UNIVERSIDAD AUTÓNOMA DE BAJA CALIFORNIA

FACULTAD DE INGENIERÍA, ARQUITECTURA Y DISEÑO

MAESTRÍA Y DOCTORADO EN CIENCIAS E INGENIERÍA

IMPLEMENTACIÓN DEL MÉTODO WDM PARA ESTIMAR EL ESPECTRO DIRECCIONAL DEL OLEAJE Y TRANSMITIRLO VÍA SATÉLITE

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

MAESTRO EN INGENIERÍA

P R E S E N T A :

JUAN FRANCISCO MARTÍNEZ OSUNA

CODIRECTORES

DR. HUMBERTO CERVANTES DE ÁVILA
DR. FRANCISCO J. OCAMPO TORRES

ENSENADA, BAJA CALIFORNIA, MÉXICO, NOVIEMBRE DE 2019

UNIVERSIDAD AUTÓNOMA DE BAJA CALIFORNIA

FACULTAD DE INGENIERÍA, ARQUITECTURA Y DISEÑO

MAESTRÍA Y DOCTORADO EN CIENCIAS E INGENIERÍA

Implementación del método WDM para estimar el espectro direccional del oleaje y transmitirlo vía satélite

TESIS

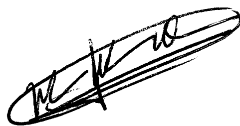
Que para obtener el grado de maestría en ingeniería presenta:

Juan Francisco Martínez Osuna

Aprobada por:

Humberto Cervantes de Ávila
Director de tesis

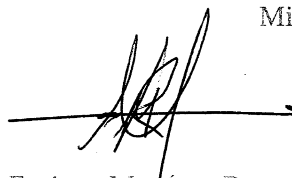
Francisco J. Ocampo Torres
Director de tesis



Manuel Moisés Miranda Velasco
Miembro del comité

Carlos Gómez Agis
Miembro del comité

í



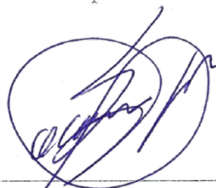
Miguel Enrique Martínez Rosas
Miembro del comité

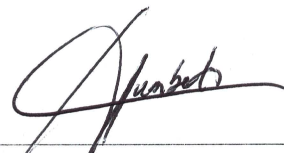
Ensenada Baja California, México. Diciembre de 2019

Resumen de la tesis que presenta **Juan Francisco Martínez Osuna** como requisito parcial para la obtención del grado de Maestro en Ingeniería.

Implementación del método WDM para estimar el espectro direccional del oleaje y transmitirlo vía satélite

Resumen aprobado por:





Dr. Franscisco J. Ocampo Torres
Director de tesis

Dr. Humerto Cervantes de Ávila
Director de tesis

Hoy en día, la investigación científica demanda la adquisición de una gran cantidad de datos y el avance de las telecomunicaciones puede no estar a la altura de las circunstancias, especialmente cuando se requiere monitoreo de variables ambientales en sitios lejanos y transmisión de datos en tiempo real. En respuesta a esta necesidad, se realizó este trabajo que consiste en implementar el Método Direccional de Wavelets (WDM, por sus siglas en inglés) al ejecutar programas en una computadora de a bordo, con el fin de estimar el espectro direccional del oleaje (EDO) in situ y transmitirlo en tiempo real a través del sistema de telemetría de IRIDIUM.

Se utilizó un arreglo de 6 alambres de capacitancia para medir el nivel de la superficie del mar y un conjunto de sensores de movimiento para corregir las mediciones y referirlas a un sistema inercial. Los sensores fueron instalados en una Boya Oceanográfica y de Meteorología Marina (BOMM) en la Bahía de Todos Santos cerca de Ensenada, B. C., México. Se implementó un algoritmo de comunicación bidireccional en una computadora de a bordo para ejecutar programas in situ a petición del usuario, y transmitir los resultados a una estación terrena a través de la red de satélites IRIDIUM. El algoritmo de comunicación fue desarrollado en lenguaje de programación C++ y puede ser utilizado para otros fines. Los programas para estimar el espectro direccional del oleaje fueron desarrollados en lenguaje de programación Python. Además, se desarrolló un programa en lenguaje M (Matlab) para despliegue del espectro direccional del oleaje en la estación terrena.

La información obtenida de la estimación del espectro direccional es muy valiosa para el apoyo en la toma de decisiones en cuanto al tráfico marítimo. Esta información también es útil para la toma de decisiones ante contingencias de dispersión de contaminantes (por ejemplo de hidrocarburos) y en aplicaciones científicas; para validación de modelos de pronósticos de oleaje, de interacción oleaje-corrientes así como la asimilación de datos en modelos para la predicción del oleaje.

Palabras clave: sensores, sistemas de adquisición de datos, oleaje, espectro direccional.

A mi familia:
Mis queridos padres y hermanos

Agradecimientos

Quiero expresar mi mas sincero agradecimiento a Humberto Cervantes y Paco Ocampo por haber aceptado ser directores de esta tesis y por su apoyo y guía durante todo este tiempo. A Miguel Martínez por darme la oportunidad de ingresar al posgrado. A mi comité de tesis: Carlos Gómez, Manuel Miranda y Miguel Martínez, por sus valiosas aportaciones y sugerencias para el desarrollo de este trabajo.

A Daniel P. y Marco L. quienes son parte fundamental en el desarrollo de este proyecto; de corazón ¡muchas gracias!. A Pedro O. por escuchar y sugerir sobre todo en los ratos de mutualista. A Erick R. por la paciencia en las salidas de campo. A Aimie M. por sus recomendaciones y buena vibra.

A los miembros del grupo de boyas, que apoyaron en las maniobras de instalación y mantenimiento de las BOMM y en las diferentes campañas de mediciones; ”excelentes experiencias y memorias”. A los miembros del Grupo de Oleaje del CICESE: Bernardo, Chiapas, Chinto, Diegos, Guillermo, Héctor, Jessica, Lalo, Laurita, Lucía, Melissa, Monty, Nico, Pancho, Reginaldo, Rodrigo y los demás, siempre dispuestos a apoyar y sugerir. A Héctor V. y Álex M., por estar siempre al pendiente del avance. A Magui P. por impulsarme a ingresar a la maestría.

Quiero agradecer a la Fundación Mexicana para la Ciencia (CONACYT) y a la Secretaría de Energía (SENER) ya que este estudio se realiza en el marco del proyecto 201441 del Consorcio Mexicano de Investigación del Golfo de México (CIGOM), financiado por ambas instituciones.

A la UABC por todo lo aprendido desde la licenciatura; simplemente mi *alma mater*. Al CICESE por todas las facilidades prestadas para realizar los estudios de posgrado, particularmente a Modesto O. por todo el apoyo.

A Damaris por su paciencia, apoyo, lonches y cariño durante todo este tiempo: ¡Muchas gracias!. Por último, pero no menos importante, todas las gracias a mi familia, a mi hermosa madre, a mis hermanos(as), a mi padre y a mi tío David. A todos gracias por apoyarme siempre y por sus constantes muestras de cariño.

