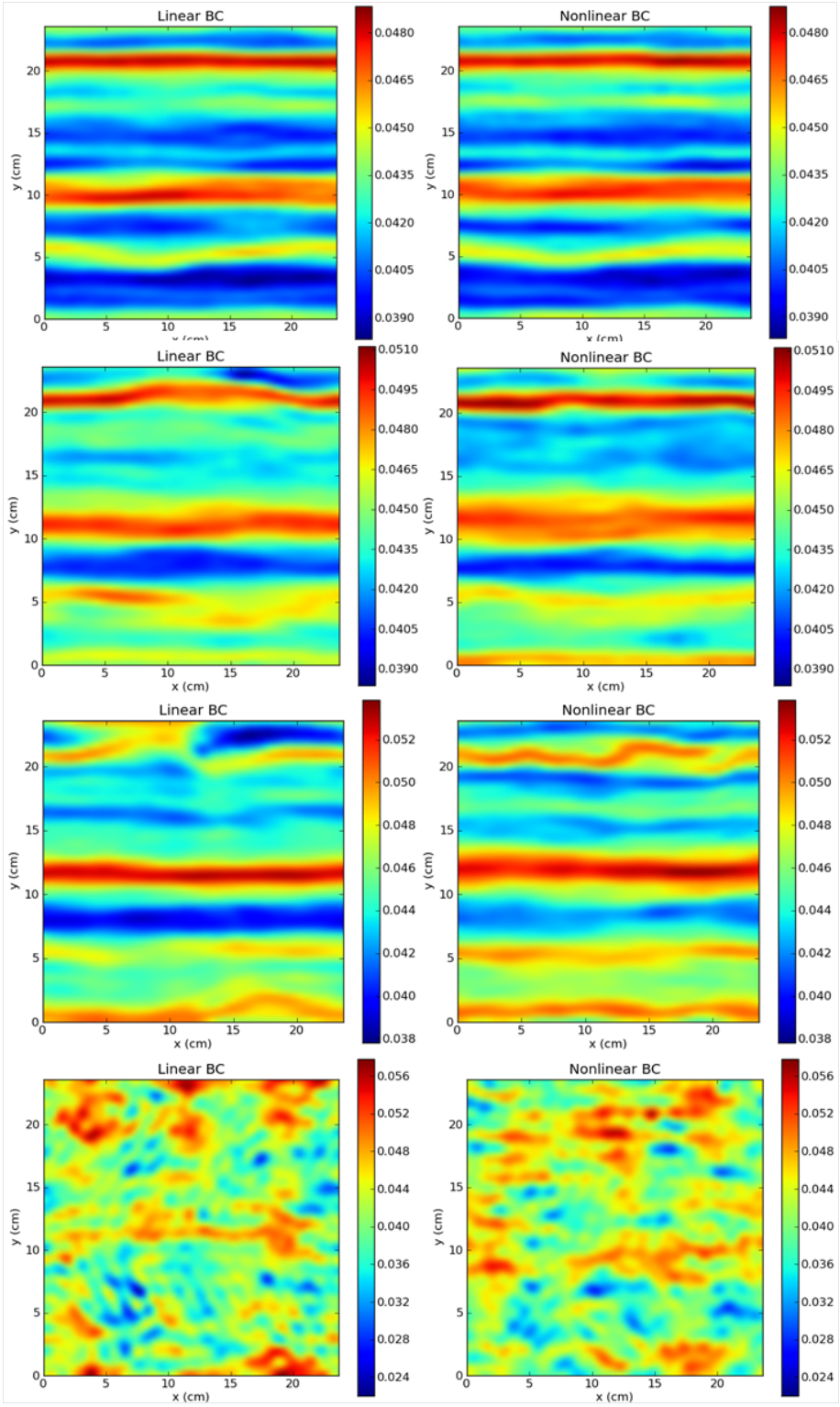
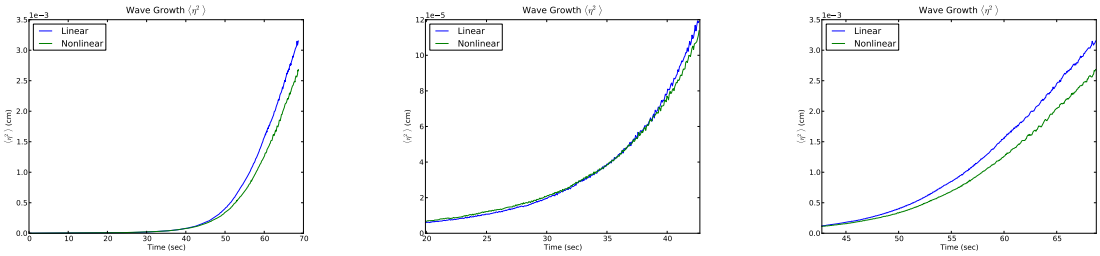


Figure 4.6: Streamwise velocity u at interface $x^0 = 0$ at time $t = 2.6$s, 15.37s, 26.44s, and 66.8s (top to bottom). Left column: results from linearized normal stress BC. Right column: results from nonlinear normal stress BC



(a) (b) (c)

Figure 4.7: Wave growth–root-mean-square of water elevation $\langle \eta^2 \rangle$. (a) shows full stage of wave growth. (b) shows the transition stage ($20s < t < 40s$). (c) shows the exponential gwoth stage ($t > 40s$). Green: numerical solution with nonlinear normal stress BC. Blue: numerical solution with linearized normal stress BC.

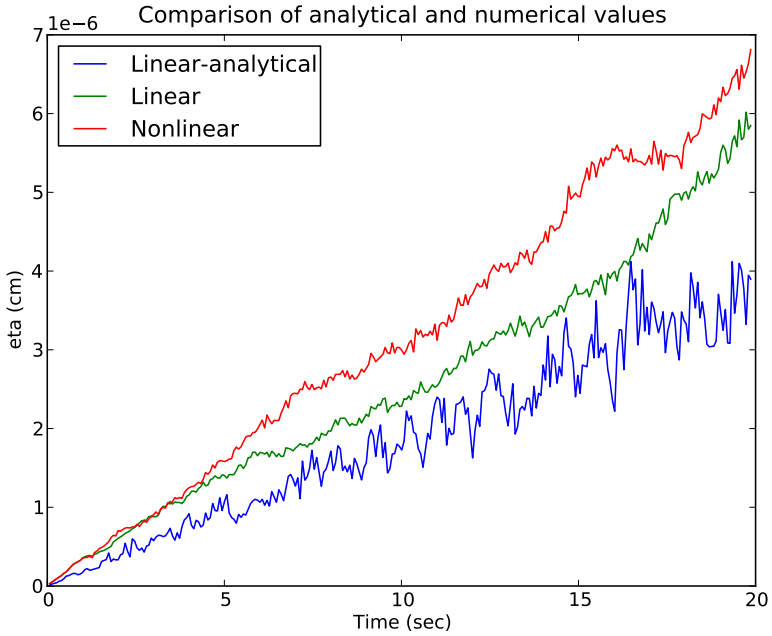


Figure 4.8: Comparison of analytical solution [57] and numerical solutions for linear wave growth at the initial stage($t < 20$s). Blue: analytical solution. Red: numerical solution with nonlinear normal stress BC. Green: numerical solution with linearized normal stress BC.

4.7.4 Evolution of interfacial properties

Figure 4.9 shows the evolution of some interfacial air properties. For both linear and nonlinear normal stress BC cases, wind shear stress τ_s remains almost constant in the linear growth stage and linearly increases in the exponential growth stage. The friction velocity u_a^*, which is related to τ and can be calculated by Eq (4.69), also has the same trend and gives the averaged value $u_a^* \approx 8.616$ (cm/sec). The root-mean-square of shear stress fluctuation $\sqrt{\langle \tau_s'^2 \rangle}$ shows similar trends for both linear and nonlinear cases. One reason why both cases can not be distinguished clearly for shear stress-related properties is because the shear stress BC used is the same as that of Lin et al. (2008). Due to different normal stress BCs, on the other hand, root-mean-square of pressure fluctuation $\sqrt{\langle p_a'^2 \rangle}$ and form stress D_p show different results between linear BC and nonlinear BC. The stress D_p is related to p_a' and can be calculated by

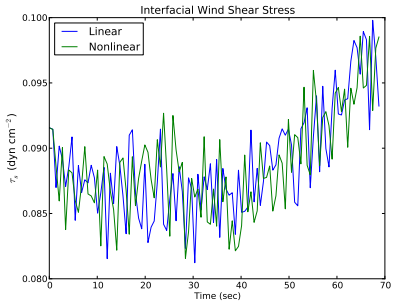
$$D_p = \frac{1}{L_1 L_2} \int \int p_a' \left(\frac{\partial \eta}{\partial x^1} + \frac{\partial \eta}{\partial x^2} \right) \mathrm{d}x^1 \mathrm{d}x^2 \tag{4.70}$$

It is found that at exponential growth stage, D_p for linear BC case is larger than nonlinear BC case, which is consistent with the results of wave growth in Figure 4.7. Accordingly, it can be concluded that at exponential growth stage using linearized normal stress BC causes over-estimated growth rate and form stress.

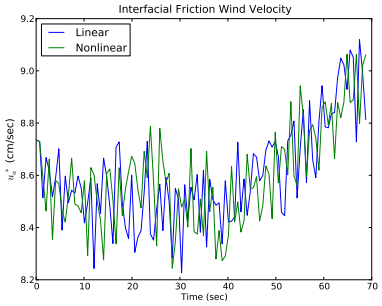
4.7.5 Summary

We offer several concluding remarks drawn for the work proposed by this work:

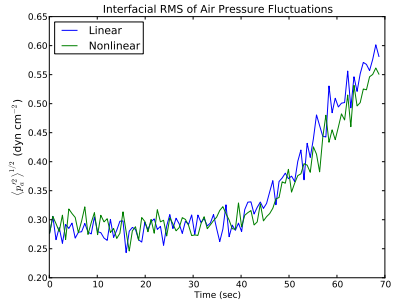
1. A high-resolution numerical tool has been developed for high Reynolds number problem using pseudospectral method. It is successfully applied for the simulation of air-water coupled two-phase flow.



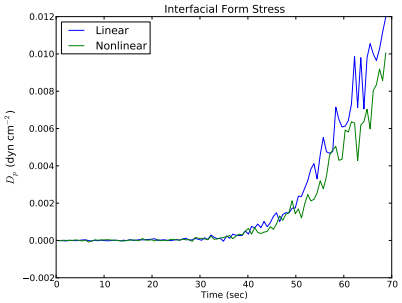
(a) Wind shear stress τ_s



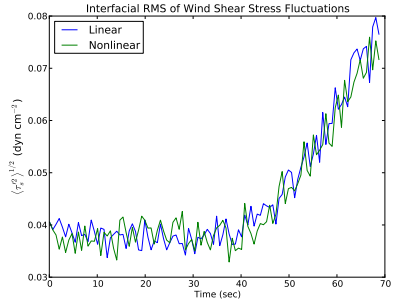
(b) Friction velocity u_a^*



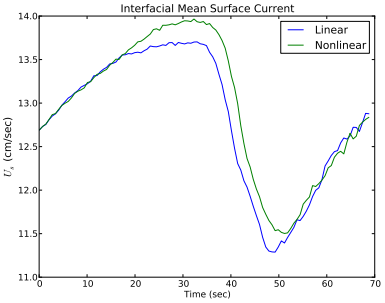
(c) Root-mean-square of pressure fluctuation $\sqrt{\langle p_a'^2 \rangle}$



(d) Form stress D_p



(e) Root-mean-square of shear stress fluctuation $\sqrt{\langle \tau_s'^2 \rangle}$



(f) Mean surface current U_s

Figure 4.9: Interfacial air properties

2. Linear and nonlinear normal stress BCs for wind-wave generation process have been compared and studied using DNS.

3. From the the results of wind-wave generation, we can conclude that

   (a) Linear growth ($t < 40$s) stage: for the growth rate, faster in the case of

nonlinear normal stress BC but slower in the case of linear normal stress BC.

(b) Exponential growth ($t > 40$s): for the growth rate, slower in the case of nonlinear normal stress BC but faster in the case of linear normal stress BC. Using linearized stress BC formulation, form stress D_p is also over-estimated at this stage.

4.8 Case study–wave modeling using surface-fitted moving grid

4.8.1 Decaying vortex

To test the convergence for the proposed algorithm, a series of benchmark cases are performed and compared. As associated exact solution exists, shown as Eq. (4.71) to Eq. (4.73), two-dimensional decaying vortex is an appropriate candidate to be used for comparison, particularly for unsteady Navier-Stokes problems.

$$u^1 = \sin(x^1)\cos(x^0)\exp(-2\nu t) \tag{4.71}$$

$$u^0 = -\cos(x^1)\sin(x^0)\exp(-2\nu t) \tag{4.72}$$

$$p = \frac{\rho}{4}[\cos(2x^0) + \cos(2x^1)]\exp(-4\nu t) \tag{4.73}$$

Owing to the pseudospectral method mixing two numerical schemes for spatial discretization where the spectral Fourier differentiation is used for x^1-axis, low order finite difference scheme for vertical x^0-axis therefore turns out to be the bottleneck that dominates the spatial convergence. Accordingly, given domain size $(L_0, L_1) = (2\pi, 2\pi)$, four cases are set up with (N_0, N_1) equal to $(40, 32)$, $(80, 32)$, $(160, 32)$, and $(320, 32)$, respectively. Grid size doubles only for Δx^0, while Δx^1 keeps constant. By fixing CFL=0.5, time step Δt can be computed for each case, which implies that smaller Δx^0 will associate with smaller Δt, and vice versa. Density and viscosity are

chosen to be $\rho = \nu = 1$. All cases are performed with total length of 0.5 seconds.

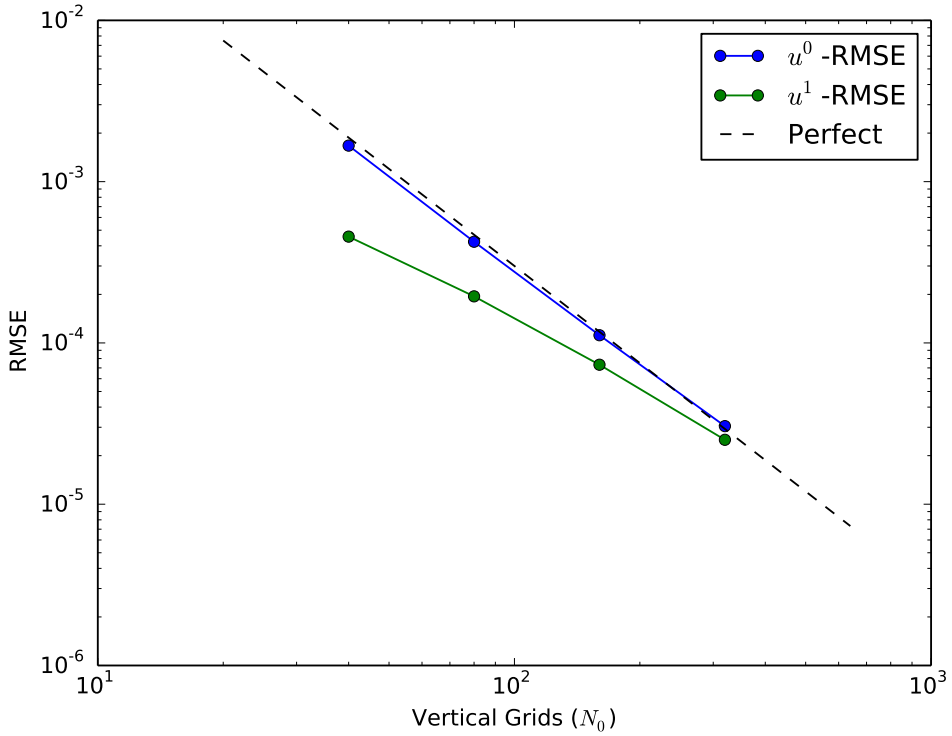


Figure 4.10: Results of decaying vortex for N_0 versus root-mean-square error of velocity (RMSE). Solid lines: numerical results. Dashed line: the perfect second order convergence.

Figure 4.10 shows the results for N_0 versus root-mean-square error (RMSE) of velocities u^0 and u^1. By comparing with exact solution Eq. (4.71) to Eq. (4.73), RMSE can be obtained by the average of whole time horizon, as well as average of all grid points. Owing to the second order of accuracy chosen for finite difference scheme, u^0 is expected with second order convergence with respect to N_0 (or Δx^0). The theoretical second order convergence with slope equal to -2 is shown as the black dash line. Obviously, u^0-RMSE shows perfect agreement with theoretical line, while

u^1-RMSE is not as optimal since it is not dominated by the vertical finite difference scheme.