Índice general

Agradecimientos II

1. Introducción 1

1.1. Generalidades 1

1.1.1. Espectro Direccional del Oleaje 4

1.1.2. Método Direccional de Wavelet - WDM 5

1.1.3. Antecedentes 6

1.1.4. Técnicas para medición del oleaje in situ 7

1.2. Justificación 13

1.3. Objetivos 13

2. Metodología 16

2.1. Boya Oceanográfica y de Meteorología Marina - BOMM 17

2.2. Sensores 19

2.2.1. Alambres de capacitancia - OSS Wave Staff 19

2.2.2. Sensor de Movimiento - EKINOX2-M SBG-systems 21

2.2.3. Estación meteorológica - GILL GMX600 24

2.3. Sistema de Adquisición de Datos 26

2.3.1. Computadora de a bordo - Nitrogen6x 26

2.4. Estimación del espectro direccional del oleaje (EDO) 30

2.5. Transmisión de datos 35

2.5.1. IRIDIUM 36

2.5.2. Tranceptor satelital - TAOGLAS Spartan STS.01 36

2.6. Sitio y campaña de mediciones 37

3. Implementación del método WDM y Resultados **39**

3.1. Configuración de la computadora de a bordo 39

3.2. Estimación del Espectro Direccional del Oleaje 43

3.3. Algoritmo de comunicación bidireccional 47

3.3.1. Estación terrena 47

3.3.2. Boya 48

3.4. Recuperación y despliegue del EDO 54

4. Conclusiones **57**

Índice de figuras

1.1. Características de una onda sinusoidal en el espacio y el tiempo. a) Se ilustra la amplitud, altura, pendiente, cresta, valle y longitud de la onda. b) Se muestra el periodo de la onda. 2
1.2. Representación espectral del oleaje (Holthuijsen, 2007). 3
1.3. Distribución de la energía en las frecuencias y espectro direccional del oleaje (EDO). 5
1.4. Plataformas boyantes con sensores para estimación del oleaje 8
1.5. Plataformas fijas con sensores para estimación del oleaje 10
1.6. Sensores sumergidos para estimación del oleaje 11
1.7. Sensores remotos para estimación del oleaje 12
1.8. Diagrama a bloques del sistema para estimar el espectro direccional del oleaje 14

2.1. Fotografía de la BOMM (izq.) y la boya Tether (der.), instaladas cerca de la Isla Todos Santos, Ensenada, B.C. 17
2.2. Esquema de una Boya Oceanográfica y de Meteorología Marina del CICESE 19
2.3. Alambres de capacitancia - OSS Wave Staff instalados en una BOMM. 20
2.4. Arreglo geométrico de las posiciones de los alambres de capacitancia en la BOMM - Vista de planta 21
2.5. Definición del sentido del movimiento y los ángulos de orientación e inclinación de la boya. 22
2.6. Sensor de Movimiento - EKINOX2-M SBG-systems 23

2.7. Estación Meteorológica GILL-GMX600 24
2.8. Microcomputadora de a bordo Nitrogen6x - (https://boundarydevices.com) 28
2.9. Microcomputadora de a bordo Nitrogen6x - (https://boundarydevices.com) 29
2.10. Descripción de los ángulos de rotación provocados por los movimientos de cabeceo, alabeo y guiñada. Las figuras son ilustrativas; las dimensiones de las boyas, la distancia de separación y las inclinaciones no corresponden con la escala real. 32
2.11. Constelación LEO de Iridium 36
2.12. Transceptor satelital para transmisión de los datos a través de la red IRIDIUM. 37
2.13. Localización del sitio para pruebas de flotabilidad y funcionamiento de las BOMM, al noroeste de la Isla Todos Santos, B.C.¨ 38

3.1. Diagrama a bloques de la programación para obtener el espectro direccional del oleaje. 43
3.2. Diagrama de flujo del programa principal para obtener el EDO. . . . 44
3.3. Diagrama de flujo de los programas secundarios para obtener el EDO. 44
3.4. Información del correo electrónico.¨ 48
3.5. Diagrama de flujo para la configuración del transceptor.¨ 50
3.6. Diagrama de flujo para estimar el EDO.¨ 52
3.7. Diagrama de flujo para la transmisión del EDO.¨ 54
3.8. Espectros direccionales del oleaje del día 2017-11-17 10:00 (izquierda) y el día 2017-11-17 10:00 (derecha). Los círculos representan a las frecuencias .1Hz, .2Hz, .3Hz y .4Hz. La barra de colores indica la densidad de energía y sus unidades son: $\frac{m^2}{Hz\ rad}$ 56

Índice de cuadros

2.1. Variables registradas por la estación meteorológica GMX-600 25
2.2. Características eléctricas de la computadora de a bordo Nitrogen6x . 28

Capítulo 1

Introducción

1.1. Generalidades

Una onda, en su definición más simple, es una señal (de información o perturbación) que se propaga con movimiento oscilatorio a través de un medio (Fleisch and Kinnaman, 2015). Las características principales de una onda se muestran en la figura 1.1. Una cresta corresponde a la parte más alta que puede alcanzar una onda mientras que un valle corresponde a la parte más baja; la longitud de onda, $\boldsymbol{L}$, es la distancia horizontal entre dos crestas sucesivas; la altura de la onda, $\boldsymbol{H}$, es la distancia vertical entre la cresta y el valle y el tiempo que transcurre para que dos crestas consecutivas pasen por el mismo punto se define como el periodo de la onda, $\boldsymbol{T}$, mientras que el inverso, $\boldsymbol{1/T}$, es la frecuencia de la onda, $\boldsymbol{F}$, la cual se define como el número de repeticiones por unidad de tiempo de cualquier fenómeno o suceso periódico.

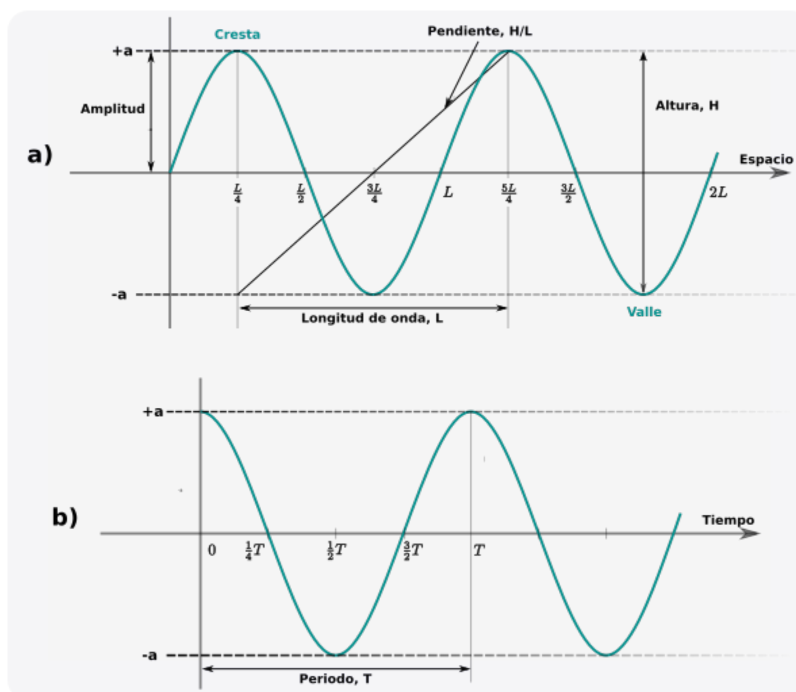


Figura 1.1: Características de una onda sinusoidal en el espacio y el tiempo. a) Se ilustra la amplitud, altura, pendiente, cresta, valle y longitud de la onda. b) Se muestra el periodo de la onda.

Las olas son una perturbación de la superficie del mar provocada por la propagación de la energía mecánica proporcionada por el viento, en la interfaz océano-atmósfera. El oleaje puede considerarse como un conjunto de ondas de pendiente pequeña con amplitudes, frecuencias y dirección de propagación dadas. El oleaje puede propagarse grandes distancias con relativamente poca disipación de energía.

Con algunos parámetros estadísticos, se pueden describir las condiciones promedio del oleaje (por ejemplo, la altura típica, el periodo y la dirección predominante) y proporcionan información un tanto limitada del fenómeno de interés en este trabajo. Con la utilización del concepto espectral se puede obtener una descripción detallada y más robusta del oleaje. La descripción espectral del fenómeno se basa en la suposición de que el movimiento aleatorio de la superficie del mar puede tratarse como la superposición (suma infinita) de un gran número de componentes armónicos (Holthuijsen, 2007) (ver fig. 1.2).

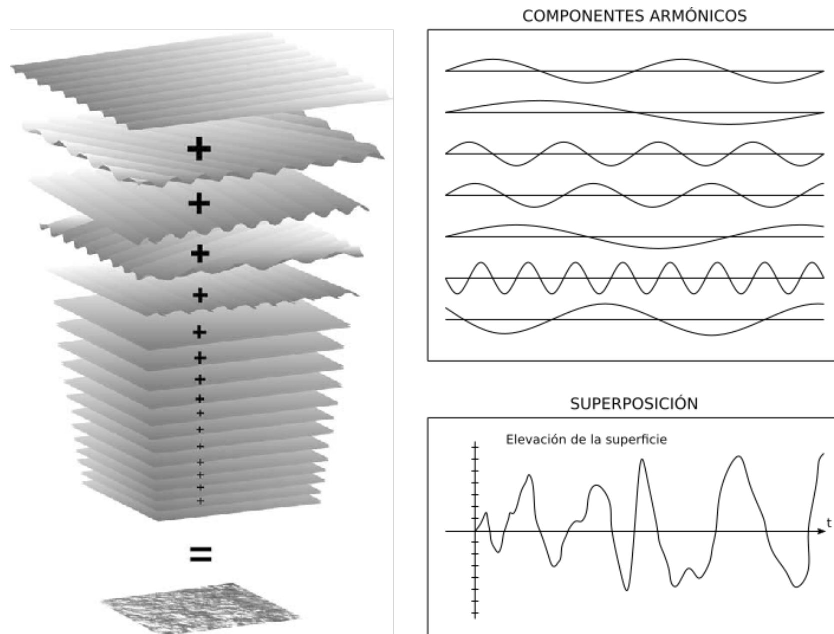


Figura 1.2: Representación espectral del oleaje (Holthuijsen, 2007).

La importancia de estudiar el oleaje reside principalmente en el aspecto ingenieril y científico.

Desde el punto de vista ingenieril, el estudio del oleaje se enfoca en los campos de la ingeniería civil y naval, como lo es la arquitectura naval e ingeniería costera y portuaria, entre otros. Además, es de gran apoyo para la toma de decisiones en la gestión costera, seguridad marítima, etc.

Desde el punto de vista científico, el oleaje es uno de los mecanismos más importantes que influyen en los procesos de transferencia de energía, momentum, calor y gases entre la atmósfera y el océano por lo que su comprensión aporta considerablemente al estudio y entendimiento del clima global. Otro aspecto importante consiste en el estudio de la dinámica del oleaje y sus mecanismos de propagación. Además, su estudio es muy útil para validación de modelos de pronósticos de oleaje, de interacción oleaje-corrientes y/o asimilación de datos en modelos para la predicción del oleaje.

1.1.1. Espectro Direccional del Oleaje

Con el espectro direccional, es posible describir algunas propiedades del oleaje, ya sea regular o irregular. Es una representación completa de la distribución de la energía en el dominio de las frecuencias y direcciones de propagación de cada una de sus componentes. La distribución de la energía en el dominio de las frecuencias puede representarse con el espectro de frecuencia $\boldsymbol{S(\omega)}$, mientras que en el dominio de la dirección se representa con la función de distribución direccional $\boldsymbol{D(\omega,\theta)}$.

El espectro direccional del oleaje proporciona información sobre diferentes sistemas de olas, que pueden propagarse en distintas direcciones en el océano. Además, se puede obtener información de otros parámetros integrales importantes como la altura significante del oleaje ($\boldsymbol{Hs}$), el periodo dominante de las olas ($\boldsymbol{Tp}$) y la dirección promedio de su propagación, entre otros.

En la figura 1.3 se muestra un ejemplo de un espectro direccional del oleaje (izq.) para un estado del mar (der.) en donde la energía está contenida principalmente en dos sistemas de olas. Se observa energía en un sistema de olas de periodo largo ($\sim$ 20 segundos) proveniente del suroeste y en otro de periodo corto ($\sim$ 6,66 segundos) propagándose en sentido contrario hacia el suroeste. El espectro direccional del oleaje facilita identificar el periodo y la dirección de propagación de las olas, lo cual no puede ser inferido de manera visual del estado del mar.

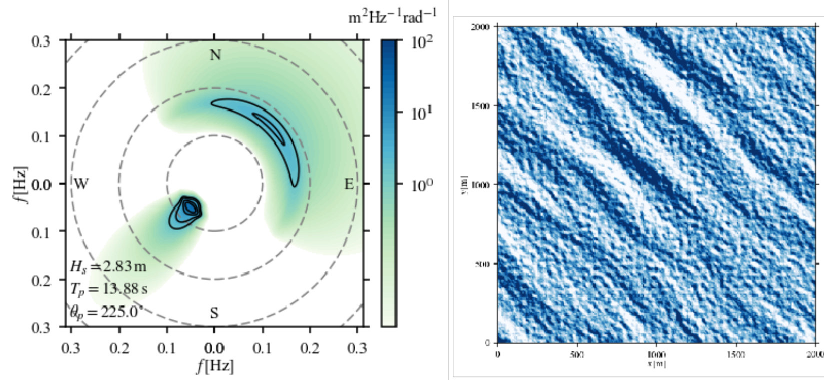


Figura 1.3: Distribución de la energía en las frecuencias y espectro direccional del oleaje (EDO).

1.1.2. Método Direccional de Wavelet - WDM

La información direccional de las olas se obtiene in situ generalmente a partir de las mediciones de las propiedades de las olas (por ejemplo, la elevación de la superficie del mar y el vector de la pendiente de la ola) en un punto o de la misma propiedad (por ejemplo, la elevación de la superficie del mar) en tres o más puntos (Donelan et al., 2015).

El método WDM (Wavelet Directional Method) fue propuesto por Donelan et al. (1996) como una alternativa a los métodos convencionales para estimar el espectro direccional del oleaje. Los métodos convencionales son, como ejemplo, el DFTM (Directional Fourier Transform Method), el MLM (Maximum Likelihood Method) empleado por Capon (1979) utilizando un arreglo de sensores para determinar la velocidad vectorial de las ondas propagadas y el MEM (Maximum Entropy Method) utilizado por Lygre and Krogstad (1986) como una mejor alternativa al aplicarse a datos de sensores de inclinación. El método WDM se basa en la transformada de Wavelet, que se define como la convolución entre la señal de la superficie libre del mar y una función conocida como Wavelet Madre (Torrence and Compo, 1998). Consiste en obtener la energía y la fase de cada componente espectral a partir de la señal que se detecta en diferentes pares de sensores de los cuales se conoce su separación y por lo tanto se puede calcular la magnitud y dirección del número de onda obteniendo

con esto una estimación del espectro direccional.