4.8.2 Linear viscous wave

Considering a 2D linearized Navier-Stokes equations, aka Stokes' flow in that the nonlinear convection terms are neglected, according to [44], its exact solution for deep water waves propagating along x^1-direction can be described as

$$\begin{aligned}
\eta &= a_0 \exp(N) \sin(kx^1 + \omega t) \\
u^1_{pot} &= -\omega a_0 \exp(kx^0 + N) \sin(kx^1 + \omega t) \\
u^0_{pot} &= \omega a_0 \exp(kx^0 + N) \cos(kx^1 + \omega t) \\
u^1_{vis} &= 2\nu k \beta a_0 \exp(\beta x^0 + N)[\sin(\phi) - \cos(\phi)] \\
u^0_{vis} &= -2\nu k^2 a_0 \exp(\beta x^0 + N) \sin(\phi)
\end{aligned}$$

where a_0 is the amplitude and

$$\begin{aligned}
\omega &= \sqrt{gk + \gamma k^3} \\
\beta &= \sqrt{\omega / 2\nu} \\
N &= -2\nu k^2 t \\
\phi &= \beta x^0 + k x^1 + \omega t
\end{aligned}$$

and the resultant velocity can be obtained by the superposition of potential part u^i_{pot} and viscous part u^i_{vis}:

$$\begin{aligned}
u^0 &= u^0_{pot} + u^0_{vis} \\
u^1 &= u^1_{pot} + u^1_{vis}
\end{aligned}$$

To simulate this case, we choose $\gamma = 0$, $a_0 k = 0.01$, $k = \frac{2\pi}{\lambda} = 1$ where λ is the wave length, domain size $(L_0, L_1) = (3.5, 2\pi)$, and grids $(N_0, N_1) = (128, 64)$. Note that the streamwise length of domain L_1 is equals to the length of one wave, and the depth $L_0 > \lambda/2$ meets the assumption of deep water. As mentioned as Eq. (4.18) and Eq. (4.19), the parameters used in two-sided Vinokur grid generation are given by

$$S_0 = \Delta\xi_0/\Delta\xi_u = 1.52788$$

$$S_1 = \Delta\xi_{N_0}/\Delta\xi_u = 0.39725$$

which implies compression grids near the free surface while expansion grids near the bottom. In addition, Reynolds number $Re = \frac{c}{k\nu}$ is studied for $Re = 50$ and $Re = 500$ cases. Total length $2.7T$ is performed, where wave period $T = \frac{2\pi}{\omega}$, and the time step can be determined by given CFL=0.4.

The numerical results of decaying a_0 are shown as Figure 4.11. Comparing with exact solutions, it can be found that in the numerical results a_0 decays faster. It is possible because full Navier-Stokes equations are solved in both cases, which include the effects of nonlinear convections, whereas the exact solution is based on the assumption without the nonlinear effects, i.e., $Re << 1$. However, good agreements can be found for the case of smaller Reynolds number.

4.8.3 Stokes wave

Given $a_0 k \geq 0.1$, the nonlinearity can not be neglected. In this case we will perform the simulation of weekly nonlinear third order Stokes wave by giving $a_0 k = 0.1$. This is a good example to examine the robustness of the proposed numerical approach, particularly for the stability of the surface conditions. The known exact

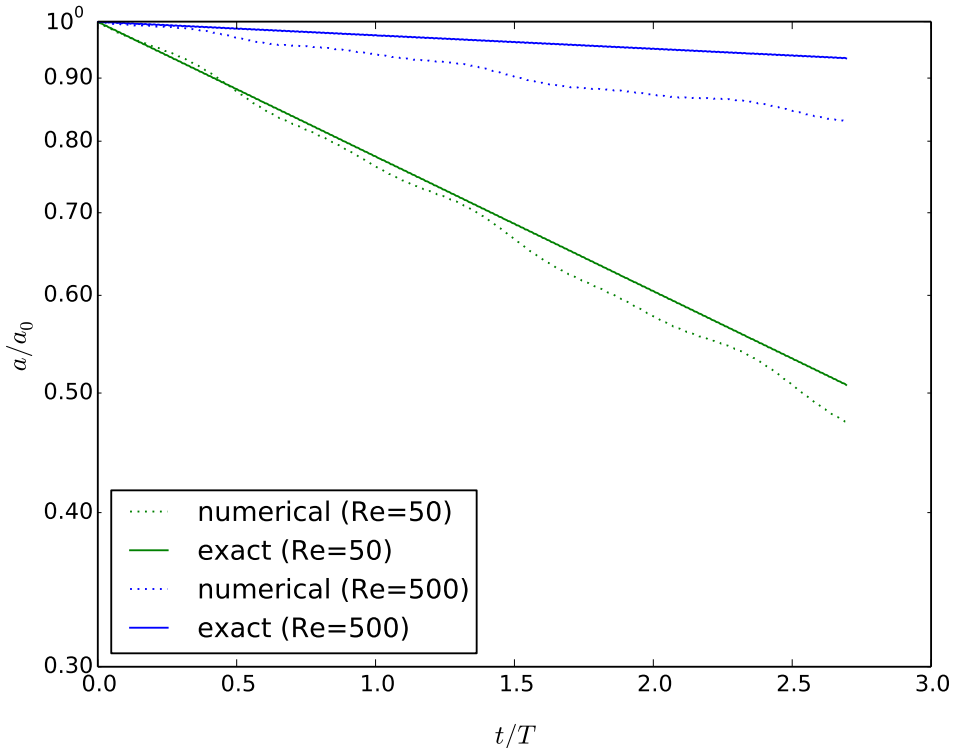


Figure 4.11: Results of a_0 decaying in linear viscous wave.

solution can be used for comparison, as well as the initial and bottom Dirichlet conditions. First of all, the frequency is given by

$$\omega = \sqrt{gk\tanh(kh)}\left\{1 + a_0^2k^2\left[\frac{8}{9}\left(\tanh^{-2}(kh) - 1\right)^2 + \tanh^{-2}(kh)\right]\right\} \tag{4.74}$$

Given $\phi = kx^1 - \omega t$, the surface elevation can be given by

$$\eta = a_0 \cos(\phi) + C_1 \cos(2\phi) - C_2 \cos(\phi) + C_3 \cos(3\phi) \tag{4.75}$$
$$C_1 = \frac{1}{4} a_0^2 k \tanh^{-1}(kh) \left(3 \tanh^{-2}(kh) - 1\right)$$
$$C_2 = \frac{3}{8} a_0^3 k^2 \left(\tanh^{-4}(kh) - 3 \tanh(kh) + 3\right)$$
$$C_3 = \frac{3}{64} a_0^3 k^2 \left[8 \tanh^{-6}(kh) + \left(\tanh^{-2}(kh) - 1\right)^2\right]$$

In addition, given $\psi = k(x^0 + h)$, the velocities u^1 (streamwise) and u^0 (vertical) can be respectively given by

$$u^1 = D_1 \cosh(\psi) \cos(\phi) + D_2 \cosh(2\psi) \cos(2\phi) + D_3 \cosh(3\psi) \cos(3\phi) \tag{4.76}$$
$$u^0 = D_1 \sinh(\psi) \sin(\phi) + D_2 \sinh(2\psi) \sin(2\phi) + D_3 \sinh(3\psi) \sin(3\phi) \tag{4.77}$$
$$D_1 = \frac{a_0 k g}{\omega} \cosh^{-1}(kh)$$
$$D_2 = \frac{3}{4} \frac{a_0^2 k^2 g}{\omega} \tanh^{-1}(kh) \left(\tanh^{-2}(kh) - 1\right)^2$$
$$D_3 = \frac{3}{64} \frac{a_0^3 k^3 g}{\omega} \left(\tanh^{-2}(kh) - 1\right) \left(\tanh^{-2}(kh) + 3\right) \left(9 \tanh^{-2}(kh) - 13\right) \cosh^{-1}(3kh)$$

The numerical case is set up with the domain size $(h, L_1) = (2\pi, 4\pi)$ with the grids $(N_0, N_1) = (97, 128)$ and the parameters for grid generation are the same as those used in the case of linear viscous wave, $S_0 = 1.52788$ and $S_1 = 0.39725$. For the wave parameters, we choose $a_0 k = 0.1$, and $k = \frac{2\pi}{\lambda} = 1$, which is equal to half of streamwise extent of domain. According to Eq. (4.74), having phase velocity $c = \frac{\omega}{k} = 3.1634$, and viscosity chosen as $\nu = 3.1634 \times 10^{-3}$, the Reynolds number of this case is therefore to be $Re = \frac{c}{k\nu} = 1000$. Wave period is $T = \frac{2\pi}{\omega} = 1.98621$s, and time step is chosen as $\Delta t = 0.013021$s. The initial grid given by the exact solution is shown as Figure 4.12. Since the wave unidirectionally propagates along x^1-axis,

the length along x^2-axis is not of significance. For x^2-axis, we choose $L_2 = \pi$ and 16 grids.

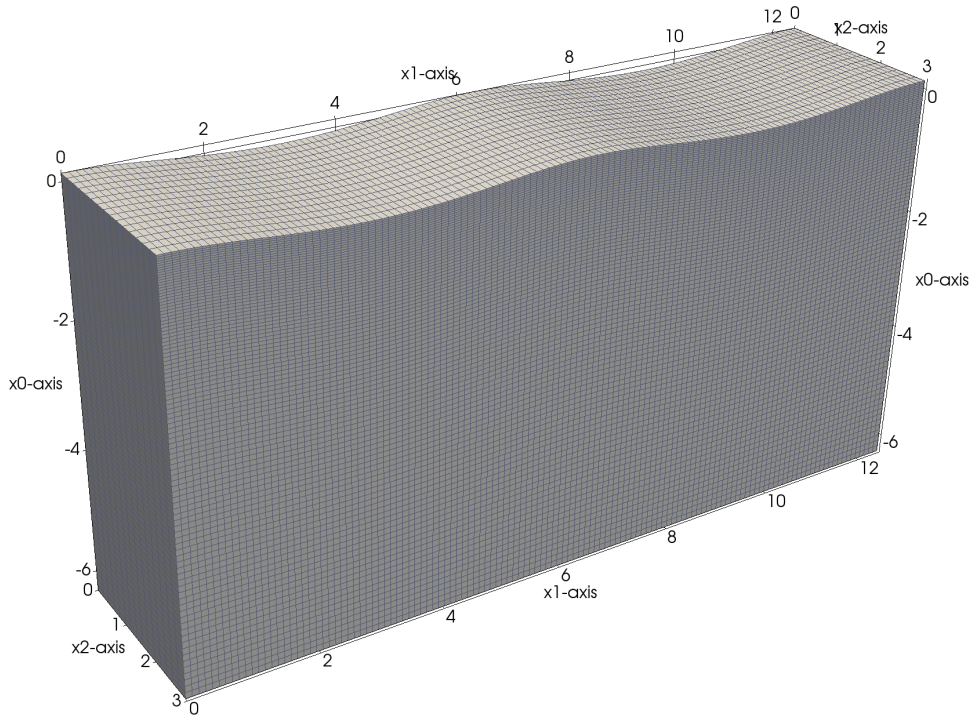
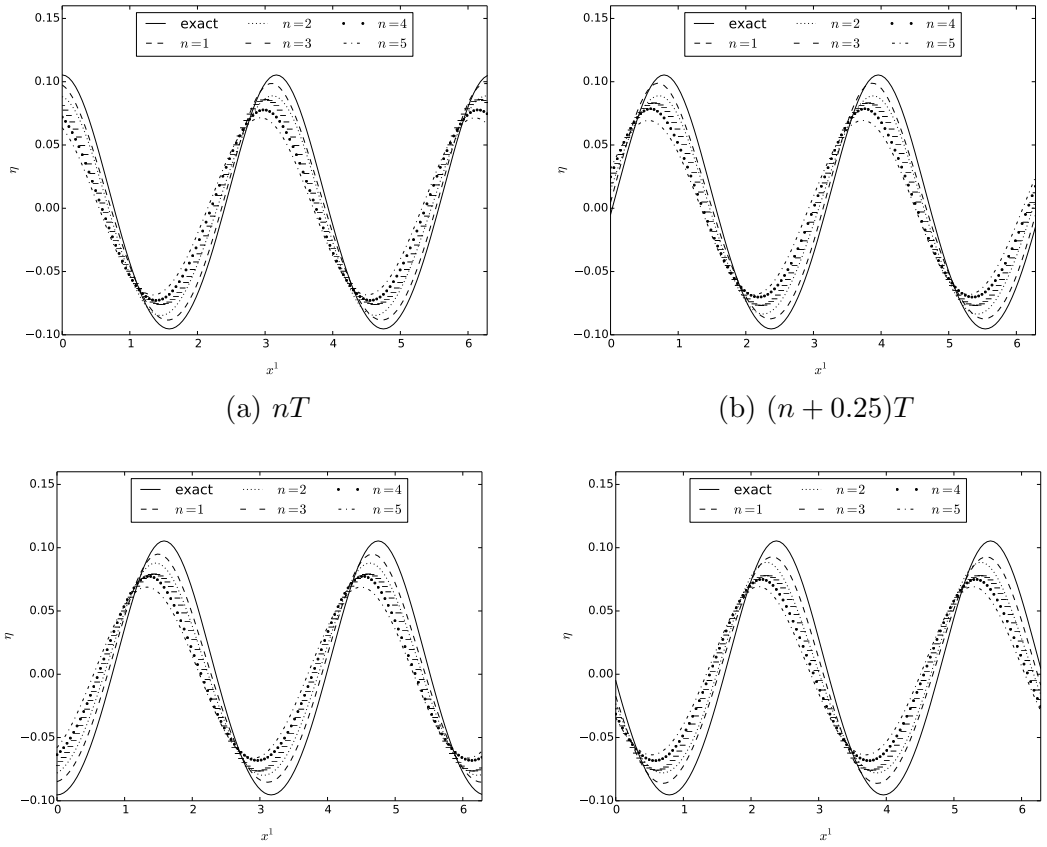


Figure 4.12: Initial grid for Stokes wave $a_0 k = 0.1$.

Figure 4.13 shows the η results for nT, $(n+0.25)T$, $(n+0.5)T$, and $(n+0.75)T$, where n ranges from 1 to 5. The exact solutions are delineated as the solid lines. Although the exact solution of Stokes wave mentioned above is used for the initial condition and Dirichlet bottom boundary condition, the governing equations being solved here are Navier-Stokes equations with the viscosity associated with $Re = 1000$. A not high enough Re value implies the viscous effects are still of significance. Therefore, waves damped by the viscosity can be observed in the results. The amplitude decays with respect to time, and the corresponding velocity also decays, which even-

tually makes the waves out of phase of exact solutions.

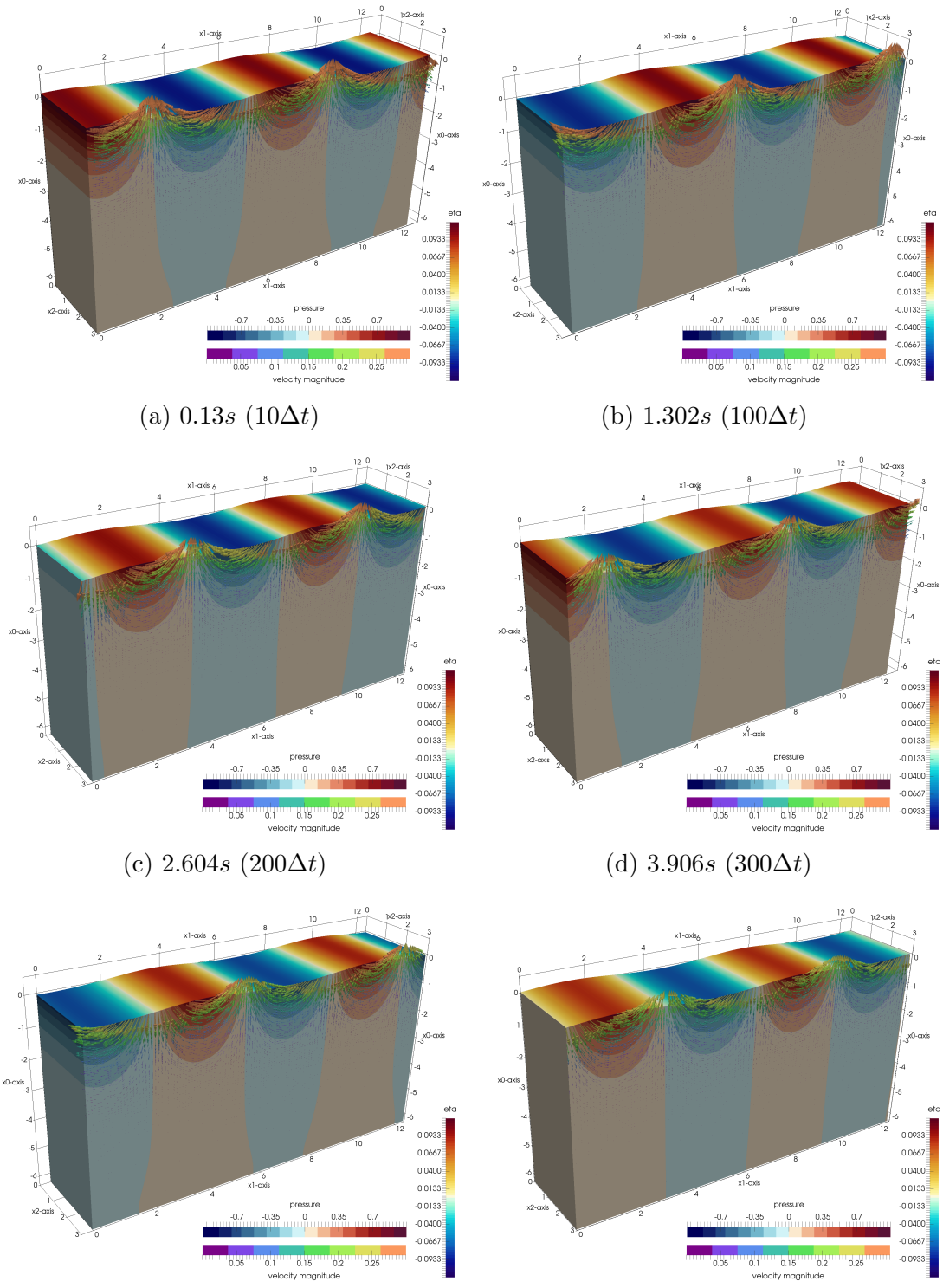
Figure 4.14 shows a couple of snapshots at different time steps, where bulk color represents the pressure, surface color represents η, and vector color represents the magnitude of velocity. The velocity vectors are randomly selected for representatives, and the vector lengths are scaled by the magnitudes of velocity. This case retains a pretty long execution, which implies that the proposed approach is stable enough for the numerical simulation of nonlinear waves.