Según Donelan et al. (2015) los métodos MLM y MEM requieren estacionariedad y producen únicamente los espectros en función de la frecuencia y dirección mientras que el método WDM proporciona además el espectro en el dominio del número de onda y dirección. Con este método, también se puede realizar análisis no estacionario de algunas propiedades de oleaje y su agrupamiento, incluyendo la descripción de esas características para los varios sistemas de oleaje presentes de manera simultánea.

1.1.3. Antecedentes

Longuet-Higgins et al. (1963) desarrollaron una técnica para estimar el espectro direccional de las ondas superficiales del océano a partir de los movimientos de una estructura flotante basándose en que la aceleración vertical de la boya y los ángulos de cabeceo (inclinación en el eje transversal) y balanceo (inclinación en el eje longitudinal) pueden utilizarse para determinar los primeros cinco coeficientes de Fourier $(a_0, a_1, b_1, a_2, b_2)$ de la distribución de energía para cada banda de frecuencia. Los resultados de esa investigación representan el desarrollo fundamental para el inicio de las actividades de observaciones y determinación de características del oleaje, utilizando sensores en estructuras flotantes.

Posteriormente, Hanson et al. (1997) hicieron uso de sensores de velocidad de cabeceo y velocidad de balanceo, un acelerómetro de 3 ejes y un cable capacitivo montados en un catamarán con el fin de estimar la altura de las olas, mientras que para estimar el espectro direccional en función de la frecuencia y la dirección de propagación del oleaje utilizaron un arreglo de cables capacitivos y el estimador Data-Adaptive Spectral Estimator (DASE) propuesto por Davis and Regier (1977) para estimar el espectro de frecuencias direccionales de las ondas de viento, incluyendo la presencia del desplazamiento Doppler. Vlachos and Tsabaris (2007) utilizaron un método paramétrico para calcular el espectro direccional del oleaje a partir de las aceleraciones verticales y horizontales de una boya. Con el método paramétrico se supone la su-

perposición de dos trenes de onda independientes que dan una mejor aproximación a la naturaleza multidireccional del campo ondulatorio.

En los trabajos anteriores se realizaron pruebas puntuales y recolección de datos in situ mas no se transmitió información del estado del mar en tiempo real.

Por su parte, Takahashi et al. (2014) diseñaron un sistema para observación de tsunamis y la deformación de la corteza terrestre utilizando sensores de presión y un sistema de Posicionamiento Preciso de Puntos (PPP). El dispositivo se montó en una boya y los datos obtenidos se enviaron en tiempo real hasta una estación base utilizando la constelación de satélites IRIDIUM.

En el Centro de Investigación Científica y de Educación Superior de Ensenada, B.C., México, (CICESE) se diseñó y construyó un sistema de adquisición de datos modular, de bajo costo, robusto y autónomo para adquisición de datos de sensores meteorológicos en el contexto de las torres para medición de flujos de CO_2 con la técnica de Eddie Covariance (Castro et al., 2017). La filosofía modular de este sistema permite que las diferentes partes del equipo puedan ser fácilmente reemplazadas sin afectar la operación del sistema completo y por su parte, la experiencia adquirida es de gran utilidad para el desarrollo del presente trabajo.

1.1.4. Técnicas para medición del oleaje in situ

Para obtener un registro en el tiempo del movimiento ascendente y descendente de la superficie del mar y poder estimar algunas características del oleaje in situ, se utilizan instrumentos diversos que pueden estar instalados en diferentes puntos de la columna de agua (por ejemplo, boyas en la superficie del mar, un sensor de presión en el fondo del mar o alambres de capacitancia a lo largo de la columna de agua) o en plataformas sobre la superficie del mar (por ejemplo, radares). Existen varias técnicas para estimar el oleaje y obtener información sobre el estado del mar y sus características. Las más comunes son:

- Plataformas boyantes

  Este método consiste en registrar el movimiento tridimensional de las partículas de agua en la superficie del mar utilizando estructuras boyantes (ver figura 1.4). La técnica más común es medir la aceleración vertical con un acelerómetro a bordo. Al integrar la aceleración dos veces se puede obtener el movimiento vertical de la boya y por lo tanto la elevación de la superficie del mar en función del tiempo.

  Existen boyas que además incorporan sensores para medir la inclinación de la superficie del mar así como su propia orientación con respecto el norte geográfico (inclinómetros y magnetómetro respectivamente). Con estos dispositivos se puede estimar la pendiente de la superficie y la dirección media de las olas.

Figura 1.4: Plataformas boyantes con sensores para estimación del oleaje

- Plataformas fijas

  Este método consiste en registrar la posición vertical de la superficie del agua utilizando un cable instalado verticalmente en una plataforma (ver figura 1.5).

Uno de los extremos del cable debe estar fijo por encima de la superficie del agua y el otro extremo debe llegar hasta un punto por debajo de la superficie.

Una técnica es medir la longitud del cable por encima de la superficie, por ejemplo, midiendo la resistencia eléctrica de la parte "seca" del cable. En la práctica se utilizan uno o dos hilos con una cadena de electrodos que provocan un cortocircuito en la superficie del agua. Otra técnica es medir la capacitancia de dos cables eléctricos en paralelo o de un solo cable eléctrico dentro de una superficie aislante. También es posible enviar una señal eléctrica de alta frecuencia por el cable, que se reflejará en la superficie del agua, determinando de nuevo la posición de la superficie del agua en el cable.

Al igual que algunos métodos con sensores en boyas, estas técnicas no proporcionan información de la dirección de las olas directamente. Para obtener dicha información, se puede utilizar un grupo o arreglo de cables. Por ejemplo, tres cables en los vértices de un triángulo pueden ser usados para estimar la pendiente de la superficie o para detectar diferencias de fase entre los cables.

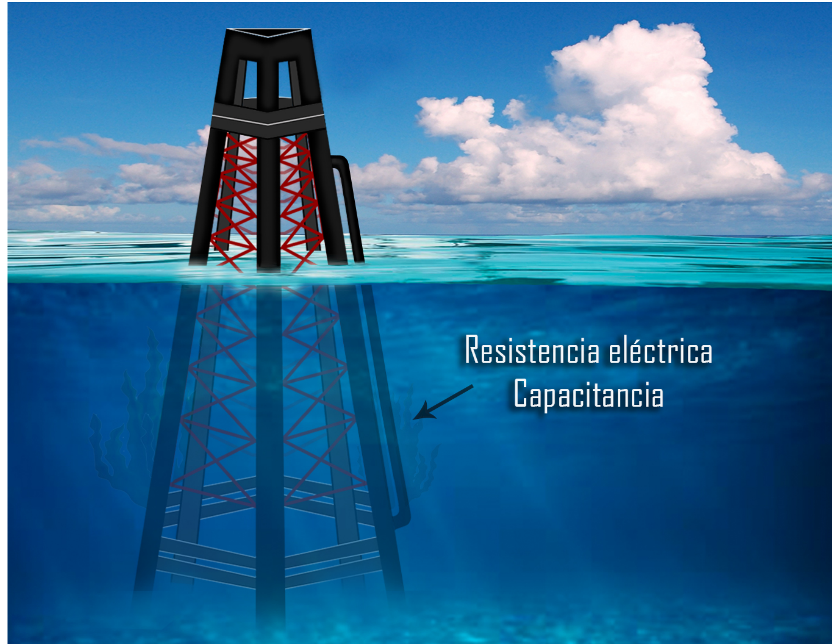


Figura 1.5: Plataformas fijas con sensores para estimación del oleaje

- Dispositivos sumergidos

  Este método consiste en instalar un dispositivo en algún punto fijo por debajo de la superficie del mar, que sea capaz de registrar algunas características de la superficie y poder así estimar el oleaje (ver figura 1.6).

  Una técnica es utilizar una ecosonda invertida para medir la distancia entre el dispositivo y la superficie del agua utilizando un haz acústico. Otra técnica consiste en un transductor de presión o un medidor de corriente para registrar las fluctuaciones de presión o el movimiento orbital (respectivamente) inducidos por las olas sobre la superficie del mar.

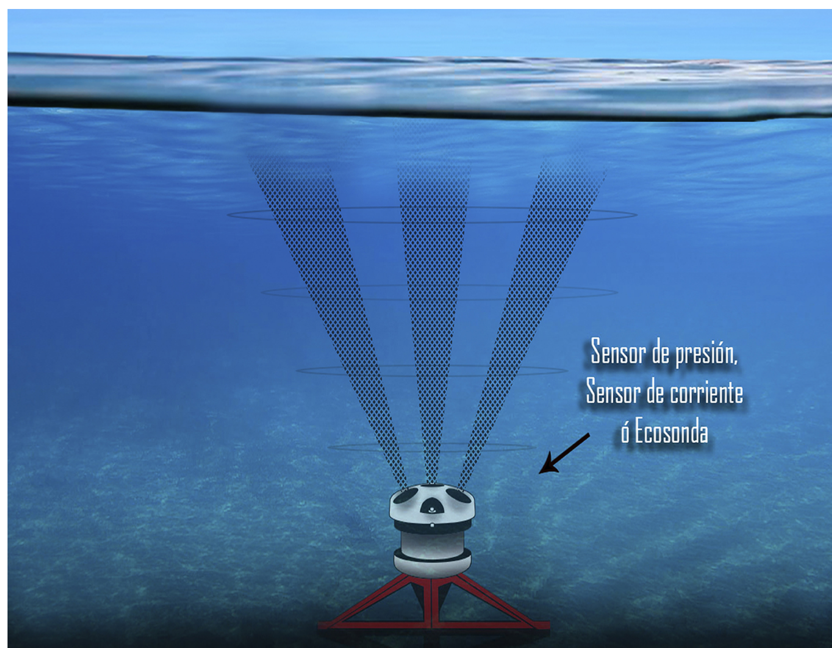


Figura 1.6: Sensores sumergidos para estimación del oleaje

- Sensores remotos

  Consiste en utilizar instrumentos instalados en plataformas que se encuentran sobre la superficie del mar, es decir, que no tienen contacto directo con el agua (ver figura 1.7). La plataforma puede ser una torre de observación en el mar, un barco, un avión o un satélite. El principio de esta técnica es recibir el reflejo de la luz visible, infrarroja o de la energía emitida por un radar, provocado por la superficie del mar.

  Imágenes:
  Una técnica es la estéreo-fotografía, la cuál permite obtener una imagen tridimensional de la superficie del mar al tomar fotografías de alta calidad de las secciones superpuestas del terreno que hay por debajo en intervalos cortos de tiempo. Las diferencias (paralaje) en las fotos superpuestas pueden convertirse en elevaciones, creando así la imagen tridimensional.

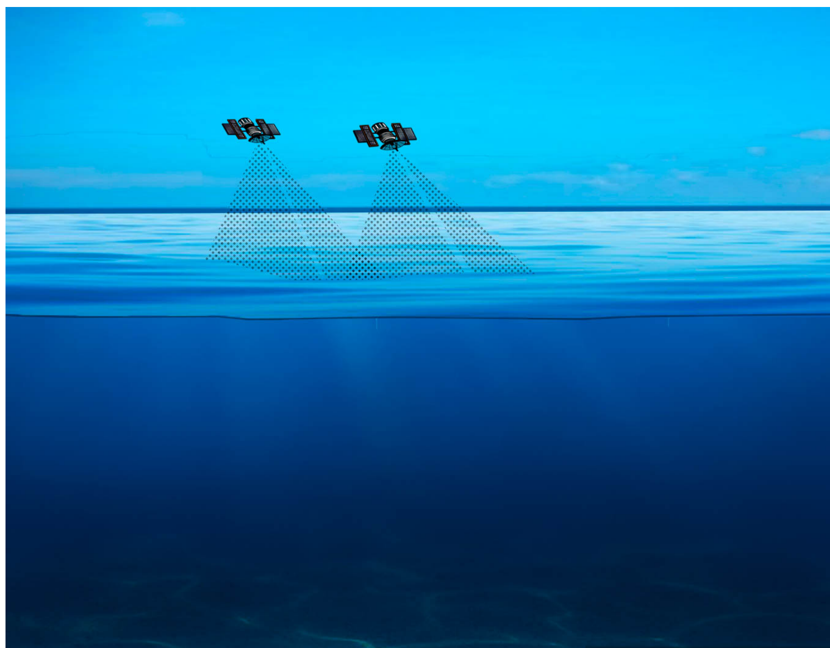


Figura 1.7: Sensores remotos para estimación del oleaje

Altimetría:

Otra técnica distinta a la fotografía pero que utiliza luz (visible o infrarroja) es el láser. Utilizado como medidor de distancia, o mejor dicho, como altímetro, un láser apuntando hacia abajo puede medir la distancia vertical desde el instrumento hasta la superficie del mar con bastante precisión.

Altimetría acústica.

Los ecosondas no sólo se utilizan como instrumentos in situ, sino también como instrumentos de tele-detección. Cuando se montan sobre el agua mirando hacia abajo, con un haz estrecho, se pueden utilizar para medir la distancia a la superficie del mar.

Altimetría de radar.

Un radar de haz estrecho también puede utilizarse como altímetro al "mirar" directamente a la superficie del mar. Si el dispositivo está situado cerca de la superficie del agua (en una plataforma fija o en un avión de bajo vuelo), el radar

es lo suficientemente preciso como para medir la elevación real de la superficie del mar directamente debajo del instrumento.

1.2. Justificación

Una serie de tiempo proveniente de sensores para estimar el oleaje sugiere el manejo de la información de la manera más compacta y comprensible posible. Al observar la serie temporal completa, se intuye la necesidad de reducir dichos datos prescindiendo de la estructura detallada de toda la serie temporal pero, a la vez, reteniendo la mayor cantidad de información posible.

La implementación de un método para estimación del espectro direccional del oleaje in situ permitirá obtener información detallada del estado del mar al momento sin la necesidad de transmitir toda la información registrada, optimizando así la utilización del ancho de banda del sistema de telemetría elegido. Esto resulta además en un beneficio económico ya que para transmitir la información se utilizan sistemas satelitales, en los cuales el cobro se hace de acuerdo al ancho de banda o cantidad de datos utilizados.

La información obtenida de la estimación del espectro direccional, es muy valiosa para el apoyo en la toma de decisiones en cuanto al tráfico marítimo. Esta información también es útil para la toma de decisiones ante contingencias de dispersión de contaminantes (por ejemplo de hidrocarburos) y en aplicaciones científicas; para validación de modelos de pronósticos de oleaje, de interacción oleaje-corrientes así como la asimilación de datos en modelos para la predicción del oleaje.

1.3. Objetivos

Objetivo general

Diseñar e implementar un algoritmo para estimar el espectro direccional del oleaje, utilizando el método WDM y transmitirlo de manera inmediata desde una Boya Oceanográfica y de Meteorología Marina (BOMM) hasta una estación terrena bajo demanda del usuario (ver fig. 1.8).