(a) nT (b) $(n + 0.25)T$

(c) $(n + 0.5)T$ (d) $(n + 0.75)T$

Figure 4.13: η results for Stokes wave $a_0k = 0.1$



(a) $0.13s$ $(10\Delta t)$

(b) $1.302s$ $(100\Delta t)$

(c) $2.604s$ $(200\Delta t)$

(d) $3.906s$ $(300\Delta t)$

(e) $5.208s$ $(400\Delta t)$

(f) $6.511s$ $(500\Delta t)$

Figure 4.14: Snapshots for the results of Stokes wave $a_0 k = 0.1$. Surface color: η; Bulk color: pressure; Vector color: velocity magnitude. Velocity vectors are randomly selected for representatives.

5. CONCLUSIONS AND FUTURE WORKS

5.1 Conclusions

- *Part 1*- The first part of this research work is to employ the phase-average spectral model SWAN to investigate the wind-wave conditions at large scale and actual conditions. The study is for Persian Gulf and Qatar, particularly aiming at the shamal wind condition unique to the area. Several concluding remarks can be made:

    1. A 5-year (2004–2008) long-term hindcasting using COAMPS winds has been performed, and the seasonal and spatial features of wave climates have been examined by the spatial distribution of statistical parameters. Two-parameter Weibull regression has been applied to every grid point, so that the contour maps of scaling parameter B and shape parameter C can be plotted in terms of seasons, respectively for wind speed U, significant wave height H_s, and peak period T_p. Long-term statistics for Persian Gulf, particularly for wintertime ans summertime shamal seasons, have been discovered and concluded. The largest and second largest average wind speed U can be found in winter and summer, respectively. In spring season the magnitude of winds basically retains the distribution similar to winter, but with weakened magnitudes. The radial distribution can be found in the wintertime covering whole basin area in which peak resides at the Iran side, whereas the strong winds in summer are with long and wide distributions covering only the northern area of the entire basin. The seasonal contour maps of B for T_p shows that Persian Gulf is in a long-fetch wind wave condition. The shallow, flat, and long basin provides with the

conditions to make the magnitudes of T_p constantly increase southward along the central axis of Gulf.

2. The effects of bathymetry particularly around Qatar have been investigated by using a long-term hindcasting configured as same as the study for long-term wave climate. The 5-year total energy deviation (TED) has been used as the indicator to investigate the wave energy deviation between the case **origin** (default case) and the tuned cases, **noBrek** (turning off depth-induced breaking) or **noRefc** (turning off refraction). TED due to wave breaking (**noBrek**) is mainly found in the Straits of Hormuz, behind islands, and the nearshore regions. On the other hand, TED due to refraction (**noRefc**) is mainly found in most shallow area of the main basin, particularly in the southern area in the east of Qatar. The result also shows that TED due to refraction (in the range of $\pm 20\%$) is in an order of magnitude larger than that due to breaking (in the range of $\pm 2\%$).

3. The effects of boundary swells (with or without remotely-generated swells), hindcasting domain size (L1, L2, and L3) and sources of wind forcing (COAMPS, NCEP, and QTRSTA) have been inter-compared and investigated, by using a proposed multi-level hindcasting approach during October and November 2010. As a constant wind source applied to a small local area (L3 domain), QTRSTA, the in-situ measurement from our own experimental towers, has been successfully used in the hindcasting model. The results show that 1) wind sources dominates the trends of H_s and T_p; 2) consideration of incoming swells as boundary conditions results in more energetic wave conditions–higher H_s, and higher percentage of wave

conditions with $T_p > 4s$; and 3) the effect of hindcasting domain size is mitigated somewhat by the use of boundary conditions, which connects locally generated waves with those generated remotely over the domain L1.

4. An analysis tool of video imagery for single camera has been developed, including image rectification and CEOF analysis. The on-site wave parameters such as mean frequency, wave angle, and wavenumber, have been extracted from the video taken at the experimental pier. In addition, block analysis has also been applied to the entire shooting window, so as to identify the spatial distribution of wave parameters. However, the shooting range is too small and the recording time is too short, so the information for swells could not be captured and can not be really used for numerical comparisons and verifications. The future experiments will have to consider a wider range and longer recording time.

- *Part 2.* The development of SPX—a general PDE framework for structured grid. SPX is the first large-scale numerical framework designed and developed by applying C++-Concept and emphasizing the new features provided by C++1y. The conclusions of SPX development can be drawn with respect to two aspects:

  - *Software development.* C++-Concept, as an ongoing advanced software technology, has been firstly shown its success to be applied to the design of large-scale numerical framework. Two significant features of C++-Concept have been found extraordinarily useful in the design process:

    1. *Deduction and dispatch based on type behavior rather than type trait.* Type dispatch is based on the definition of concepts, and each concept

defines the requirements in a general way, by checking a set of behaviors of a single type or among a group of types. Comparing to traditional generic programming that type dispatch is usually through defined and matched type traits, C++-Concept has demonstrated more generality since many redundant checking functions for generating traits can be avoided. Moreover, a sophisticated interface according to the function arguments can be easily designed using concepts, i.e., the versatile interface of `spx::array::operator()( Args... )` for subscription and slicing, and the interface of out-class binary operator overloading by checking concept of `Binary_expressible`. The two examples have shown how C++-Concept helps for the interface design in terms of "type behaviors" of function arguments rather than any deduced type trait. Comparing to traditional generic programming that a deduced or matched type trait is necessary for each function generic argument, this approach can significantly reduce unnecessary functions for type trait deduction, and reduce the number of functions with different combinations of function arguments if they could have been grouped for the same purpose. In consequence, instead of coining a lot of type traits and many non-intuitive meta-programming techniques, C++-Concept delivers the direct support for developers that can focus on the design of interaction of generic types directly according to their behaviors in higher domain-specific level rather than programming level, so as to bring in more software abstraction.

2. *Concept overloading.* Static function overloading at compile time has shown to be used on the design of performance optimization, i.e., dispatching implementations in terms of constant expression. Tradi-

tional generic programming can achieve this purpose by several alternative manners, such as by "enable-if", by type trait, or by partial specialization. However, all of the solutions still rely on an intermediately defined or deduced type to be dispatched. The design and implementation of SPX has shown that C++-Concept overloading can deal with the overloading by directly evaluating constant expression itself, which results in more clean-cut and human-readable code. For example, given a template class, its a set of overloading member functions can be "enabled" or "disabled" at compile time by developer-defined constant expressions in terms of class or function template arguments, which avoid a bunch of partially specialized classes.

– *Numerical PDE framework.* Due to enhanced new features for static type system supported by C++-Concept and C++1y, SPX has been designed as a modern generic framework. The design of SPX emphasizes efficiency, extensibility, flexibility, and usability. There are three innovative highlights can be drawn as conclusions:

1. *Concept-based numerical array and expression template.* The concept-based designed high performance numerical array has been firstly developed. In particular, the versatile subscription and slicing in arbitrary rank via a rich interface whose, by using C++-Concept, arguments of variadic template can be described by requirements and be dispatched correspondingly by their inter-behaviors. Similar techniques are also applied to the design of dense descriptors and array storage. In addition, the expression template has been developed in which C++-Concept is particularly useful in the innovative design of

out-class binary operator overloading in terms of the "interoperability of given types" itself rather than any deduced type trait or any intermediate helper type tag.

2. *Scheme-free PDE expression and auto-deducible stencils.* PDE expression and operators are generally supported, by a proposed concept-based design of stencil operator integrating with expression template. The resulting stencil, representing the numerical-solvable field differential operator, can be automatically deduced by giving any PDE expression at any given grid point.
3. *Decoupled and decomposable numerical components.* Each component required in solving PDE is decoupled, decomposable and designed individually in generic manners. Users can ensemble those components in the way as flexible as they want, i.e., easily switching the algorithms for implicit solvers, or switching the differential basis.

In consequence, SPX provides a high-level software abstraction to make user easily deal with any PDE problem without involving implementation details. Instead, they can focus on the physical problem itself. For the CFD software vision in the future, SPX firstly proves that C++-Concept technology can be incorporated with the design of large-scale numerical framework by providing its powerful basis of generic abstraction without sacrificing efficiency.

- *Part 3.* By using CFD to solve three-dimensional Navier-Stokes equations, the detail of wave development has been investigated on small scale. A curvilinear surface-fitted moving grid model has been proposed to capture non-breaking waves in detail with fully nonlinear surface conditions. Examples show the ver-

ifications of proposed algorithms for linear viscous wave and nonlinear Stokes waves. By simplifying it to a fixed rectilinear grid based on Cartesian formulations, a two-phase 3D DNS model has been developed in which air and water phase are solved separately that are coupled by the interface conditions with nonlinear normal stress and linearized shear stress. By applying a shear wind at the top of air domain, the origin of the nature for wind-wave generation from the small scale has been studied. Owing the the detail results of velocity and pressure fields, the evolution of interface properties are also analyzed accordingly. For example, the rate of growth of surface elevation, time-dependent wind shear stress, friction velocity, pressure fluctuation, form stress, shear stress fluctuation, and mean surface current are all examined.

5.2 Future works

The linkage between small scale (part 3) and large scale (part 1) can be further developed and studied. For example, the results from CFD model can be integrated with the results from phase-average wind-wave model, or the drag coefficient obtained by DNS model can be used as the parameters in spectral source term for SWAN.

For software development, retaining the concept-based design, more elements can be further developed into SPX, i.e., parallel computing using GPU, SIMD, and MPI, more computational schemes such as unstructured grid and meshless methods, and more physical models such as turbulent modeling.

REFERENCES

[1] Eigen. http://eigen.tuxfamily.org/index.php.

[2] The NCEP climate forecast system reanalysis. *Bulletin of the American Meteorological Society*, 91(8):1015–1057, August 2010.

[3] ISO/IEC TS 19217:2015 - Information technology – Programming languages – C++ Extensions for concepts, 2015.

[4] M. A. Al-Zanaidi and W. H. Hui. Turbulent airflow over water waves-a numerical study. *Journal of Fluid Mechanics*, 148:225–246, November 1984.

[5] C Amante and B.W. Eakins. ETOPO1 one arc-minute global relief model: procedures, data sources and analysis. Technical report, National Ocean and Atmospheric Administration, 2009.

[6] Alejandro M. Aragón. A C++11 implementation of arbitrary-rank tensors for high-performance computing. *Computer Physics Communications*, 185(6):1681–1696, June 2014.

[7] Mats Aspnäs, Artur Signell, and Jan Westerholm. Efficient assembly of sparse matrices using hashing. In Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Waśniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, number 4699 in Lecture Notes in Computer Science, 900–907. Springer Berlin Heidelberg, January 2007.

[8] Hans-Jörg Barth. Characteristics of the wind regime north of Jubail, Saudi Arabia, based on high resolution wind data. *Journal of Arid Environments*, 47(3):387–402, March 2001.

[9] S. E. Belcher and J. C. R. Hunt. Turbulent shear flow over slowly moving waves. *Journal of Fluid Mechanics*, 251:109–148, June 1993.

[10] N. Booij, R. C. Ris, and L. H. Holthuijsen. A third-generation wave model for coastal regions: 1. Model description and validation. *Journal of Geophysical Research: Oceans*, 104(C4):7649–7666, 1999.

[11] David L. Brown, Ricardo Cortez, and Michael L. Minion. Accurate projection methods for the incompressible NavierStokes equations. *Journal of Computational Physics*, 168(2):464–499, April 2001.

[12] Luigi Cavaleri and Paola Malanotte Rizzoli. Wind wave prediction in shallow water: Theory and applications. *Journal of Geophysical Research: Oceans*, 86(C11):10961–10973, 1981.

[13] Dmitry V. Chalikov and Vladimir K. Makin. Models of the wave boundary layer. *Boundary-Layer Meteorology*, 56(1-2):83–99, July 1991.

[14] J. E. Cohen and S. E. Belcher. Turbulent shear flow over fast-moving waves. *Journal of Fluid Mechanics*, 386:345–371, May 1999.

[15] Fred W. Dobson. Measurements of atmospheric pressure on wind-generated sea waves. *Journal of Fluid Mechanics*, 48(01):91–127, July 1971.

[16] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[17] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988.

[18] Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ Concepts. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, 295–308. ACM, 2006.

[19] M. I. El-Sabh and T. S. Murty. Storm surges in the Arabian Gulf. *Natural Hazards*, 1(4):371–385, December 1989.

[20] Walid Elshorbagy, Mir Hammadul Azam, and Koichi Taguchi. Hydrodynamic characterization and modeling of the Arabian Gulf. *Journal of Waterway Port Coastal and Ocean Engineering*, 132(1):47–56, January 2006.

[21] K. O. Emery. Sediments and water of Persian Gulf. *AAPG Bulletin*, 40(10):2354–2383, 1956.

[22] Faculty of Civil Engineering and Geosciences. *SWAN User Manual (Cycle III Version 40.91)*. Environmental Fluid Mechanics Section, Delft University of Technology, 2012.

[23] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Grady Booch. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, November 1994.

[24] P. R. Gent and P. A. Taylor. A numerical model of the air flow above water waves. *Journal of Fluid Mechanics*, 77(01):105–128, September 1976.

[25] L. Gorrell, B. Raubenheimer, Steve Elgar, and R. T. Guza. SWAN predictions of waves observed in shallow water onshore of complex bathymetry. *Coastal Engineering*, 58(6):510–516, June 2011.

[26] Peter Gottschling. *Discovering Modern C++: An Intensive Course for Scientists, Engineers, and Programmers*. Addison-Wesley Professional, December 2015.

[27] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in C++. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, 291–310. ACM, 2006.

[28] Douglas Gregor, Jeremy Siek, Jeremiah Willcock, Jaakko Järvi, Ronald Garcia, and Andrew Lumsdaine. Concepts for C++ 0x revision. Technical report, August 2005.

[29] The Wamdi Group. The WAM model—A third generation ocean wave prediction model. *Journal of Physical Oceanography*, 18(12):1775–1810, December 1988.

[30] The WAMDI Group. The WAM modela third generation ocean wave prediction model. *Journal of Physical Oceanography*, 18(12):1775–1810, December 1988.

[31] Yvonne Gusdal, Ana Carrasco, Birgitte R. Furevik, and Øyvind Sætra. Validation of the operational wave model WAM and SWAN - 2009. Technical Report 18/2010 Oceanography, Norwegian Meteorological Institute, October 2010.

[32] Richard M. Hodur. The Naval Research Laboratory's coupled ocean/atmosphere mesoscale prediction system (COAMPS). *Monthly Weather Review*, 125(7):1414–1430, July 1997.

[33] K.T. Holland, R.A. Holman, T.C. Lippmann, J. Stanley, and N. Plant. Practical use of video imagery in nearshore oceanographic field studies. *IEEE Journal of Oceanic Engineering*, 22(1):81–92, January 1997.

[34] K. Iglberger, G. Hager, J. Treibig, and U. Rüde. Expression templates revisited: A performance analysis of current methodologies. *SIAM Journal on Scientific*

Computing, 34(2):C42–C69, January 2012.

[35] K. Iglberger, G. Hager, J. Treibig, and U. Rude. High performance smart expression template math libraries. In *2012 International Conference on High Performance Computing and Simulation (HPCS)*, 367–373, July 2012.

[36] S. J. Jacobs. An asymptotic theory for the turbulent flow over a progressive water wave. *Journal of Fluid Mechanics*, 174:69–80, January 1987.

[37] Jaakko Järvi, Douglas Gregor, Jeremiah Willcock, Andrew Lumsdaine, and Jeremy Siek. Algorithm specialization in generic programming: Challenges of constrained generics in C++. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, 272–282. ACM, 2006.

[38] Harold Jeffreys. On the formation of water waves by wind. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, 107(742):189–206, February 1925.

[39] Harold Jeffreys. On the formation of water waves by wind (second paper). *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, 110(754):241–247, February 1926.

[40] J.M. Kaihatu, K.L. Edwards, and W.C. O'Reilly. Model predictions of nearshore processes near complex bathymetry. In *OCEANS '02 MTS/IEEE*, volume 2, 685–691, October 2002.

[41] B. Kamranzad, A. Etemad-shahidi, and V. Chegini. Assessment of wave energy variation in the Persian Gulf. *Ocean Engineering*, 70:72–80, September 2013.

[42] D. A. Knoll and D. E. Keyes. Jacobian-free NewtonKrylov methods: a survey of approaches and applications. *Journal of Computational Physics*, 193(2):357–

397, January 2004.

[43] G. J. Komen, K. Hasselmann, and K. Hasselmann. On the existence of a fully developed wind-sea spectrum. *Journal of Physical Oceanography*, 14(8):1271–1285, August 1984.

[44] Horace Lamb. *Hydrodynamics*. New York,: Dover publications, 1945.

[45] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[46] Mei-Ying Lin, Chin-Hoh Moeng, Wu-Ting Tsai, Peter P. Sullivan, and Stephen E. Belcher. Direct numerical simulation of wind-wave generation processes. *Journal of Fluid Mechanics*, 616:1–30, December 2008.

[47] Dinesh Manian, James M. Kaihatu, and Emily M. Zechman. Using Genetic Algorithms to optimize bathymetric sampling for predictive model input. *Journal of Atmospheric and Oceanic Technology*, 29(3):464–477, October 2011.