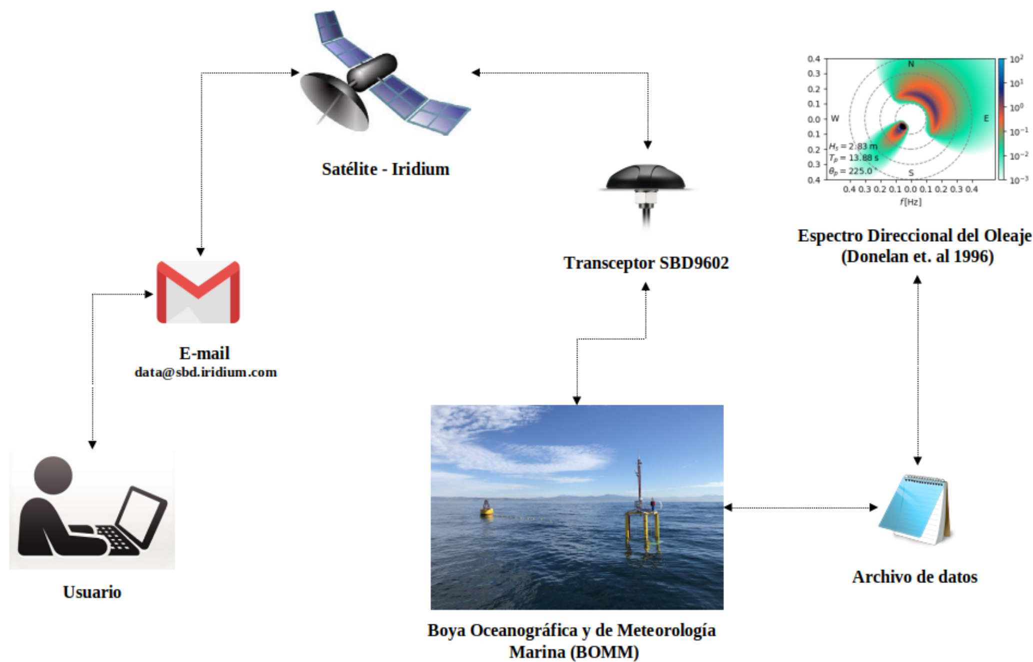


Figura 1.8: Diagrama a bloques del sistema para estimar el espectro direccional del oleaje

Objetivos específicos

- Contruir un Sistema de Adquisición de Datos para pruebas de laboratorio incorporando una computadora de abordo Nitrogen6x de Boundary Devices.
- Obtener el espectro direccional del oleaje mediante la ejecución de programas en la computadora de a bordo, desarrollados en lenguaje de programación Python.

- Diseñar e implementar un protocolo de comunicación bidireccional entre una boya y una estación terrena utilizando un transceptor satelital.
- Desarrollar el software para despliegue de los datos.

Capítulo 2

Metodología

En el Centro de Investigación Científica y de Educación Superior de Ensenada (CICESE) se han construido Boyas Oceanográficas y de Meteorología Marina (BOMM) como parte de las actividades de un consorcio para investigación científica y desarrollo de proyectos multidisciplinarios relacionados a posibles impactos ambientales de la industria del gas y petróleo, en los ecosistemas marinos del Golfo de México. Estas boyas registran información de diversos sensores oceanográficos y meteorológicos, la almacenan en memoria y envían únicamente banderas de estado (por ejemplo el promedio en el tiempo de las diferentes variables) a través de satélite hasta un servidor de cómputo.

Los avances tecnológicos permiten incorporar sensores de alta frecuencia de muestreo y computadoras con gran capacidad de procesamiento y almacenamiento en las que se pueden implementar algoritmos avanzados de cómputo, por ejemplo, para estimar el espectro direccional del oleaje in situ y transmitirlo inmediatamente vía satélite hasta una estación terrena.

2.1. Boya Oceanográfica y de Meteorología Marina - BOMM

La BOMM es una plataforma de metal de aproximadamente 14 metros de longitud diseñada y construida principalmente para realizar mediciones de las variables dinámicas más importantes en los procesos de la interacción entre el océano y la atmósfera y para estimar el flujo de gases, momentum y calor que ocurren entre los dos fluidos más importantes en nuestro planeta: el océano y la atmósfera, así como para determinar las condiciones atmosféricas y oceánicas en mar abierto. Particularmente, con esta boya se puede medir la elevación de la superficie del mar en un arreglo de puntos así como los movimientos lineales y angulares de la estructura con el fin de estimar la dirección de propagación del oleaje. (Peláez-Zapata (2018); García-Nava (2006); Graber et al. (2000); Ocampo-Torres et al. (2010)).

La BOMM se sujeta a una boya auxiliar (Tether) con un cable de acero de 30 metros de longitud (ver fig. 2.1), de tal manera que tiene libre movilidad en un radio definido. La Tether por su parte, está anclada al fondo marino mediante un peso muerto de aproximadamente dos toneladas.

Figura 2.1: Fotografía de la BOMM (izq.) y la boya Tether (der.), instaladas cerca de la Isla Todos Santos, Ensenada, B.C.

La estructura se divide en 3 partes: Mástil, Caja y Pie (ver fig. 2.2)

El mástil (parte superior), de 4 metros de longitud, se utiliza para soporte de los sensores no sumergibles (meteorológicos y atmosféricos), dispositivos de señalización y de transmisión de datos. La caja (parte central) es una estructura pentagonal de 3.5 metros de longitud y 2 metros de diámetro en la que se encuentran instalados los sensores para medición de la elevación de la superficie del mar. El pie es de 6.5 metros de longitud y contiene los sensores sumergibles (oceanográficos), sensor de movimiento, la instrumentación electrónica y las baterías.

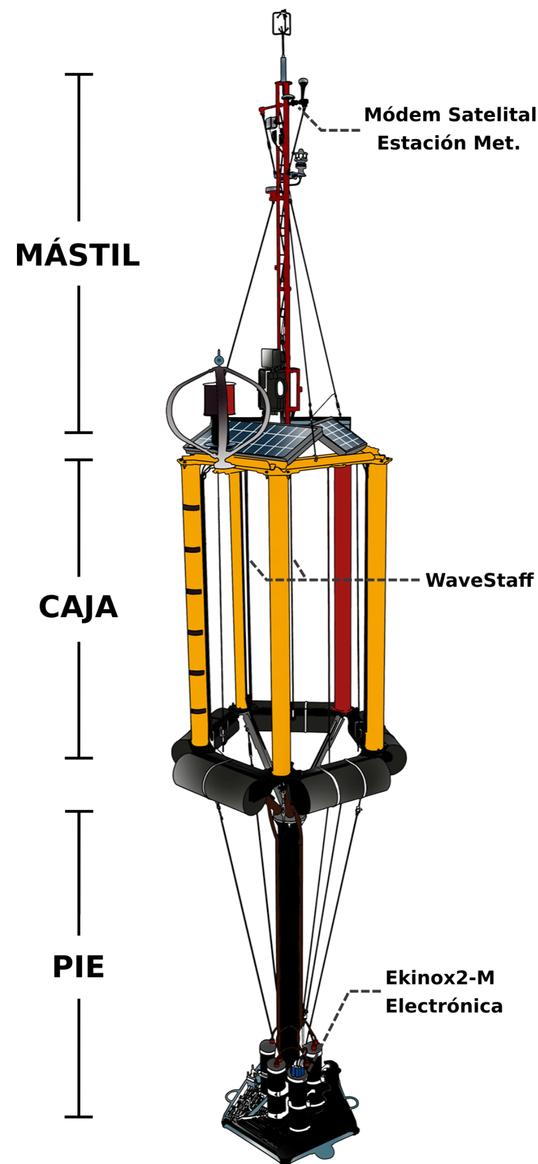


Figura 2.2: Esquema de una Boya Oceanográfica y de Meteorología Marina del CICESE