[48] John W. Miles. On the generation of surface waves by shear flows. *Journal of Fluid Mechanics*, 3(02):185–204, November 1957.

[49] M.H. Moeini, A. Etemad-Shahidi, and V. Chegini. Wave modeling and extreme value analysis off the northern coast of the Persian Gulf. *Applied Ocean Research*, 32(2):209–218, April 2010.

[50] Mohammad Hadi Moeini, Amir Etemad-Shahidi, Vahid Chegini, and Iraj Rahmani. Wave data assimilation using a hybrid approach in the Persian Gulf. *Ocean Dynamics*, 62(5):785–797, May 2012.

[51] Mohammad Hadi Moeini, Amir Etemad-Shahidi, Vahid Chegini, Iraj Rahmani, and Mona Moghaddam. Error distribution and correction of the predicted wave

characteristics over the Persian Gulf. *Ocean Engineering*, 75:81–89, January 2014.

[52] M. Moradi and K. Kabiri. Spatial modeling and data management of the Persian Gulf wave atlas. In *2012 IEEE Colloquium on Humanities, Science and Engineering (CHUSER)*, 177–182, December 2012.

[53] S. Neelamani, K. Al-Salem, and K. Rakha. Extreme waves for Kuwaiti territorial waters. *Ocean Engineering*, 34(10):1496–1504, July 2007.

[54] S. Neelamani, K. Al-Salem, and K. Rakha. Extreme waves in the Arabian Gulf. *Journal of Coastal Research*, SI 50 (Proceedings of the 9th International Coastal Symposium):322–328, 2007.

[55] Mark Orzech, Jay Veeramony, and Stylianos Flampouris. Optimizing spectral wave estimates with adjoint-based sensitivity maps. *Ocean Dynamics*, 64(4):487–505, April 2014.

[56] A. Parvaresh, S. Hassanzadeh, and M.H. Bordbar. Statistical analysis of wave parameters in the north coast of the Persian Gulf. *Annales Geophysicae*, 23(6):2031–2038, 2005.

[57] O. M. Phillips. On the generation of waves by turbulent wind. *Journal of Fluid Mechanics*, 2(05):417–445, July 1957.

[58] Nathaniel G. Plant, Kacey L. Edwards, James M. Kaihatu, Jayaram Veeramony, Larry Hsu, and K. Todd Holland. The effect of bathymetric filtering on nearshore process model results. *Coastal Engineering*, 56(4):484–493, April 2009.

[59] K. Rakha, K. Al-Salem, and S. Neelamani. Hydrodynamic atlas for the Arabian Gulf. *Journal of Coastal Research*, SI 50 (Proceedings of the 9th International Coastal Symposium):550–554, 2007.

[60] D. Resio, S. Bratos, and E. Thompson. Meteorology and wave climate. In *Vincent, L., and Demirbilek, Z. (editors), Coastal Engineering Manual, Part II, Hydrodynamics*, number Chapter II-2, Engineer Manual 1110-2-1100. U.S. Army Corps of Engineers, 2002.

[61] D. S. Riley, M. A. Donelan, and W. H. Hui. An extended Miles' theory for wave generation by wind. *Boundary-Layer Meteorology*, 22(2):209–225, February 1982.

[62] R. C. Ris, L. H. Holthuijsen, and N. Booij. A third-generation wave model for coastal regions: 2. Verification. *Journal of Geophysical Research: Oceans*, 104(C4):7667–7681, 1999.

[63] W. E. Rogers, J. M. Kaihatu, H. A. H. Petit, N. Booij, and L. H. Holthuijsen. Diffusion reduction in an arbitrary scale third generation wind wave model. *Ocean Engineering*, 29(11):1357–1390, September 2002.

[64] W. Erick Rogers, James M. Kaihatu, Larry Hsu, Robert E. Jensen, James D. Dykes, and K. Todd Holland. Forecasting and hindcasting waves with the SWAN model in the Southern California Bight. *Coastal Engineering*, 54(1):1–15, January 2007.

[65] ISO SC22/WG21. Minutes of WG21 meeting, July 13, 2009, July 2009. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2920.html.

[66] Boris Schling. *The Boost C++ Libraries*. XML Press, 2011.

[67] A. Sheremet and G. W. Stone. Observations of nearshore wave dissipation over muddy sea beds. *Journal of Geophysical Research: Oceans*, 108(C11):3357, 2003.

[68] Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine. Concepts for C++ 0x. Technical report, January 2005.

[69] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In Denis Caromel, Rodney R. Oldehoeft, and Marydell Tholburn, editors, *Computing in Object-Oriented Parallel Environments*, number 1505 in Lecture Notes in Computer Science, 59–70. Springer Berlin Heidelberg, January 1998.

[70] J.G. Siek and A. Lumsdaine. The Matrix Template Library: generic components for high-performance scientific computing. *Computing in Science Engineering*, 1(6):70–71, November 1999.

[71] Arindam Singha and Reza Sadr. Characteristics of surface layer turbulence in coastal area of Qatar. *Environmental Fluid Mechanics*, 12(6):515–531, April 2012.

[72] R. L. Snyder, F. W. Dobson, J. A. Elliott, and R. B. Long. Array measurements of atmospheric pressure fluctuations above surface gravity waves. *Journal of Fluid Mechanics*, 102:1–59, January 1981.

[73] Hilary F. Stockdon and Rob A. Holman. Estimation of wave phase speed and nearshore bathymetry from video imagery. *Journal of Geophysical Research*, 105(C9):22015–22033, 2000.

[74] Bjarne Stroustrup. Parameterized types for C++. *Journal of Object-Oriented Programming*, 1(5):5–16, January 1989.

[75] Bjarne Stroustrup. The C++0x "remove concepts" decision, July 2009. http://www.drdobbs.com/cpp/the-c0x-remove-concepts-decision/218600111.

[76] Bjarne Stroustrup. *The C++ Programming Language, 4th Edition*. Addison-Wesley Professional, May 2013.

[77] Bjarne Stroustrup and Andrew Sutton. A concept design for the STL. Technical Report Tech. Rep N 3351, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee, 2012.

[78] Peter P. Sullivan, James C. McWILLIAMS, and Chin-Hoh Moeng. Simulation of turbulent flow over idealized water waves. *Journal of Fluid Mechanics*, 404:47–85, February 2000.

[79] Andrew Sutton. Origin, 2013. https://github.com/asutton/origin.

[80] Andrew Sutton and Bjarne Stroustrup. Design of concept libraries for C++. In Anthony Sloane and Uwe Aßmann, editors, *Software Language Engineering*, number 6940 in Lecture Notes in Computer Science, 97–118. Springer Berlin Heidelberg, January 2012.

[81] Andrew Sutton, Bjarne Stroustrup, and Gabriel Dos Reis. Concepts lite. 2013.

[82] Andrew Sutton, Bjarne Stroustrup, and Gabriel Dos Reis. Concepts lite: Constraining templates with predicates. 2013.

[83] Prasad G. Thoppil and Patrick J. Hogan. Persian Gulf response to a wintertime shamal wind event. *Deep Sea Research Part I: Oceanographic Research Papers*, 57(8):946–955, August 2010.

[84] Wu-ting Tsai and Li-ping Hung. Three-dimensional modeling of small-scale processes in the upper boundary layer bounded by a dynamic ocean surface. *Journal of Geophysical Research: Oceans*, 112(C2):C02019, 2007.

[85] U.S. Energy Information Admnistration. World oil transit chokepoints. *Analysis Briefs*, 1–14, 2012.

[86] Ap van Dongeren, Nathaniel Plant, Anna Cohen, Dano Roelvink, Merrick C. Haller, and Patricio Catalán. Beach wizard: nearshore bathymetry estimation

through assimilation of model computations and remote observations. *Coastal Engineering*, 55(12):1016–1027, December 2008.

[87] Cornelis A. Van Duin and Peter A. E. M. Janssen. An analytic model of the generation of surface gravity waves by turbulent air flow. *Journal of Fluid Mechanics*, 236:197–215, March 1992.

[88] Todd Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.

[89] Todd L. Veldhuizen. Arrays in Blitz++. In Denis Caromel, Rodney R. Oldehoeft, and Marydell Tholburn, editors, *Computing in Object-Oriented Parallel Environments*, number 1505 in Lecture Notes in Computer Science, 223–230. Springer Berlin Heidelberg, January 1998.

[90] Di Yang and Lian Shen. Simulation of viscous flows with undulatory boundaries. Part I: Basic solver. *Journal of Computational Physics*, 230(14):5488–5509, June 2011.

[91] Di Yang and Lian Shen. Simulation of viscous flows with undulatory boundaries: Part II. Coupling with other solvers for two-fluid computations. *Journal of Computational Physics*, 230(14):5510–5531, June 2011.

APPENDIX A

MATHEMATICAL DERIVATION

A.1 Surface dynamics

Considering an impermeable, continuously differentiable, topologically unchangeable (non-breaking), and no-slip liquid-gas interface represented, we have unit tangent vector $\mathbf{t_i}$ and unit normal vector $\mathbf{n^i}$ respectively defined by

$$\mathbf{t}_i = t^i \mathbf{a}_i = \frac{\mathbf{a}_i}{\|\mathbf{a}_i\|} = \frac{\mathbf{a}_i}{\sqrt{g_{ii}}} \tag{1}$$

$$\mathbf{n}_i = n_i \mathbf{a}^i = \frac{\mathbf{a}^i}{\|\mathbf{a}^i\|} = \frac{\mathbf{a}^i}{\sqrt{g^{ii}}} \tag{2}$$

where t^i and n_i are respectively contravariant and covariant components, $\mathbf{a}_i$ and $\mathbf{a}^i$ are respectively covariant and contravariant basis, and g_{ij} and g^{ij} are respectively covariant and contravariant metric tensors. Here we employ the surface-fitting curvilinear coordinates, i.e., the covariant basis always fit on the surface, so that we may further define the tensor index 0 along the direction penetrating the interface while the indices 1 and 2 are tangential directions. As illustrated in Figure A.1, the forces

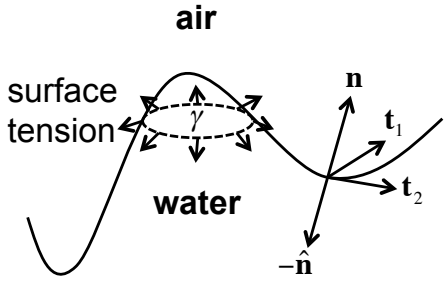


Figure A.1: Illustration of air-water interface.

projected onto the surface are balanced by the surface tension. Given a constant surface tension γ (force per unit length) for water, to represent the continuity of stress across interface, the stress balance equation can be written as

$$\mathbf{T}^a \cdot \mathbf{n} - \mathbf{T}^w \cdot \mathbf{n} = \gamma \mathbf{n} \left(\nabla \cdot \mathbf{n} \right) - \nabla \gamma \tag{3}$$

where $\nabla \cdot \mathbf{n}$ is the mean curvature, denoted as κ,

$$\kappa = \nabla \cdot \mathbf{n} \tag{4}$$

$\mathbf{T}$ is Cauchy stress tensor, i.e., for Newtonian Navier-Stokes equations on Cartesian coordinates

$$T^{ij} = -p\delta_j^i + \mu \left(\frac{\partial u^i}{\partial x^j} + \frac{\partial u^j}{\partial x^i} \right) \tag{5}$$

and the superscripts a and w indicate air and water phases respectively. $\mathbf{T}^a \cdot \mathbf{n}$ indicates the traction force exerted by air on water, while $\mathbf{T}^w \cdot \hat{\mathbf{n}} = -\mathbf{T}^w \cdot \mathbf{n}$ indicates the traction force exerted by water on air. Note that Eq. (3) contains both normal and tangential components. Taking $\mathbf{n} \cdot$ (3) and $\mathbf{t} \cdot$ (3) will separate Eq. (3) into the surface-normal and surface-tangential stress balance equations, which can be respectively written as below.

Normal Stress Balance:

$$\mathbf{n} \cdot \mathbf{T}^a \cdot \mathbf{n} - \mathbf{n} \cdot \mathbf{T}^w \cdot \mathbf{n} = \gamma \kappa \tag{6}$$

Tangential Stress Balance:

$$\mathbf{t} \cdot \mathbf{T}^a \cdot \mathbf{n} - \mathbf{t} \cdot \mathbf{T}^w \cdot \mathbf{n} = \mathbf{t} \cdot \nabla\gamma \tag{7}$$

Assuming γ is constant everywhere, we have $\nabla\gamma = 0$, and Eq. (7) becomes

$$\mathbf{t} \cdot \mathbf{T}^w \cdot \mathbf{n} = \mathbf{t} \cdot \mathbf{T}^a \cdot \mathbf{n} \tag{8}$$

If there is no external tangential stress from air phase,

$$\mathbf{t} \cdot \mathbf{T}^w \cdot \mathbf{n} = 0 \tag{9}$$

representing a shear-free surface. Accordingly, the relationship of continuity of tangential stress can be used for the Neumann boundary conditions for solving momentum equations in Navier-Stokes.

A.2 Derivation of surface stress in Cartesian domain

Given surface elevation $\eta = x^0 = f(x^1, x^2, t)$, a scalar function $F = x^0 - \eta$ can be defined in which the air-water interface can be implicitly expressed by the iso-surface of $F = 0$. Therefore, normal vector can be defined by

$$\mathbf{n} = \frac{\nabla F}{\|\nabla F\|} = \frac{(1, -\eta_{x1}, -\eta_{x2})}{\sqrt{1 + \eta_{x1}^2 + \eta_{x2}^2}} \tag{10}$$

where $()_{x1}$ indicates $\frac{\partial}{\partial x^1}$ and similar for $()_{x2}$. Similarly, according to Eq. (1), two tangent vector $\mathbf{t}_1$ and $\mathbf{t}_2$ respectively along covariant basis $\mathbf{a_1} = \frac{\partial \mathbf{r}}{\partial x^1}$ and $\mathbf{a_2} = \frac{\partial \mathbf{r}}{\partial x^2}$ in

Cartesian coordinates can be written as

$$\mathbf{t}_1 = \frac{\mathbf{a}_1}{\|\mathbf{a}_1\|} = \frac{\frac{\partial \mathbf{r}}{\partial x^1}}{\left\|\frac{\partial \mathbf{r}}{\partial x^1}\right\|} = \frac{(\eta_{x1}, 1, 0)}{\sqrt{\eta_{x1}^2 + 1}} \tag{11}$$

$$\mathbf{t}_2 = \frac{\mathbf{a}_2}{\|\mathbf{a}_2\|} = \frac{\frac{\partial \mathbf{r}}{\partial x^2}}{\left\|\frac{\partial \mathbf{r}}{\partial x^2}\right\|} = \frac{(\eta_{x2}, 0, 1)}{\sqrt{\eta_{x2}^2 + 1}} \tag{12}$$

where position vector at surface $\mathbf{r} = (\eta, x^1, x^2)$. The mean curvature κ can be easily obtained accordingly:

$$\kappa = \nabla \cdot \mathbf{n} = \frac{-\eta_{x2x2}\left(1 + \eta_{x1}^2\right) - \eta_{x1x1}\left(1 + \eta_{x2}^2\right) + 2\eta_{x1}\eta_{x2}\eta_{x1x2}}{\left(1 + \eta_{x1}^2 + \eta_{x2}^2\right)^{\frac{3}{2}}} \tag{13}$$

A.2.1 Derivation of surface normal stress

Expanding $\mathbf{n} \cdot \mathbf{T} \cdot \mathbf{n}$ in Eq. (6) in Cartesian coordinate by using Eq. (10) and symmetric $T^{ij} = T^{ji}$ results in

$$\begin{aligned}
\mathbf{n} \cdot \mathbf{T} \cdot \mathbf{n} &= \begin{bmatrix} n_0 & n_1 & n_2 \end{bmatrix} \begin{bmatrix} T^{00} & T^{01} & T^{02} \\ T^{10} & T^{11} & T^{12} \\ T^{20} & T^{21} & T^{22} \end{bmatrix} \begin{bmatrix} n_0 \\ n_1 \\ n_2 \end{bmatrix} \\
&= \frac{1}{\|\nabla F\|^2} \left(\begin{bmatrix} 1 & -\eta_{x1} & -\eta_{x2} \end{bmatrix} \begin{bmatrix} T^{00} & T^{01} & T^{02} \\ T^{10} & T^{11} & T^{12} \\ T^{20} & T^{21} & T^{22} \end{bmatrix} \begin{bmatrix} 1 \\ -\eta_{x1} \\ -\eta_{x2} \end{bmatrix} \right) \\
&= \frac{1}{\|\nabla F\|^2} \left(T^{00} - 2\eta_{x1} T^{01} - 2\eta_{x2} T^{02} + \eta_{x1}^2 T^{11} + 2\eta_{x1}\eta_{x2} T^{12} + \eta_{x2}^2 T^{22} \right)
\end{aligned} \tag{14}$$

which can be further expanded by using Eq. (5)

$$\begin{aligned}\mathbf{n}\cdot\mathbf{T}\cdot\mathbf{n} = \frac{1}{\|\nabla F\|^2}\Bigg\{ & -p + 2\mu\left(\frac{\partial u^0}{\partial x^0}\right) - 2\mu\eta_{x1}\left(\frac{\partial u^0}{\partial x^1}+\frac{\partial u^1}{\partial x^0}\right) - 2\mu\eta_{x2}\left(\frac{\partial u^0}{\partial x^2}+\frac{\partial u^2}{\partial x^0}\right) \\ & - p\eta_{x1}^2 + 2\mu\eta_{x1}^2\left(\frac{\partial u^1}{\partial x^1}\right) + 2\mu\eta_{x1}\eta_{x2}\left(\frac{\partial u^1}{\partial x^2}+\frac{\partial u^2}{\partial x^1}\right) \\ & -p\eta_{x2}^2 + 2\mu\eta_{x2}^2\left(\frac{\partial u^2}{\partial x^2}\right)\Bigg\} \end{aligned} \tag{15}$$

Note that $-p(1+\eta_{x1}^2+\eta_{x2}^2) = -p\,\|\nabla F\|^2$. After rearranging, the final result comes up with

$$\begin{aligned}\mathbf{n}\cdot\mathbf{T}\cdot\mathbf{n} = -p + \frac{2\mu}{\|\nabla F\|^2}\Bigg\{ & \left(\frac{\partial u^0}{\partial x^0}\right) - \eta_{x1}\left(\frac{\partial u^0}{\partial x^1}+\frac{\partial u^1}{\partial x^0}\right) - \eta_{x2}\left(\frac{\partial u^0}{\partial x^2}+\frac{\partial u^2}{\partial x^0}\right) \\ & +\eta_{x1}^2\left(\frac{\partial u^1}{\partial x^1}\right) + \eta_{x1}\eta_{x2}\left(\frac{\partial u^1}{\partial x^2}+\frac{\partial u^2}{\partial x^1}\right) + \eta_{x2}^2\left(\frac{\partial u^2}{\partial x^2}\right)\Bigg\} \end{aligned} \tag{16}$$

Substituting Eq. (16) back into Eq. (6) with the consideration of static pressure $\rho g\eta$, we have

$$-p^w + M^w + \rho^w g\eta = -p^a + M^a + \rho^a g\eta - \gamma\kappa \tag{17}$$

where

$$\begin{aligned}M^\alpha = \frac{2\mu^\alpha}{(1+\eta_{x1}^2+\eta_{x2}^2)}\Bigg\{ & \left(\frac{\partial u^0}{\partial x^0}\right) - \eta_{x1}\left(\frac{\partial u^0}{\partial x^1}+\frac{\partial u^1}{\partial x^0}\right) - \eta_{x2}\left(\frac{\partial u^0}{\partial x^2}+\frac{\partial u^2}{\partial x^0}\right) \\ & +\eta_{x1}^2\left(\frac{\partial u^1}{\partial x^1}\right) + \eta_{x1}\eta_{x2}\left(\frac{\partial u^1}{\partial x^2}+\frac{\partial u^2}{\partial x^1}\right) + \eta_{x2}^2\left(\frac{\partial u^2}{\partial x^2}\right)\Bigg\} \end{aligned} \tag{18}$$

The superscript α could be a or w to indicate air or water phase. For example, if $\alpha = a$ then velocities u^i are evaluated in air phase, and similar for water. If there is

no coupled air phase, the pressure on the water surface can be simplified as

$$p^w = \rho^w g\eta + M^w + \gamma\kappa \tag{19}$$

A.2.2 Derivation of surface tangential stress

Eq. (8) can be respectively written for $\mathbf{t}_1$ and $\mathbf{t}_2$ as below.

$$\mathbf{t}_1 \cdot \mathbf{T}^w \cdot \mathbf{n} = \mathbf{t}_1 \cdot \mathbf{T}^a \cdot \mathbf{n} \tag{20}$$

$$\mathbf{t}_2 \cdot \mathbf{T}^w \cdot \mathbf{n} = \mathbf{t}_2 \cdot \mathbf{T}^a \cdot \mathbf{n} \tag{21}$$

As defined $\sigma^\alpha_{t1} = \mathbf{t}_1 \cdot \mathbf{T}^\alpha \cdot \mathbf{n}$ and $\sigma^\alpha_{t2} = \mathbf{t}_2 \cdot \mathbf{T}^\alpha \cdot \mathbf{n}$ where α could be a for air or w for water, the two equations can be expanded respectively. First of all, expanding σ^α_{t1} by using Eq. (11), Eq. (10), and symmetric $T^{ij} = T^{ji}$ results in

$$\begin{aligned}
\sigma^\alpha_{t1} = \mathbf{t}_1 \cdot \mathbf{T} \cdot \mathbf{n} &= \begin{bmatrix} t_1^0 & t_1^1 & t_1^2 \end{bmatrix} \begin{bmatrix} T^{00} & T^{01} & T^{02} \\ T^{10} & T^{11} & T^{12} \\ T^{20} & T^{21} & T^{22} \end{bmatrix} \begin{bmatrix} n_0 \\ n_1 \\ n_2 \end{bmatrix} \\
&= \frac{1}{\|\mathbf{t}_1\| \, \|\nabla F\|} \left(\begin{bmatrix} \eta_{x1} & 1 & 0 \end{bmatrix} \begin{bmatrix} T^{00} & T^{01} & T^{02} \\ T^{10} & T^{11} & T^{12} \\ T^{20} & T^{21} & T^{22} \end{bmatrix} \begin{bmatrix} 1 \\ -\eta_{x1} \\ -\eta_{x2} \end{bmatrix} \right) \\
&= \frac{1}{\|\mathbf{t}_1\| \, \|\nabla F\|} \left(\eta_{x1} T^{00} + \underbrace{T^{01} - \eta_{x1}^2 T^{01}}_{=(1-\eta_{x1}^2)T^{01}} - \eta_{x1} T^{11} - \eta_{x1}\eta_{x2} T^{02} - \eta_{x2} T^{12} \right)
\end{aligned} \tag{22}$$

which can be further expanded by using Eq. (5)

$$\begin{aligned}\sigma_{t1}^{\alpha} = \frac{1}{\|\mathbf{t}_1\|\,\|\nabla F\|}\Bigg\{&\eta_{x1}\left(\cancel{p} + 2\mu^{\alpha}\frac{\partial u^0}{\partial x^0}\right) + (1-\eta_{x1}^2)\mu^{\alpha}\left(\frac{\partial u^0}{\partial x^1} + \frac{\partial u^1}{\partial x^0}\right)\\ &-\eta_{x1}\left(\cancel{p} + 2\mu^{\alpha}\frac{\partial u^1}{\partial x^1}\right) - \eta_{x1}\eta_{x2}\mu^{\alpha}\left(\frac{\partial u^0}{\partial x^2} + \frac{\partial u^2}{\partial x^0}\right)\\ &-\eta_{x2}\mu^{\alpha}\left(\frac{\partial u^1}{\partial x^2} + \frac{\partial u^2}{\partial x^1}\right)\Bigg\}\end{aligned} \tag{23}$$

where the pressure terms are eventually canceled out. On the other hand, similar expansion of σ_{t2}^{α} by using Eq. (12) and Eq. (10) results in

$$\begin{aligned}\sigma_{t2}^{\alpha} = \mathbf{t}_2\cdot\mathbf{T}\cdot\mathbf{n} &= \begin{bmatrix} t_2^0 & t_2^1 & t_2^2\end{bmatrix}\begin{bmatrix} T^{00} & T^{01} & T^{02}\\ T^{10} & T^{11} & T^{12}\\ T^{20} & T^{21} & T^{22}\end{bmatrix}\begin{bmatrix} n_0\\ n_1\\ n_2\end{bmatrix}\\ &= \frac{1}{\|\mathbf{t}_2\|\,\|\nabla F\|}\left(\begin{bmatrix}\eta_{x2} & 0 & 1\end{bmatrix}\begin{bmatrix} T^{00} & T^{01} & T^{02}\\ T^{10} & T^{11} & T^{12}\\ T^{20} & T^{21} & T^{22}\end{bmatrix}\begin{bmatrix} 1\\ -\eta_{x1}\\ -\eta_{x2}\end{bmatrix}\right)\\ &= \frac{1}{\|\mathbf{t}_2\|\,\|\nabla F\|}\left(\eta_{x2}T^{00} + T^{20} - \eta_{x1}\eta_{x2}T^{01} - \eta_{x1}T^{21} - \eta_{x2}^2T^{02} - \eta_{x2}T^{22}\right)\end{aligned} \tag{24}$$

where $T^{20} - \eta_{x2}^2T^{02} = (1-\eta_{x2}^2)T^{02}$. Substituting Eq. (5) results in

$$\begin{aligned}\sigma_{t2}^{\alpha} = \frac{1}{\|\mathbf{t}_2\|\,\|\nabla F\|}\Bigg\{&\eta_{x2}\left(\cancel{p} + 2\mu^{\alpha}\frac{\partial u^0}{\partial x^0}\right) - \eta_{x2}\left(\cancel{p} + 2\mu^{\alpha}\frac{\partial u^2}{\partial x^2}\right)\\ &-\eta_{x1}\mu^{\alpha}\left(\frac{\partial u^2}{\partial x^1} + \frac{\partial u^1}{\partial x^2}\right) + (1-\eta_{x2}^2)\mu^{\alpha}\left(\frac{\partial u^0}{\partial x^2} + \frac{\partial u^2}{\partial x^0}\right)\\ &-\eta_{x1}\eta_{x2}\mu^{\alpha}\left(\frac{\partial u^0}{\partial x^1} + \frac{\partial u^1}{\partial x^0}\right)\Bigg\}\end{aligned} \tag{25}$$

where the pressure terms are canceled out as well. By rearranging Eq. (23) and Eq. (25), the final results for σ^{α}_{t1} and σ^{α}_{t2} are respectively expressed as

$$\sigma^{\alpha}_{t1} = \frac{\mu^{\alpha}}{G_1}\left\{2\eta_{x1}\left(\frac{\partial u^0}{\partial x^0} - \frac{\partial u^1}{\partial x^1}\right) - \eta_{x2}\left(\frac{\partial u^1}{\partial x^2} + \frac{\partial u^2}{\partial x^1}\right) + (1-\eta^2_{x1})\left(\frac{\partial u^0}{\partial x^1} + \frac{\partial u^1}{\partial x^0}\right) - \eta_{x1}\eta_{x2}\left(\frac{\partial u^0}{\partial x^2} + \frac{\partial u^2}{\partial x^0}\right)\right\} \tag{26}$$

$$\sigma^{\alpha}_{t2} = \frac{\mu^{\alpha}}{G_2}\left\{2\eta_{x2}\left(\frac{\partial u^0}{\partial x^0} - \frac{\partial u^2}{\partial x^2}\right) - \eta_{x1}\left(\frac{\partial u^2}{\partial x^1} + \frac{\partial u^1}{\partial x^2}\right) + (1-\eta^2_{x2})\left(\frac{\partial u^0}{\partial x^2} + \frac{\partial u^2}{\partial x^0}\right) - \eta_{x1}\eta_{x2}\left(\frac{\partial u^0}{\partial x^1} + \frac{\partial u^1}{\partial x^0}\right)\right\} \tag{27}$$

where

$$G_1 = \sqrt{(1+\eta^2_{x1})(1+\eta^2_{x1}+\eta^2_{x2})} \tag{28}$$

$$G_2 = \sqrt{(1+\eta^2_{x2})(1+\eta^2_{x1}+\eta^2_{x2})} \tag{29}$$

A.3 Surface-fitted curvilinear grid and basic assumptions

Set

$$S^i_j = \frac{\partial \xi^i}{\partial x^j} \tag{30}$$

Given surface-fitted grid generated by vertically perturbing (along axis-0) the reference rectilinear domain with the height of surface elevation $\eta = x^0 = f(x^1, x^2, t)$, then we have

$$S^1_0 = S^2_0 = S^2_1 = S^1_2 = 0 \tag{31}$$

Note that the grid is non-orthogonal, but the skewness is small enough to be neglected. Therefore, by chain rule the differential operators can be transformed to ξ^i plane as

$$\frac{\partial}{\partial x^0} = S_0^0 \frac{\partial}{\partial \xi^0} + \cancel{S_0^1 \frac{\partial}{\partial \xi^1}} + \cancel{S_0^2 \frac{\partial}{\partial \xi^2}} = S_0^0 \frac{\partial}{\partial \xi^0} \tag{32}$$

$$\frac{\partial}{\partial x^1} = S_1^0 \frac{\partial}{\partial \xi^0} + S_1^1 \frac{\partial}{\partial \xi^1} + \cancel{S_1^2 \frac{\partial}{\partial \xi^2}} = S_1^0 \frac{\partial}{\partial \xi^0} + S_1^1 \frac{\partial}{\partial \xi^1} \tag{33}$$

$$\frac{\partial}{\partial x^2} = S_2^0 \frac{\partial}{\partial \xi^0} + \cancel{S_2^1 \frac{\partial}{\partial \xi^1}} + S_2^2 \frac{\partial}{\partial \xi^2} = S_2^0 \frac{\partial}{\partial \xi^0} + S_2^2 \frac{\partial}{\partial \xi^2} \tag{34}$$

The continuity equation can be transformed accordingly:

$$\frac{\partial u^0}{\partial x^0} + \frac{\partial u^1}{\partial x^1} + \frac{\partial u^2}{\partial x^2} = 0 \tag{35}$$

$$\Rightarrow \quad S_0^0 \frac{\partial u^0}{\partial \xi^0} + S_1^0 \frac{\partial u^1}{\partial \xi^0} + S_1^1 \frac{\partial u^1}{\partial \xi^1} + S_2^0 \frac{\partial u^2}{\partial \xi^0} + S_2^2 \frac{\partial u^2}{\partial \xi^2} = 0 \tag{36}$$

Further, assuming that horizontal axes between x-domain and ξ-domain are not only coincident but also linearly scaled, S_1^1 and S_2^2 are therefore to be constant. Hence,

$$S_1^1 = \text{constant} \Rightarrow \frac{\partial S_1^1}{\partial \xi^i} = 0 \tag{37}$$

$$S_2^2 = \text{constant} \Rightarrow \frac{\partial S_2^2}{\partial \xi^i} = 0 \tag{38}$$

A.4 The relationships between S_j^i and η at surface

The coordinate configuration in Cartesian domain is shown as Figure 4.2, where $h' = f(x^1, x^2)$ represents the possibly uneven bottom. On the other hand, as shown in Figure 4.3, given the arbitrary range of $x^0 : [z_2 + h', z_1 + \eta]$ mapped to reference domain with fixed $\xi^0 : [z2, z1]$, which identical to the range of unperturbed Cartesian

grid, the $\xi^0 = f(x^0, x^1, x^2)$ can be therefore defined as

$$\xi^0 = \frac{x^0 - z_2 - h'}{z_1 + \eta - z_2 - h'}(z_1 - z_2) + z_2 \underbrace{=}_{z_1 - z_2 = h} \frac{x^0 - z_2 - h'}{h + \eta - h'} h + z_2 \tag{39}$$

Accordingly, at surface $x^0 = z_1 + \eta$ we have S_j^0 terms:

$$\begin{aligned} S_0^0\big|_{x^0 = z_1 + \eta} = \frac{\partial \xi^0}{\partial x^0} &= \frac{(h + \eta - h')}{(h + \eta - h')^2} h \\ &= \frac{1}{h + \eta - h'} h \end{aligned} \tag{40}$$

$$\begin{aligned} S_1^0\big|_{x^0 = z_1 + \eta} = \frac{\partial \xi^0}{\partial x^1} &= \frac{-h'_{x1}(h + \eta - h') + (-\eta_{x1} + h'_{x1})(x^0 - z_2 - h')}{(h + \eta - h')^2} h \\ &\underbrace{=}_{z_1 - z_2 = h} \frac{\cancel{-h'_{x1}} + \cancel{h'_{x1}} - \eta_{x1}}{h + \eta - h'} h \\ &= \frac{-\eta_{x1}}{h + \eta - h'} h \end{aligned} \tag{41}$$

$$\begin{aligned} S_2^0\big|_{x^0 = z_1 + \eta} = \frac{\partial \xi^0}{\partial x^2} &= \frac{-h'_{x2}(h + \eta - h') + (-\eta_{x2} + h'_{x2})(x^0 - z_2 - h')}{(h + \eta - h')^2} h \\ &\underbrace{=}_{z_1 - z_2 = h} \frac{\cancel{-h'_{x2}} + \cancel{h'_{x2}} - \eta_{x2}}{h + \eta - h'} h \\ &= \frac{-\eta_{x2}}{h + \eta - h'} h \end{aligned} \tag{42}$$

where $()_{x1}$ indicates $\frac{\partial}{\partial x^1}$ and similar for $()_{x2}$. By comparing the results of S_1^0 and S_2^0 with the result of S_0^0, two important relationships can be found as

$$S_1^0 = -\eta_{x1} S_0^0 \tag{43}$$

$$S_2^0 = -\eta_{x2} S_0^0 \tag{44}$$

Note that these relationships are also held for other mapping ranges. For examples, [84] uses the mapping $x^0 : [-h, \eta] \rightarrow \xi^0 : [0, 1]$, and [90] uses the mapping $x^0 : [-h + h', \eta] \rightarrow \xi^0 : [0, 1]$. Both can obtain the same Eq. (43) and Eq. (44), so we do not repeat the derivations here.