2.2. Sensores

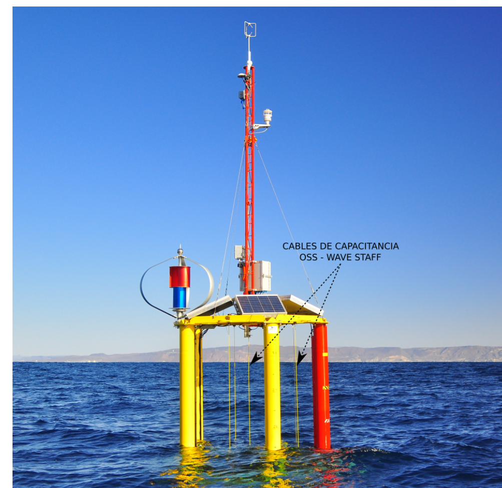
2.2.1. Alambres de capacitancia - OSS Wave Staff

Los sensores de nivel capacitivos miden el cambio que se produce en la capacitancia entre dos placas provocado por los cambios de nivel de un líquido. Estos dispositivos están formados ya sea por una varilla aislada conectada a un transmisor

y al fluido, o por una varilla no aislada conectada al transmisor y a la pared de un tanque o a una sonda de referencia. A medida que el nivel del fluido sube (o baja) y llena más espacio entre las placas, la capacitancia total aumenta proporcionalmente y un circuito electrónico llamado puente de capacitancia mide la capacitancia total proporcionando una medición continua del nivel del líquido considerado.

El OSS - Wave Staff (fig. 2.3) es un sensor de nivel del agua del tipo capacitivo que combina un empaquetado resistente, sellado e impermeable, un circuito estable de detección de temperatura y un microprocesador de baja potencia que puede ser programado para muestreo y transmisión de datos automáticamente o para toma de muestras bajo demanda. Opera de 5.5VDC a 40VDC y cuenta con salida de datos analógica, digital del tipo RS232 y 2 alarmas. El formato de salida de datos por puerto serie incluye el nivel y la temperatura del agua y puede ser en ASCII o en binario; las salidas de alarma son del tipo Open-Drain a 350mA.

(a) Alambre de capacitancia



(b) Wave Staff instalados en una BOMM

Figura 2.3: Alambres de capacitancia - OSS Wave Staff instalados en una BOMM.

Entre los sensores que integran la BOMM existen 6 alambres de capacitancia de 3.5 metros de longitud instalados en cada uno de los lados y centro de la caja (ver figura 2.4). Estos sensores se utilizan para obtener mediciones de la elevación de la

superficie libre del mar a una tasa de muestreo de 20Hz. Físicamente, los alambres están distribuidos en un arreglo geométrico de tal manera que permite calcular el espectro del oleaje en función de la frecuencia, dirección y número de onda.

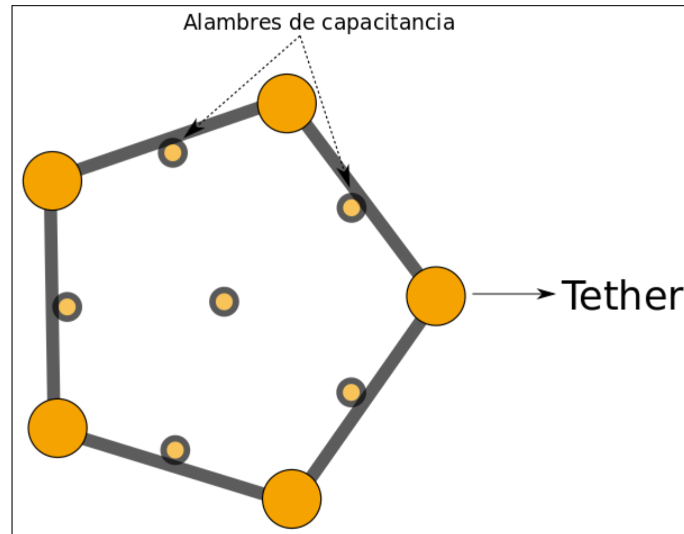


Figura 2.4: Arreglo geométrico de las posiciones de los alambres de capacitancia en la BOMM - Vista de planta

2.2.2. Sensor de Movimiento - EKINOX2-M SBG-systems

Las observaciones de la elevación de la superficie libre del mar se ven afectadas directamente por el movimiento propio de la boya.

Para corregir estas observaciones y referirlas a un sistema inercial o sin movimiento, se cuenta con un sensor de movimiento instalado en la base del pie de la boya, adquiriendo datos de aceleración y la razón de cambio de los movimientos angulares a una tasa de muestreo de 100 Hz con los que se pueden inferir los movimientos lineales (o de traslación) y angulares (o de rotación) de la boya en las tres dimensiones (ver fig. 2.5).

Los movimientos lineales que afectan a la boya son:
Avance - Movimiento a lo largo del eje longitudinal $\boldsymbol{x}$.

Deriva - Movimiento a lo largo del eje transversal $\boldsymbol{y}$.

Arfada - Movimiento de ascenso y descenso, es decir, a lo largo del eje vertical $\boldsymbol{z}$.

Los movimientos que definen la inclinación y la orientación de la boya son:

Alabeo - Movimiento de rotación alrededor del eje $\boldsymbol{x}$.

Cabeceo - Movimiento de rotación alrededor del eje $\boldsymbol{y}$.

Guiñada - Movimiento de rotación alrededor del eje $\boldsymbol{z}$. Su punto de referencia es el norte geográfico.

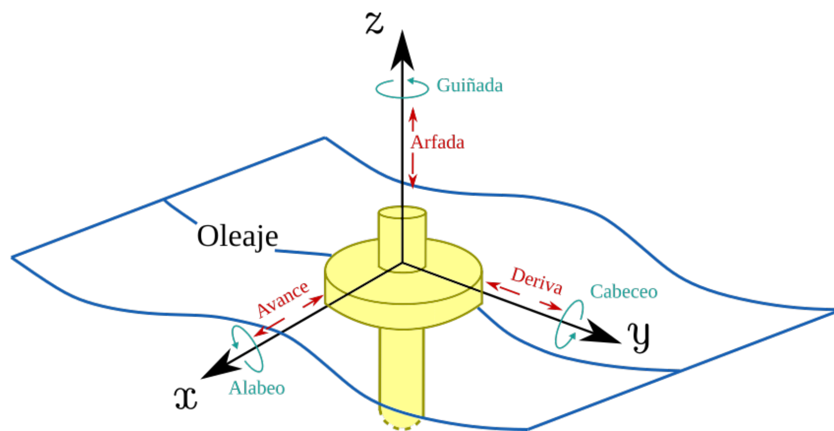


Figura 2.5: Definición del sentido del movimiento y los ángulos de orientación e inclinación de la boya.

El sensor Ekinox2-M (Fig. 2.6) es un sistema de referencia de movimiento submarino (MRU; Subsea Motion Reference Unit) que combina un sistema de medición inercial (IMU; Inertial Motion Unit) y un filtro de Kalman extendido (EKF) para obtener con precisión las variables que describen el movimiento de la boya.