A.5 Derivation of curvilinear M^α for surface normal stress

Having Eq. (18)

$$M^\alpha = \frac{2\mu^\alpha}{1 + \eta_{x1}^2 + \eta_{x2}^2} Q^\alpha \tag{45}$$

where

$$\begin{aligned} Q^\alpha = & \frac{\partial u^0}{\partial x^0} - \eta_{x1}\left(\frac{\partial u^0}{\partial x^1} + \frac{\partial u^1}{\partial x^0}\right) - \eta_{x2}\left(\frac{\partial u^0}{\partial x^2} + \frac{\partial u^2}{\partial x^0}\right) \\ & + \eta_{x1}^2\left(\frac{\partial u^1}{\partial x^1}\right) + \eta_{x1}\eta_{x2}\left(\frac{\partial u^1}{\partial x^2} + \frac{\partial u^2}{\partial x^1}\right) + \eta_{x2}^2\left(\frac{\partial u^2}{\partial x^2}\right) \end{aligned} \tag{46}$$

and α indicates the phase, i.e., dynamic viscosity μ^a and μ^w for air or water respectively. Replacing $\frac{\partial u^0}{\partial x^0}$ by continuity equation

$$\frac{\partial u^0}{\partial x^0} = -\frac{\partial u^1}{\partial x^1} - \frac{\partial u^2}{\partial x^2} \tag{47}$$

results in

$$\begin{aligned} Q^\alpha = & -\frac{\partial u^1}{\partial x^1} - \frac{\partial u^2}{\partial x^2} - \eta_{x1}\left(\frac{\partial u^0}{\partial x^1} + \frac{\partial u^1}{\partial x^0}\right) - \eta_{x2}\left(\frac{\partial u^0}{\partial x^2} + \frac{\partial u^2}{\partial x^0}\right) \\ & + \eta_{x1}^2\left(\frac{\partial u^1}{\partial x^1}\right) + \eta_{x1}\eta_{x2}\left(\frac{\partial u^1}{\partial x^2} + \frac{\partial u^2}{\partial x^1}\right) + \eta_{x2}^2\left(\frac{\partial u^2}{\partial x^2}\right) \end{aligned} \tag{48}$$

After rearranging we have

$$
\begin{aligned}
Q^{\alpha} = & (\eta_{x1}^2 - 1)\left(\frac{\partial u^1}{\partial x^1}\right) + (\eta_{x2}^2 - 1)\left(\frac{\partial u^2}{\partial x^2}\right) \\
& - \eta_{x1}\left(\frac{\partial u^0}{\partial x^1} + \frac{\partial u^1}{\partial x^0}\right) - \eta_{x2}\left(\frac{\partial u^0}{\partial x^2} + \frac{\partial u^2}{\partial x^0}\right) + \eta_{x1}\eta_{x2}\left(\frac{\partial u^1}{\partial x^2} + \frac{\partial u^2}{\partial x^1}\right)
\end{aligned} \tag{49}
$$

Applying Eq. (32) to (34) to expand Q^{α} in curvilinear domain results in

$$
\begin{aligned}
Q^{\alpha} = & (\eta_{x1}^2 - 1)\left(S_1^0\frac{\partial u^1}{\partial \xi^0} + S_1^1\frac{\partial u^1}{\partial \xi^1}\right) + (\eta_{x2}^2 - 1)\left(S_2^0\frac{\partial u^2}{\partial \xi^0} + S_2^2\frac{\partial u^2}{\partial \xi^2}\right) \\
& - \eta_{x1}\left(S_1^0\frac{\partial u^0}{\partial \xi^0} + S_1^1\frac{\partial u^0}{\partial \xi^1} + S_0^0\frac{\partial u^1}{\partial \xi^0}\right) - \eta_{x2}\left(S_2^0\frac{\partial u^0}{\partial \xi^0} + S_2^2\frac{\partial u^0}{\partial \xi^2} + S_0^0\frac{\partial u^2}{\partial \xi^0}\right) \\
& + \eta_{x1}\eta_{x2}\left(S_2^0\frac{\partial u^1}{\partial \xi^0} + S_2^2\frac{\partial u^1}{\partial \xi^2} + S_1^0\frac{\partial u^2}{\partial \xi^0} + S_1^1\frac{\partial u^2}{\partial \xi^1}\right)
\end{aligned} \tag{50}
$$

After rearranging we have

$$
\begin{aligned}
Q^{\alpha} = & \underbrace{(-\eta_{x2}^2 - 1)\left(S_1^1\frac{\partial u^1}{\partial \xi^1}\right)}_{Q_1^{\alpha}} + (\eta_{x1}^2 + \eta_{x2}^2)\left(S_1^1\frac{\partial u^1}{\partial \xi^1}\right) + (\eta_{x1}^2 - 1)\left(S_1^0\frac{\partial u^1}{\partial \xi^0}\right) \\
& + \underbrace{(-\eta_{x1}^2 - 1)\left(S_2^2\frac{\partial u^2}{\partial \xi^2}\right)}_{Q_1^{\alpha}} + (\eta_{x1}^2 + \eta_{x2}^2)\left(S_2^2\frac{\partial u^2}{\partial \xi^2}\right) + (\eta_{x2}^2 - 1)\left(S_2^0\frac{\partial u^2}{\partial \xi^0}\right) \\
& \underbrace{-\eta_{x1}\left(S_1^1\frac{\partial u^0}{\partial \xi^1}\right)}_{Q_1^{\alpha}} - \eta_{x1}\left(S_1^0\frac{\partial u^0}{\partial \xi^0} + S_0^0\frac{\partial u^1}{\partial \xi^0}\right) \\
& \underbrace{-\eta_{x2}\left(S_2^2\frac{\partial u^0}{\partial \xi^2}\right)}_{Q_1^{\alpha}} - \eta_{x2}\left(S_2^0\frac{\partial u^0}{\partial \xi^0} + S_0^0\frac{\partial u^2}{\partial \xi^0}\right) \\
& \underbrace{+\eta_{x1}\eta_{x2}\left(S_2^2\frac{\partial u^1}{\partial \xi^2} + S_1^1\frac{\partial u^2}{\partial \xi^1}\right)}_{Q_1^{\alpha}} + \eta_{x1}\eta_{x2}\left(S_2^0\frac{\partial u^1}{\partial \xi^0} + S_1^0\frac{\partial u^2}{\partial \xi^0}\right)
\end{aligned} \tag{51}
$$

which can be further written in the decomposition

$$Q^{\alpha} \quad = Q_1^{\alpha} + Q_2^{\alpha} \tag{52}$$

where

$$\begin{aligned} Q_1^{\alpha} = &-(\eta_{x2}^2+1)\left(S_1^1\frac{\partial u^1}{\partial \xi^1}\right) - (\eta_{x1}^2+1)\left(S_2^2\frac{\partial u^2}{\partial \xi^2}\right) \\ &- \eta_{x1}\left(S_1^1\frac{\partial u^0}{\partial \xi^1}\right) - \eta_{x2}\left(S_2^2\frac{\partial u^0}{\partial \xi^2}\right) + \eta_{x1}\eta_{x2}\left(S_2^2\frac{\partial u^1}{\partial \xi^2} + S_1^1\frac{\partial u^2}{\partial \xi^1}\right) \end{aligned} \tag{53}$$

and

$$\begin{aligned} Q_2^{\alpha} = &(\eta_{x1}^2+\eta_{x2}^2)\left(S_1^1\frac{\partial u^1}{\partial \xi^1}\right) + (\eta_{x1}^2-1)\left(S_1^0\frac{\partial u^1}{\partial \xi^0}\right) \\ &+ (\eta_{x1}^2+\eta_{x2}^2)\left(S_2^2\frac{\partial u^2}{\partial \xi^2}\right) + (\eta_{x2}^2-1)\left(S_2^0\frac{\partial u^2}{\partial \xi^0}\right) \\ &- \eta_{x1}\left(S_1^0\frac{\partial u^0}{\partial \xi^0} + S_0^0\frac{\partial u^1}{\partial \xi^0}\right) - \eta_{x2}\left(S_2^0\frac{\partial u^0}{\partial \xi^0} + S_0^0\frac{\partial u^2}{\partial \xi^0}\right) + \eta_{x1}\eta_{x2}\left(S_2^0\frac{\partial u^1}{\partial \xi^0} + S_1^0\frac{\partial u^2}{\partial \xi^0}\right) \\ &= 0 \end{aligned} \tag{54}$$

To prove $Q_2^{\alpha} = 0$, first of all we have to use Eq. (43) and (44). Rearranging Q_2^{α} results in

$$\begin{aligned} Q_2^{\alpha} = &(\eta_{x1}^2+\eta_{x2}^2)\left(S_1^1\frac{\partial u^1}{\partial \xi^1}\right) + (\eta_{x1}^2+\eta_{x2}^2)\left(S_2^2\frac{\partial u^2}{\partial \xi^2}\right) \\ &+ [\eta_{x1}^2 S_1^0 \underbrace{-S_1^0 - \eta_{x1}S_0^0}_{=0} + \eta_{x1}\eta_{x2}S_2^0]\left(\frac{\partial u^1}{\partial \xi^0}\right) \\ &+ [\eta_{x2}^2 S_2^0 \underbrace{-S_2^0 - \eta_{x2}S_0^0}_{=0} + \eta_{x1}\eta_{x2}S_1^0]\left(\frac{\partial u^2}{\partial \xi^0}\right) \\ &- \left[\eta_{x1}S_1^0 + \eta_{x2}S_2^0\right]\left(\frac{\partial u^0}{\partial \xi^0}\right) \end{aligned} \tag{55}$$

Therefore, Q_2^α comes up with

$$\begin{aligned} Q_2^\alpha = &(\eta_{x1}^2 + \eta_{x2}^2)\left(S_1^1 \frac{\partial u^1}{\partial \xi^1}\right) + (\eta_{x1}^2 + \eta_{x2}^2)\left(S_2^2 \frac{\partial u^2}{\partial \xi^2}\right) \\ &+ \eta_{x1}(\eta_{x1} S_1^0 + \eta_{x2} S_2^0)\left(\frac{\partial u^1}{\partial \xi^0}\right) + \eta_{x2}(\eta_{x2} S_2^0 + \eta_{x1} S_1^0)\left(\frac{\partial u^2}{\partial \xi^0}\right) \\ &- (\eta_{x1} S_1^0 + \eta_{x2} S_2^0)\left(\frac{\partial u^0}{\partial \xi^0}\right) \end{aligned} \tag{56}$$

Again by using Eq. (43) and (44), we have

$$\eta_{x1} S_1^0 + \eta_{x2} S_2^0 = -\eta_{x1}^2 S_0^0 - \eta_{x2}^2 S_0^0 = -S_0^0(\eta_{x1}^2 + \eta_{x2}^2) \tag{57}$$

Substituting this fact back to the last three terms of Q_2^α comes up with

$$\begin{aligned} Q_2^\alpha = &(\eta_{x1}^2 + \eta_{x2}^2)\left(S_1^1 \frac{\partial u^1}{\partial \xi^1}\right) + (\eta_{x1}^2 + \eta_{x2}^2)\left(S_2^2 \frac{\partial u^2}{\partial \xi^2}\right) \\ &- \eta_{x1} S_0^0(\eta_{x1}^2 + \eta_{x2}^2)\left(\frac{\partial u^1}{\partial \xi^0}\right) - \eta_{x2} S_0^0(\eta_{x1}^2 + \eta_{x2}^2)\left(\frac{\partial u^2}{\partial \xi^0}\right) \\ &+ S_0^0(\eta_{x1}^2 + \eta_{x2}^2)\left(\frac{\partial u^0}{\partial \xi^0}\right) \end{aligned} \tag{58}$$

Hence,

$$Q_2^\alpha = (\eta_{x1}^2 + \eta_{x2}^2)\left\{S_1^1\left(\frac{\partial u^1}{\partial \xi^1}\right) + S_2^2\left(\frac{\partial u^2}{\partial \xi^2}\right) - \eta_{x1} S_0^0\left(\frac{\partial u^1}{\partial \xi^0}\right) - \eta_{x2} S_0^0\left(\frac{\partial u^2}{\partial \xi^0}\right) + S_0^0\left(\frac{\partial u^0}{\partial \xi^0}\right)\right\}$$

Again replacing the third and fourth terms by Eq. (43) and (44) results in

$$Q_2^\alpha = (\eta_{x1}^2 + \eta_{x2}^2)\left\{S_1^1\left(\frac{\partial u^1}{\partial \xi^1}\right) + S_2^2\left(\frac{\partial u^2}{\partial \xi^2}\right) + S_1^0\left(\frac{\partial u^1}{\partial \xi^0}\right) + S_2^0\left(\frac{\partial u^2}{\partial \xi^0}\right) + S_0^0\left(\frac{\partial u^0}{\partial \xi^0}\right)\right\}$$

Obviously, the terms in the bracket are exactly equal to the continuity equation expressed as Eq. (36), which leads in zero:

$$Q_2^\alpha = (\eta_{x1}^2 + \eta_{x2}^2)\{0\} = 0 \tag{59}$$

In consequence,

$$M^\alpha = \frac{2\mu^\alpha}{1 + \eta_{x1}^2 + \eta_{x2}^2} Q_1^\alpha \tag{60}$$

or

$$\begin{aligned} M^\alpha = & \frac{2\mu^\alpha}{1 + \eta_{x1}^2 + \eta_{x2}^2} \left\{ -(\eta_{x2}^2 + 1) \left(S_1^1 \frac{\partial u^1}{\partial \xi^1} \right) - (\eta_{x1}^2 + 1) \left(S_2^2 \frac{\partial u^2}{\partial \xi^2} \right) \right. \\ & \left. -\eta_{x1} \left(S_1^1 \frac{\partial u^0}{\partial \xi^1} \right) - \eta_{x2} \left(S_2^2 \frac{\partial u^0}{\partial \xi^2} \right) + \eta_{x1}\eta_{x2} \left(S_2^2 \frac{\partial u^1}{\partial \xi^2} + S_1^1 \frac{\partial u^2}{\partial \xi^1} \right) \right\} \end{aligned} \tag{61}$$

A.6 Derivation of curvilinear $\frac{\partial u^1}{\partial \xi^0}$ and $\frac{\partial u^2}{\partial \xi^0}$ from surface tangential stress

Our goal is to derive $\frac{\partial u^1}{\partial \xi^0}$ and $\frac{\partial u^2}{\partial \xi^0}$ at the surface from σ_{t1}^α and σ_{t2}^α. An important trick will be imposed in the procedure of derivations is to eliminate $\frac{\partial}{\partial \xi^0}$ terms on the right hand side, since for pseudospectral method evaluating lumped vertical and horizontal terms will involve in mixed numerical schemes, i.e., finite difference and Fourier differentiation. If we keep only horizontal terms on right hand side to represent the surface conditions for $\frac{\partial u^1}{\partial \xi^0}$ and $\frac{\partial u^2}{\partial \xi^0}$, the evaluations merely involve in spectral methods and result in higher accuracy.

First of all, after expanding Eq. (26) and Eq. (27) in curvilinear coordinates

using Eq. (32) to Eq. (34), we have

$$\begin{aligned}
\frac{G_1}{\mu^\alpha}\sigma_{t1}^\alpha &= 2\eta_{x1}\left(\frac{\partial u^0}{\partial x^0}-\frac{\partial u^1}{\partial x^1}\right)-\eta_{x2}\left(\frac{\partial u^1}{\partial x^2}+\frac{\partial u^2}{\partial x^1}\right)\\
&\quad+(1-\eta_{x1}^2)\left(\frac{\partial u^0}{\partial x^1}+\frac{\partial u^1}{\partial x^0}\right)-\eta_{x1}\eta_{x2}\left(\frac{\partial u^0}{\partial x^2}+\frac{\partial u^2}{\partial x^0}\right)\\
&= 2\eta_{x1}\left(S_0^0\frac{\partial u^0}{\partial \xi^0}-S_1^0\frac{\partial u^1}{\partial \xi^0}-S_1^1\frac{\partial u^1}{\partial \xi^1}\right)\\
&\quad-\eta_{x2}\left(S_2^0\frac{\partial u^1}{\partial \xi^0}+S_2^2\frac{\partial u^1}{\partial \xi^2}+S_1^0\frac{\partial u^2}{\partial \xi^0}+S_1^1\frac{\partial u^2}{\partial \xi^1}\right)\\
&\quad+(1-\eta_{x1}^2)\left(S_1^0\frac{\partial u^0}{\partial \xi^0}+S_1^1\frac{\partial u^0}{\partial \xi^1}+S_0^0\frac{\partial u^1}{\partial \xi^0}\right)\\
&\quad-\eta_{x1}\eta_{x2}\left(S_2^0\frac{\partial u^0}{\partial \xi^0}+S_2^2\frac{\partial u^0}{\partial \xi^2}+S_0^0\frac{\partial u^2}{\partial \xi^0}\right)
\end{aligned}\tag{62}$$

$$\begin{aligned}
\frac{G_2}{\mu^\alpha}\sigma_{t2}^\alpha &= 2\eta_{x2}\left(\frac{\partial u^0}{\partial x^0}-\frac{\partial u^2}{\partial x^2}\right)-\eta_{x1}\left(\frac{\partial u^2}{\partial x^1}+\frac{\partial u^1}{\partial x^2}\right)\\
&\quad+(1-\eta_{x2}^2)\left(\frac{\partial u^0}{\partial x^2}+\frac{\partial u^2}{\partial x^0}\right)-\eta_{x1}\eta_{x2}\left(\frac{\partial u^0}{\partial x^1}+\frac{\partial u^1}{\partial x^0}\right)\\
&= 2\eta_{x2}\left(S_0^0\frac{\partial u^0}{\partial \xi^0}-S_2^0\frac{\partial u^2}{\partial \xi^0}-S_2^2\frac{\partial u^2}{\partial \xi^2}\right)\\
&\quad-\eta_{x1}\left(S_2^0\frac{\partial u^1}{\partial \xi^0}+S_2^2\frac{\partial u^1}{\partial \xi^2}+S_1^0\frac{\partial u^2}{\partial \xi^0}+S_1^1\frac{\partial u^2}{\partial \xi^1}\right)\\
&\quad+(1-\eta_{x2}^2)\left(S_2^0\frac{\partial u^0}{\partial \xi^0}+S_2^2\frac{\partial u^0}{\partial \xi^2}+S_0^0\frac{\partial u^2}{\partial \xi^0}\right)\\
&\quad-\eta_{x1}\eta_{x2}\left(S_1^0\frac{\partial u^0}{\partial \xi^0}+S_1^1\frac{\partial u^0}{\partial \xi^1}+S_0^0\frac{\partial u^1}{\partial \xi^0}\right)
\end{aligned}\tag{63}$$

After term by term rearranging, they can be rewritten as below. Note that the final coefficients of E_0^0, E_0^1, E_0^2, F_0^0, F_0^1, and F_0^2 are derived by using Eq. (43) and Eq. (44).

$$\frac{G_1}{\mu^\alpha}\sigma^\alpha_{t1} = E^0_0\frac{\partial u^0}{\partial \xi^0} + E^0_1\frac{\partial u^0}{\partial \xi^1} + E^0_2\frac{\partial u^0}{\partial \xi^2} + E^1_0\frac{\partial u^1}{\partial \xi^0} + E^1_1\frac{\partial u^1}{\partial \xi^1} + E^1_2\frac{\partial u^1}{\partial \xi^2} + \cancelto{0}{E^2_0\frac{\partial u^2}{\partial \xi^0}} + E^2_1\frac{\partial u^2}{\partial \xi^1} \tag{64}$$

$$\begin{aligned}
E^0_0 &= 2\eta_{x1}S^0_0 + (1-\eta^2_{x1})S^0_1 - \eta_{x1}\eta_{x2}S^0_2 & E^0_1 &= (1-\eta^2_{x1})S^1_1\\
&= 2\eta_{x1}S^0_0 - \eta_{x1}(1-\eta^2_{x1})S^0_0 + \eta_{x1}\eta^2_{x2}S^0_0 & E^1_1 &= -2\eta_{x1}S^1_1\\
&= S^0_0\eta_{x1}(1+\eta^2_{x1}+\eta^2_{x2}) & E^2_1 &= -\eta_{x2}S^1_1\\
E^1_0 &= -2\eta_{x1}S^0_1 - \eta_{x2}S^0_2 + (1-\eta^2_{x1})S^0_0 & E^0_2 &= -\eta_{x1}\eta_{x2}S^2_2\\
&= 2\eta^2_{x1}S^0_0 + \eta^2_{x2}S^0_0 + (1-\eta^2_{x1})S^0_0 & E^1_2 &= -\eta_{x2}S^2_2\\
&= S^0_0(\eta^2_{x1}+\eta^2_{x2}+1)\\
E^2_0 &= -\eta_{x2}S^0_1 - \eta_{x1}\eta_{x2}S^0_0\\
&= \eta_{x1}\eta_{x2}S^0_0 - \eta_{x1}\eta_{x2}S^0_0\\
&= 0
\end{aligned}$$

$$\frac{G_2}{\mu^\alpha}\sigma^\alpha_{t2} = F^0_0\frac{\partial u^0}{\partial \xi^0} + F^0_1\frac{\partial u^0}{\partial \xi^1} + F^0_2\frac{\partial u^0}{\partial \xi^2} + \cancelto{0}{F^1_0\frac{\partial u^1}{\partial \xi^0}} + F^1_2\frac{\partial u^1}{\partial \xi^2} + F^2_0\frac{\partial u^2}{\partial \xi^0} + F^2_1\frac{\partial u^2}{\partial \xi^1} + F^2_2\frac{\partial u^2}{\partial \xi^2} \tag{65}$$

$$\begin{aligned}
F^0_0 &= 2\eta_{x2}S^0_0 + (1-\eta^2_{x2})S^0_2 - \eta_{x1}\eta_{x2}S^0_2 & F^0_1 &= -\eta_{x1}\eta_{x2}S^1_1\\
&= 2\eta_{x2}S^0_0 - \eta_{x2}(1-\eta^2_{x2})S^0_0 + \eta^2_{x1}\eta_{x2}S^0_0 & F^2_1 &= -\eta_{x1}S^1_1\\
&= S^0_0\eta_{x2}(1+\eta^2_{x1}+\eta^2_{x2}) & F^0_2 &= (1-\eta^2_{x2})S^2_2\\
F^1_0 &= -\eta_{x1}S^0_2 - \eta_{x1}\eta_{x2}S^0_0 & F^1_2 &= -\eta_{x1}S^2_2\\
&= \eta_{x1}\eta_{x2}S^0_0 - \eta_{x1}\eta_{x2}S^0_0 & F^2_2 &= -2\eta_{x2}S^2_2\\
&= 0\\
F^2_0 &= -2\eta_{x2}S^0_2 - \eta_{x1}S^0_1 + (1-\eta^2_{x2})S^0_0\\
&= 2\eta^2_{x2}S^0_0 + \eta^2_{x1}S^0_0 + (1-\eta^2_{x2})S^0_0\\
&= S^0_0(1+\eta^2_{x1}+\eta^2_{x2})
\end{aligned}$$

To eliminate $\frac{\partial u^0}{\partial \xi^0}$, we replace it by using continuity equation Eq. (36):

$$S_0^0 \frac{\partial u^0}{\partial \xi^0} = -S_1^0 \frac{\partial u^1}{\partial \xi^0} - S_1^1 \frac{\partial u^1}{\partial \xi^1} - S_2^0 \frac{\partial u^2}{\partial \xi^0} - S_2^2 \frac{\partial u^2}{\partial \xi^2} \tag{66}$$

Substituting into $E_0^0 \frac{\partial u^0}{\partial \xi^0}$ of Eq. (64) results in

$$E_0^0 \frac{\partial u^0}{\partial \xi^0} = \eta_{x1}(1 + \eta_{x1}^2 + \eta_{x2}^2) \left[-S_1^0 \frac{\partial u^1}{\partial \xi^0} - S_1^1 \frac{\partial u^1}{\partial \xi^1} - S_2^0 \frac{\partial u^2}{\partial \xi^0} - S_2^2 \frac{\partial u^2}{\partial \xi^2} \right] \tag{67}$$

Again, after term by term rearranging Eq. (64), we have

$$\begin{aligned}
\frac{G_1}{\mu^\alpha}\sigma_{t1}^\alpha = & [(1-\eta_{x1}^2)S_1^1]\frac{\partial u^0}{\partial \xi^1} \\
& + [-\eta_{x1}\eta_{x2}S_2^2]\frac{\partial u^0}{\partial \xi^2} \\
& + [S_0^0(1+\eta_{x1}^2+\eta_{x2}^2)\underbrace{-\eta_{x1}S_1^0(1+\eta_{x1}^2+\eta_{x2}^2)}_{S_1^0=-\eta_{x1}S_0^0 \text{ (Eq. 43)}}]\frac{\partial u^1}{\partial \xi^0} \\
& + [-2\eta_{x1}S_1^1\underbrace{-\eta_{x1}S_1^1(1+\eta_{x1}^2+\eta_{x2}^2)}]\frac{\partial u^1}{\partial \xi^1} \\
& + [-\eta_{x2}S_2^2]\frac{\partial u^1}{\partial \xi^2} \\
& + [\underbrace{-\eta_{x1}S_2^0(1+\eta_{x1}^2+\eta_{x2}^2)}_{S_2^0=-\eta_{x2}S_0^0 \text{ (Eq. 44)}}]\frac{\partial u^2}{\partial \xi^0} \\
& + [-\eta_{x2}S_1^1]\frac{\partial u^2}{\partial \xi^1} \\
& + [\underbrace{-\eta_{x1}S_2^2(1+\eta_{x1}^2+\eta_{x2}^2)}]\frac{\partial u^2}{\partial \xi^2}
\end{aligned}$$

Note that the underbrace terms come from the continuity equation. Simplified equa-

tion reads

$$\begin{aligned}
\frac{G_1}{\mu^\alpha}\sigma_{t1}^\alpha = &[(1-\eta_{x1}^2)S_1^1]\frac{\partial u^0}{\partial \xi^1} \\
&+[-\eta_{x1}\eta_{x2}S_2^2]\frac{\partial u^0}{\partial \xi^2} \\
&+[S_0^0(1+\eta_{x1}^2+\eta_{x2}^2)(1+\eta_{x1}^2)]\boldsymbol{\frac{\partial u^1}{\partial \xi^0}} \\
&+[S_1^1\eta_{x1}(-3-\eta_{x1}^2-\eta_{x2}^2)]\frac{\partial u^1}{\partial \xi^1} \\
&+[-\eta_{x2}S_2^2]\frac{\partial u^1}{\partial \xi^2} \\
&+[\eta_{x1}\eta_{x2}S_0^0(1+\eta_{x1}^2+\eta_{x2}^2)]\boldsymbol{\frac{\partial u^2}{\partial \xi^0}} \\
&+[-\eta_{x2}S_1^1]\frac{\partial u^2}{\partial \xi^1} \\
&+[-\eta_{x1}S_2^2(1+\eta_{x1}^2+\eta_{x2}^2)]\frac{\partial u^2}{\partial \xi^2}
\end{aligned} \tag{68}$$

Bold terms are still regarding vertical differentiation $\frac{\partial}{\partial \xi^0}$. If we want to employ Eq. (68) as the surface condition of $\frac{\partial u^1}{\partial \xi^0}$, we need to further eliminate $\frac{\partial u^2}{\partial \xi^0}$ to make sure no vertical differentiation needs to be evaluated.

Similarly, replacing $F_0^0\frac{\partial u^0}{\partial \xi^0}$ of Eq. (65) with Eq. (66) results in

$$F_0^0\frac{\partial u^0}{\partial \xi^0} = \eta_{x2}(1+\eta_{x1}^2+\eta_{x2}^2)\left[-S_1^0\frac{\partial u^1}{\partial \xi^0}-S_1^1\frac{\partial u^1}{\partial \xi^1}-S_2^0\frac{\partial u^2}{\partial \xi^0}-S_2^2\frac{\partial u^2}{\partial \xi^2}\right] \tag{69}$$

After term by term rearranging Eq. (65), we have

$$
\begin{aligned}
\frac{G_2}{\mu^\alpha}\sigma^\alpha_{t2} = & [-\eta_{x1}\eta_{x2}S_1^1]\frac{\partial u^0}{\partial \xi^1} \\
& + [(1-\eta_{x2}^2)S_2^2]\frac{\partial u^0}{\partial \xi^2} \\
& + [\underbrace{-\eta_{x2}S_1^0(1+\eta_{x1}^2+\eta_{x2}^2)}_{S_1^0=-\eta_{x1}S_0^0 \text{ (Eq. 43)}}]\frac{\partial u^1}{\partial \xi^0} \\
& + [\underbrace{-\eta_{x2}S_1^1(1+\eta_{x1}^2+\eta_{x2}^2)}]\frac{\partial u^1}{\partial \xi^1} \\
& + [-\eta_{x1}S_2^2]\frac{\partial u^1}{\partial \xi^2} \\
& + [S_0^0(1+\eta_{x1}^2+\eta_{x2}^2)\underbrace{-\eta_{x2}S_2^0(1+\eta_{x1}^2+\eta_{x2}^2)}_{S_2^0=-\eta_{x2}S_0^0 \text{ (Eq. 44)}}]\frac{\partial u^2}{\partial \xi^0} \\
& + [-\eta_{x1}S_1^1]\frac{\partial u^2}{\partial \xi^1} \\
& + [-2\eta_{x2}S_2^2\underbrace{-\eta_{x2}S_2^2(1+\eta_{x1}^2+\eta_{x2}^2)}]\frac{\partial u^2}{\partial \xi^2}
\end{aligned}
$$

As the same, the underbrace terms come from the continuity equation. Simplified equation becomes

$$\begin{aligned}\frac{G_2}{\mu^\alpha}\sigma^\alpha_{t2} = &[-\eta_{x1}\eta_{x2}S^1_1]\frac{\partial u^0}{\partial \xi^1}\\ &+[(1-\eta^2_{x2})S^2_2]\frac{\partial u^0}{\partial \xi^2}\\ &+[\eta_{x1}\eta_{x2}S^0_0(1+\eta^2_{x1}+\eta^2_{x2})]\frac{\boldsymbol{\partial u^1}}{\boldsymbol{\partial \xi^0}}\\ &+[-\eta_{x2}S^1_1(1+\eta^2_{x1}+\eta^2_{x2})]\frac{\partial u^1}{\partial \xi^1}\\ &+[-\eta_{x1}S^2_2]\frac{\partial u^1}{\partial \xi^2}\\ &+[S^0_0(1+\eta^2_{x1}+\eta^2_{x2})(1+\eta^2_{x2})]\frac{\boldsymbol{\partial u^2}}{\boldsymbol{\partial \xi^0}}\\ &+[-\eta_{x1}S^1_1]\frac{\partial u^2}{\partial \xi^1}\\ &+[S^2_2\eta_{x2}(-3-\eta^2_{x1}-\eta^2_{x2})]\frac{\partial u^2}{\partial \xi^2}\end{aligned} \tag{70}$$

Similar to Eq. (68), bold terms are regarding vertical differentiation $\frac{\partial}{\partial \xi^0}$. If we want to employ Eq. (70) as the surface condition of $\frac{\partial u^2}{\partial \xi^0}$, we need to further eliminate $\frac{\partial u^1}{\partial \xi^0}$ to make sure no vertical differentiation needs to be evaluated.

To cancel out $\frac{\partial u^2}{\partial \xi^0}$ to keep only $\frac{\partial u^1}{\partial \xi^0}$ in Eq.(68), and cancel out $\frac{\partial u^1}{\partial \xi^0}$ to keep only $\frac{\partial u^2}{\partial \xi^0}$ in Eq. (70), we can apply the operations:

$$\text{eliminate } \frac{\partial u^2}{\partial \xi^0} \text{ in (68)} \Rightarrow (1+\eta^2_{x2})\times(68)+(-\eta_{x1}\eta_{x2})\times(70) \tag{71}$$

$$\text{eliminate } \frac{\partial u^1}{\partial \xi^0} \text{ in (70)} \Rightarrow (-\eta_{x1}\eta_{x2})\times(68)+(1+\eta^2_{x1})\times(70) \tag{72}$$

The results of Eq. (71) can be written in

$$C_0^1\frac{\partial u^1}{\partial \xi^0} = -C_1^0\frac{\partial u^0}{\partial \xi^1} - C_2^0\frac{\partial u^0}{\partial \xi^2} - C_1^1\frac{\partial u^1}{\partial \xi^1} - C_2^1\frac{\partial u^1}{\partial \xi^2} - C_1^2\frac{\partial u^2}{\partial \xi^1} - C_2^2\frac{\partial u^2}{\partial \xi^2} + \left[(1+\eta_{x2}^2)\frac{G_1}{\mu^\alpha}\sigma_{t1}^\alpha - \eta_{x1}\eta_{x2}\frac{G_2}{\mu^\alpha}\sigma_{t2}^\alpha\right] \tag{73}$$

where

$$\begin{aligned}
C_1^0 &= (1-\eta_{x1}^2)S_1^1(1+\eta_{x2}^2) + \eta_{x1}^2\eta_{x2}^2 S_1^1 && = S_1^1(1+\eta_{x2}^2-\eta_{x1}^2)\\
C_2^0 &= -\eta_{x1}\eta_{x2}(1+\eta_{x2}^2)S_2^2 - \eta_{x1}\eta_{x2}(1-\eta_{x2}^2)S_2^2 && = -2\eta_{x1}\eta_{x2}S_2^2\\
C_1^1 &= S_1^1\eta_{x1}(1+\eta_{x2}^2)(-3-\eta_{x1}^2-\eta_{x2}^2) + \eta_{x1}\eta_{x2}^2 S_1^1(1+\eta_{x1}^2+\eta_{x2}^2) && = -S_1^1\eta_{x1}(3+\eta_{x1}^2+3\eta_{x2}^2)\\
C_2^1 &= -\eta_{x2}S_2^2(1+\eta_{x2}^2) + \eta_{x1}^2\eta_{x2}S_2^2 && = -\eta_{x2}S_2^2(1+\eta_{x2}^2-\eta_{x1}^2)\\
C_0^1 &= S_0^0(1+\eta_{x1}^2+\eta_{x2}^2)(1+\eta_{x1}^2)(1+\eta_{x2}^2) - \eta_{x1}^2\eta_{x2}^2 S_0^0(1+\eta_{x1}^2+\eta_{x2}^2) && = S_0^0(1+\eta_{x1}^2+\eta_{x2}^2)^2\\
C_1^2 &= -\eta_{x2}S_1^1(1+\eta_{x2}^2) + \eta_{x1}^2\eta_{x2}S_1^1 && = -S_1^1\eta_{x2}(1+\eta_{x2}^2-\eta_{x1}^2)\\
C_2^2 &= -\eta_{x1}S_2^2(1+\eta_{x1}^2+\eta_{x2}^2)(1+\eta_{x2}^2) - S_2^2\eta_{x1}\eta_{x2}^2(-3-\eta_{x1}^2-\eta_{x2}^2) && = -\eta_{x1}S_2^2(1+\eta_{x1}^2-\eta_{x2}^2)
\end{aligned}$$

After rearranging, the final form can be written as

$$\frac{\partial u^1}{\partial \xi^0} = C_0\left\{C_1\frac{\partial u^0}{\partial \xi^1} + C_2\frac{\partial u^0}{\partial \xi^2} + C_3\frac{\partial u^1}{\partial \xi^1} + C_4\frac{\partial u^1}{\partial \xi^2} + C_5\frac{\partial u^2}{\partial \xi^1} + C_6\frac{\partial u^2}{\partial \xi^2} + C_7\frac{\sigma_{t1}^\alpha}{\mu^\alpha} + C_8\frac{\sigma_{t2}^\alpha}{\mu^\alpha}\right\} \tag{74}$$

where

$$C_0 = [S_0^0(G_0)^2]^{-1} \qquad C_5 = S_1^1 A \qquad A = \eta_{x2}(1+\eta_{x2}^2-\eta_{x1}^2)$$
$$C_1 = S_1^1(\eta_{x1}^2-\eta_{x2}^2-1) \qquad C_6 = S_2^2\eta_{x1}(1+\eta_{x1}^2-\eta_{x2}^2) \qquad G_0 = (1+\eta_{x1}^2+\eta_{x2}^2)$$
$$C_2 = 2\eta_{x1}\eta_{x2}S_2^2 \qquad C_7 = (1+\eta_{x2}^2)G_1 \qquad G_1 = \sqrt{(1+\eta_{x1}^2)G_0}$$
$$C_3 = S_1^1\eta_{x1}(3+\eta_{x1}^2+3\eta_{x2}^2) \qquad C_8 = -\eta_{x1}\eta_{x2}G_2 \qquad G_2 = \sqrt{(1+\eta_{x2}^2)G_0}$$
$$C_4 = S_2^2 A$$

Similarly, the results of Eq. (72) can be written in

$$\begin{aligned} D_0^2\frac{\partial u^2}{\partial \xi^0} = &- D_1^0\frac{\partial u^0}{\partial \xi^1} - D_2^0\frac{\partial u^0}{\partial \xi^2} - D_1^1\frac{\partial u^1}{\partial \xi^1} - D_2^1\frac{\partial u^1}{\partial \xi^2} - D_1^2\frac{\partial u^2}{\partial \xi^1} - D_2^2\frac{\partial u^2}{\partial \xi^2} \\ &+ \left[(-\eta_{x1}\eta_{x2})\frac{G_1}{\mu^\alpha}\sigma_{t1}^\alpha + (1+\eta_{x2}^2)\frac{G_2}{\mu^\alpha}\sigma_{t2}^\alpha\right] \end{aligned} \tag{75}$$

where

$$D_1^0 = -\eta_{x1}\eta_{x2}(1-\eta_{x1}^2)S_1^1 - \eta_{x1}\eta_{x2}(1+\eta_{x1}^2)S_1^1 = -2\eta_{x1}\eta_{x2}S_1^1$$
$$D_2^0 = \eta_{x1}^2\eta_{x2}^2S_2^2 + (1-\eta_{x2}^2)S_2^2(1+\eta_{x1}^2) = -S_2^2(\eta_{x2}^2-\eta_{x1}^2-1)$$
$$D_1^1 = -\eta_{x1}^2\eta_{x2}S_1^1(-3-\eta_{x1}^2-\eta_{x2}^2) - S_1^1\eta_{x2}(1+\eta_{x1}^2)(1+\eta_{x1}^2+\eta_{x2}^2) = -S_1^1\eta_{x2}(1+\eta_{x2}^2-\eta_{x1}^2)$$
$$D_2^1 = \eta_{x1}\eta_{x2}^2S_2^2 - \eta_{x1}S_2^2(1+\eta_{x1}^2) = -S_2^2\eta_{x1}(1+\eta_{x1}^2-\eta_{x2}^2)$$
$$D_0^2 = -\eta_{x1}^2\eta_{x2}^2S_0^0(1+\eta_{x1}^2+\eta_{x2}^2) + S_0^0(1+\eta_{x1}^2)(1+\eta_{x2}^2)(1+\eta_{x1}^2+\eta_{x2}^2) = S_0^0(1+\eta_{x1}^2+\eta_{x2}^2)^2$$
$$D_1^2 = \eta_{x1}\eta_{x2}^2S_1^1 - \eta_{x1}^2S_1^1(1+\eta_{x1}^2) = -S_1^1\eta_{x1}(1+\eta_{x1}^2-\eta_{x2}^2)$$
$$D_2^2 = \eta_{x1}^2\eta_{x2}S_2^2(1+\eta_{x1}^2+\eta_{x2}^2) + S_2^2\eta_{x2}(1+\eta_{x1}^2)(-3-\eta_{x1}^2-\eta_{x2}^2) = -S_2^2\eta_{x2}(3+\eta_{x2}^2+3\eta_{x1}^2)$$

After rearranging, the final form can be written as

$$\frac{\partial u^2}{\partial \xi^0} = D_0 \left\{ D_1 \frac{\partial u^0}{\partial \xi^1} + D_2 \frac{\partial u^0}{\partial \xi^2} + D_3 \frac{\partial u^1}{\partial \xi^1} + D_4 \frac{\partial u^1}{\partial \xi^2} + D_5 \frac{\partial u^2}{\partial \xi^1} + D_6 \frac{\partial u^2}{\partial \xi^2} + D_7 \frac{\sigma^\alpha_{t1}}{\mu^\alpha} + D_8 \frac{\sigma^\alpha_{t2}}{\mu^\alpha} \right\} \tag{76}$$

where

$$D_0 = [S_0^0 (G_0)^2]^{-1} = C_0 \qquad D_5 = S_1^1 B \qquad B = \eta_{x1}(1 + \eta_{x1}^2 - \eta_{x2}^2)$$

$$D_1 = 2\eta_{x1}\eta_{x2} S_1^1 \qquad D_6 = S_2^2 \eta_{x2}(3 + 3\eta_{x1}^2 + \eta_{x2}^2) \qquad G_0 = (1 + \eta_{x1}^2 + \eta_{x2}^2)$$

$$D_2 = S_2^2(\eta_{x2}^2 - \eta_{x1}^2 - 1) \qquad D_7 = -\eta_{x1}\eta_{x2} G_1 \qquad G_1 = \sqrt{(1 + \eta_{x1}^2) G_0}$$

$$D_3 = S_1^1 \eta_{x2}(1 + \eta_{x2}^2 - \eta_{x1}^2) \qquad D_8 = (1 + \eta_{x1}^2) G_2 \qquad G_2 = \sqrt{(1 + \eta_{x2}^2) G_0}$$

$$D_4 = S_2^2 B$$

A.7 Derivation of curvilinear ∇^2 operator

By using Eq. (32) to (34), expanding each term of

$$\nabla^2 = \frac{\partial^2}{\partial x^0 \partial x^0} + \frac{\partial^2}{\partial x^1 \partial x^1} + \frac{\partial^2}{\partial x^2 \partial x^2} \tag{77}$$

results in

$$\begin{aligned} \frac{\partial^2}{\partial x^0 \partial x^0} &= S_0^0 \frac{\partial}{\partial \xi^0} \left(S_0^0 \frac{\partial}{\partial \xi^0} \right) \\ &= S_0^0 \frac{\partial S_0^0}{\partial \xi^0} \frac{\partial}{\partial \xi^0} + \left(S_0^0 \right)^2 \frac{\partial^2}{\partial \xi^0 \partial \xi^0} \end{aligned} \tag{78}$$

$$
\begin{aligned}
\frac{\partial^2}{\partial x^1 \partial x^1} &= \left(S_1^0 \frac{\partial}{\partial \xi^0} + S_1^1 \frac{\partial}{\partial \xi^1} \right) \left(S_1^0 \frac{\partial}{\partial \xi^0} + S_1^1 \frac{\partial}{\partial \xi^1} \right) \\
&= S_1^0 \frac{\partial S_1^0}{\partial \xi^0} \frac{\partial}{\partial \xi^0} + \left(S_1^0 \right)^2 \frac{\partial^2}{\partial \xi^0 \partial \xi^0} + S_1^0 \frac{\partial S_1^1}{\partial \xi^0} \frac{\partial}{\partial \xi^1} + S_1^0 S_1^1 \frac{\partial^2}{\partial \xi^0 \partial \xi^1} \\
&+ S_1^1 \frac{\partial S_1^0}{\partial \xi^1} \frac{\partial}{\partial \xi^0} + S_1^1 S_1^0 \frac{\partial^2}{\partial \xi^0 \partial \xi^1} + S_1^1 \frac{\partial S_1^1}{\partial \xi^1} \frac{\partial}{\partial \xi^1} + \left(S_1^1 \right)^2 \frac{\partial^2}{\partial \xi^1 \partial \xi^1} \\
&= \left(S_1^0 \frac{\partial S_1^0}{\partial \xi^0} + S_1^1 \frac{\partial S_1^0}{\partial \xi^1} \right) \frac{\partial}{\partial \xi^0} + \left(S_1^0 \frac{\partial S_1^1}{\partial \xi^0} + S_1^1 \frac{\partial S_1^1}{\partial \xi^1} \right) \frac{\partial}{\partial \xi^1} \\
&+ 2 S_1^0 S_1^1 \frac{\partial^2}{\partial \xi^0 \partial \xi^1} + \left(S_1^0 \right)^2 \frac{\partial^2}{\partial \xi^0 \partial \xi^0} + \left(S_1^1 \right)^2 \frac{\partial^2}{\partial \xi^1 \partial \xi^1}
\end{aligned} \tag{79}
$$

$$
\begin{aligned}
\frac{\partial^2}{\partial x^2 \partial x^2} &= \left(S_2^0 \frac{\partial}{\partial \xi^0} + S_2^2 \frac{\partial}{\partial \xi^2} \right) \left(S_2^0 \frac{\partial}{\partial \xi^0} + S_2^2 \frac{\partial}{\partial \xi^2} \right) \\
&= S_2^0 \frac{\partial S_2^0}{\partial \xi^0} \frac{\partial}{\partial \xi^0} + \left(S_2^0 \right)^2 \frac{\partial^2}{\partial \xi^0 \partial \xi^0} + S_2^0 \frac{\partial S_2^2}{\partial \xi^0} \frac{\partial}{\partial \xi^2} + S_2^0 S_2^2 \frac{\partial^2}{\partial \xi^0 \partial \xi^2} \\
&+ S_2^2 \frac{\partial S_2^0}{\partial \xi^2} \frac{\partial}{\partial \xi^0} + S_2^0 S_2^2 \frac{\partial^2}{\partial \xi^2 \partial \xi^0} + S_2^2 \frac{\partial S_2^2}{\partial \xi^2} \frac{\partial}{\partial \xi^2} + \left(S_2^2 \right)^2 \frac{\partial^2}{\partial \xi^2 \partial \xi^2} \\
&= \left(S_2^0 \frac{\partial S_2^0}{\partial \xi^0} + S_2^2 \frac{\partial S_2^0}{\partial \xi^2} \right) \frac{\partial}{\partial \xi^0} + \left(S_2^0 \frac{\partial S_2^2}{\partial \xi^0} + S_2^2 \frac{\partial S_2^2}{\partial \xi^2} \right) \frac{\partial}{\partial \xi^2} \\
&+ 2 S_2^0 S_2^2 \frac{\partial^2}{\partial \xi^0 \partial \xi^2} + \left(S_2^0 \right)^2 \frac{\partial^2}{\partial \xi^0 \partial \xi^0} + \left(S_2^2 \right)^2 \frac{\partial^2}{\partial \xi^2 \partial \xi^2}
\end{aligned} \tag{80}
$$

Summing three terms together turns out to be

$$
\begin{aligned}
\nabla^2 &= \left[\left(S_0^0 \right)^2 + \left(S_1^1 \right)^2 + \left(S_2^2 \right)^2 \right] \frac{\partial^2}{\partial \xi^0 \partial \xi^0} + \left(S_1^1 \right)^2 \frac{\partial^2}{\partial \xi^1 \partial \xi^1} + \left(S_2^2 \right)^2 \frac{\partial^2}{\partial \xi^2 \partial \xi^2} \\
&+ S_1^0 S_1^1 \frac{\partial^2}{\partial \xi^0 \partial \xi^1} + S_2^0 S_2^2 \frac{\partial^2}{\partial \xi^0 \partial \xi^2} \\
&+ \left[S_0^0 \frac{\partial S_0^0}{\partial \xi^0} + S_1^0 \frac{\partial S_1^0}{\partial \xi^0} + S_1^1 \frac{\partial S_1^0}{\partial \xi^1} + S_2^0 \frac{\partial S_2^0}{\partial \xi^0} + S_2^2 \frac{\partial S_2^0}{\partial \xi^2} \right] \frac{\partial}{\partial \xi^0} \\
&+ \underbrace{\left[S_1^0 \frac{\partial S_1^1}{\partial \xi^0} + S_1^1 \frac{\partial S_1^1}{\partial \xi^1} \right]}_{=0 \text{ due to Eq. (37)}} \frac{\partial}{\partial \xi^1} + \underbrace{\left[S_2^0 \frac{\partial S_2^2}{\partial \xi^0} + S_2^2 \frac{\partial S_2^2}{\partial \xi^2} \right]}_{=0 \text{ due to Eq. (38)}} \frac{\partial}{\partial \xi^2}
\end{aligned} \tag{81}
$$

On the other hand, given contravariant metric tensor g^{ij} as

$$g^{ij} = \nabla\xi^i \cdot \nabla\xi^j = \frac{\partial \xi^i}{\partial x^k}\frac{\partial \xi^j}{\partial x^k} = S^i_k S^j_k \tag{82}$$

by using Eq. (31), expanding g^{ij} results in

$$\begin{aligned}
g^{00} &= S^0_0 S^0_0 + S^0_1 S^0_1 + S^0_2 S^0_2 = \left(S^0_0\right)^2 + \left(S^0_1\right)^2 + \left(S^0_2\right)^2 \\
g^{11} &= \cancel{S^1_0 S^1_0} + S^1_1 S^1_1 + \cancel{S^1_2 S^1_2} = \left(S^1_1\right)^2 \\
g^{22} &= \cancel{S^2_0 S^2_0} + \cancel{S^2_1 S^2_1} + S^2_2 S^2_2 = \left(S^2_2\right)^2 \\
g^{01} &= \cancel{S^0_0 S^1_0} + S^0_1 S^1_1 + \cancel{S^0_2 S^1_2} = S^0_1 S^1_1 \\
g^{02} &= \cancel{S^0_0 S^2_0} + \cancel{S^0_1 S^2_1} + S^0_2 S^2_2 = S^0_2 S^2_2 \\
g^{12} &= \cancel{S^1_0 S^2_0} + \cancel{S^1_1 S^2_1} + \cancel{S^1_2 S^2_2} = 0
\end{aligned}$$

Substituting these results into Eq. (81) leads in

$$\nabla^2 = g^{00}\frac{\partial^2}{\partial\xi^0\partial\xi^0} + g^{11}\frac{\partial^2}{\partial\xi^1\partial\xi^1} + g^{22}\frac{\partial^2}{\partial\xi^2\partial\xi^2} + 2g^{01}\frac{\partial^2}{\partial\xi^0\partial\xi^1} + 2g^{02}\frac{\partial^2}{\partial\xi^0\partial\xi^2} + C_0\frac{\partial}{\partial\xi^0} \tag{83}$$

where

$$C_0 = \left[S^0_0\frac{\partial S^0_0}{\partial\xi^0} + S^0_1\frac{\partial S^0_1}{\partial\xi^0} + S^1_1\frac{\partial S^0_1}{\partial\xi^1} + S^0_2\frac{\partial S^0_2}{\partial\xi^0} + S^2_2\frac{\partial S^0_2}{\partial\xi^2}\right] \tag{84}$$

Considering pseudospectral method with periodic horizontal dimensions, Fourier basis is applied to ξ^1 and ξ^2 axes. Given that S^1_1 and S^2_2 are constants, in this case only g^{11} and g^{22} are constants, while the other g^{ij} terms are non-constants. Applying Fourier transform to any term with non-constant coefficient will result in non-linear

convolutions even if it is linear term, which implies that such type of term can only be evaluated explicitly and can not be the implicit linear operators on the left hand side of equations. On the basis of mathematical restrictions, in Eq. (83) only $g^{11}\frac{\partial^2}{\partial\xi^1\partial\xi^1}$ and $g^{22}\frac{\partial^2}{\partial\xi^2\partial\xi^2}$ can be solved implicitly. To make the solution constrained with the $\frac{\partial^2}{\partial\xi^0\partial\xi^0}$ term, an additional constant H_c is introduced to make Eq. (83) become:

$$\nabla^2 = \nabla^2_{\mathrm{d}} + \nabla^2_{\mathrm{off}} \tag{85}$$

$$\nabla^2_{\mathrm{d}} = H_c\frac{\partial^2}{\partial\xi^0\partial\xi^0} + g^{11}\frac{\partial^2}{\partial\xi^1\partial\xi^1} + g^{22}\frac{\partial^2}{\partial\xi^2\partial\xi^2} \tag{86}$$

$$\nabla^2_{\mathrm{off}} = (g^{00} - H_c)\frac{\partial^2}{\partial\xi^0\partial\xi^0} + 2g^{01}\frac{\partial^2}{\partial\xi^0\partial\xi^1} + 2g^{02}\frac{\partial^2}{\partial\xi^0\partial\xi^2} + C\frac{\partial}{\partial\xi^0} \tag{87}$$

where H_c is relative to the order of g^{00}, i.e., $H_c = 1$ for the mapping addressed in section A.4, while $H_c = \frac{1}{h^2}$ for [84] and [90]. The decomposition of ∇^2 shown above is ready to be used in the numerical solving, i.e., the solution of momentum equation or the Poisson's equation for pressure field. For example, ∇^2_{d} can be solved implicitly using any solver, while ∇^2_{off} needs to be evaluated explicitly and lumped with the loading term on the right hand side.