El dispositivo está protegido con una carcasa de titanio que soporta hasta 6000 mts. de profundidad y la precisión de este sensor es de 2.5 cm. en los movimientos lineales y de 0,05° en los movimientos angulares.

Figura 2.6: Sensor de Movimiento - EKINOX2-M SBG-systems

Las variables que mide el Ekinox2-M son:

- Aceleración en x, y y z $[\frac{m}{s^2}]$
- Tasa de cambio del ángulo en x, y y z $[\frac{rad}{s}]$
- Delta de velocidad en x, y y z (sculling)$[\frac{m}{s^2}]$
- Delta del ángulo en x, y y z (coning) $[\frac{rad}{s}]$
- Temperatura interna del sensor [°C]

El dispositivo utiliza un sistema de coordenadas de "mano derecha" con el eje $\boldsymbol{x}$ positivo apuntando en la dirección de sujeción de la boya (hacia la Tether), el eje $\boldsymbol{y}$ positivo apuntando a 90° del eje $\boldsymbol{x}$ en sentido horario y el eje $\boldsymbol{z}$ positivo apuntando hacia abajo, es decir, hacia el fondo del mar. Como el sistema de referencia de la BOMM tiene el eje z positivo hacia arriba, los datos de salida del Ekinox2-M se deben corregir para que se adapten al sistema de referencia de la boya.

2.2.3. Estación meteorológica - GILL GMX600

Las estaciones meteorológicas Gill MaxiMet son dispositivos robustos, sin partes móviles, que incorporan diversos sensores para la medición de parámetros meteorológicos y ambientales conforme a las normas internacionales.

La estación meteorológica GILL GMX-600 (Fig. 2.7) registra datos de velocidad y dirección del viento, temperatura del aire, humedad relativa, presión barométrica y precipitación pluvial, entre otros, a una tasa de muestreo de 1 Hz. La información obtenida de los diferentes sensores se combina en una sola cadena de datos que puede ser del tipo ASCII transmitida por los puertos de comunicación RS232/RS422/RS485 (2 hilos, punto a punto), MODBUS RTU, NMEA y/o salidas tipo SDI-12. Las unidades, la tasa de transmisión y los formatos de salida son seleccionables por el usuario.

Figura 2.7: Estación Meteorológica GILL-GMX600

La estación GMX-600 se encuentra instalada en el mástil de la boya a 4.5 mts de altura aprox. respecto al nivel de flotación y tiene una marca que se utiliza como referencia para calcular la dirección relativa del viento.

La dirección verdadera del viento es calculada por la estación a partir de su orientación respecto al norte magnético y a la declinación magnética en función de la posición dada por el GPS incorporado. Si la marca de la estación meteorológica está alineada con el norte, las direcciones verdadera y relativa son las mismas.

La estación tiene incorporada una brújula con la que se puede estimar la orientación de la boya calculando la diferencia entre la dirección verdadera (θ_{Wtrue}) y la dirección relativa (θ_{Wrel}) del viento, es decir:

$$\psi_{maximet} = (\theta_{Wtrue} - \theta_{Wrel} + \phi_{maximet}) \quad mod \quad 360 \tag{2.1}$$

donde ϕ_{maximet} indica la diferencia entre la orientación de la marca de la estación meteorológica y el norte de la boya y el *módulo 360* fuerza a que el ángulo esté entre 0 y 360 grados. El ángulo ψ_{maximet} es positivo en sentido horario y se mide desde el norte. Conocer la orientación de la boya es necesario para determinar la direccionalidad del campo de olas.

La tabla 2.1 muestra las diferentes variables registradas y sus unidades.

Sensores de la estación GMX-600

Variable	**Unidades**
Velocidad del viento relativa	$\frac{m}{s^2}$
Dirección del viento relativa	grados
Orientación	grados
Temperatura del aire	°C
Humedad relativa	%
Presión barométrica	mBar
Precipitación pluvial	mm

Cuadro 2.1: Variables registradas por la estación meteorológica GMX-600

2.3. Sistema de Adquisición de Datos

Un sistema de adquisición de datos (DAS por sus siglas en inglés) es un equipo dedicado a medir y registrar periódicamente diversas variables físicas (o ambientales) y transmitir la información al momento en caso necesario. Comúnmente está formado por una unidad de procesamiento y almacenamiento, módulos de adquisición de datos, de acondicionamiento de señal y de regulación de voltajes, transmisores de datos y en sitios remotos, un sistema de electrificación por energía renovable.

Las bases de datos generadas por estos sistemas se pueden utilizar para hacer estudios ambientales y climáticos así como para la elaboración de predicciones a partir de modelos numéricos.

El núcleo del sistema de adquisición de las BOMM es una computadora de a bordo cuya función principal consiste en solicitar los datos provenientes de los diferentes sensores, almacenarlos en memoria y prepararlos para ser enviados vía satélite. Dicha computadora ejecuta programas de adquisición desarrollados en lenguaje de programación Python y de transmisión de datos desarrollados en lenguaje de programación C++ bajo el sistema operativo linux.

2.3.1. Computadora de a bordo - Nitrogen6x

La Nitrogen6x (Boundary Devices) es una computadora de abordo con sistema operativo Linux para aplicaciones en sistemas embebidos basada en el procesador ARM-Cortex i.MX6 A9 de NXP/Freescale (http://www.nxp.com). Cuenta, entre otras cosas, con un bus de memoria amplio de 64 bits a 1066MHz tipo DDR3, interface HDMI integrado, Ethernet Gigabit y canales de visualización adicionales con un alto nivel de integración. La serie de procesadores de aplicaciones i.MX6 combina plataformas escalables con amplios niveles de integración y capacidades de procesamiento de alta eficiencia en energía, adecuados especialmente para aplicaciones multimedia.

Las especificaciones de hardware de la computadora Nitrogen6X son las siguientes:

- Procesador de 4 núcleos (Quad-Core ARM® Cortex A9) a 1GHz por núcleo
- Memoria RAM de 1GByte (64-bit wide DDR3 @ 532MHz)
- Tres puertos de pantalla (RGB, LVDS, y HDMI 1.4a)
- Dos puertos para cámara (1xParallel, 1x MIPI CSI-2)
- Multi-stream con capacidad de video de alta definición entregando decodificación H.264 de 1080p60, codificación 1080p30 y reproducción de video 3D en alta definición
- Sistema de gráficos Triple Play que consiste en una unidad Quad-shader 3D y una unidad separada 2-D, un motor de aceleración Vertex de OpenVG para aceleración superior en 3D, 2D e interfaz de usuario
- ATA 2.5 serie (SATA) a 3Gbps
- Ranura dual para tarjeta SD 3.0/SDXC
- Puerto PCIe
- Audio analógico (Audifono/mic) y Digital (HDMI)
- Ethernet 10/100 Gb
- Interfaz JTAG de 10-pin
- 3 puertos USB de alta velocidad (2xHost, 1xOTG)
- 1 puerto 1xCAN2
- Puerto de comunicación tipo I^2C
- Reloj de tiempo real (RTC) con batería de respaldo
- Puerto de E/S de propósito general para control de dispositivos

- Temperatura de operación 0°C-70°C (Versión Industrial para temperaturas de −40°C a +85°C)

Características eléctricas.

Parámetro	Valor
Voltaje de entrada	5V
Consumo de energía	1.5W
Reloj del CPU	1.0 GHz

Cuadro 2.2: Características eléctricas de la computadora de a bordo Nitrogen6x

Conexiones.

En las siguientes imágenes se muestra la computadora de a bordo Nitrogen6x y sus diferentes puertos para conexión de periféricos.

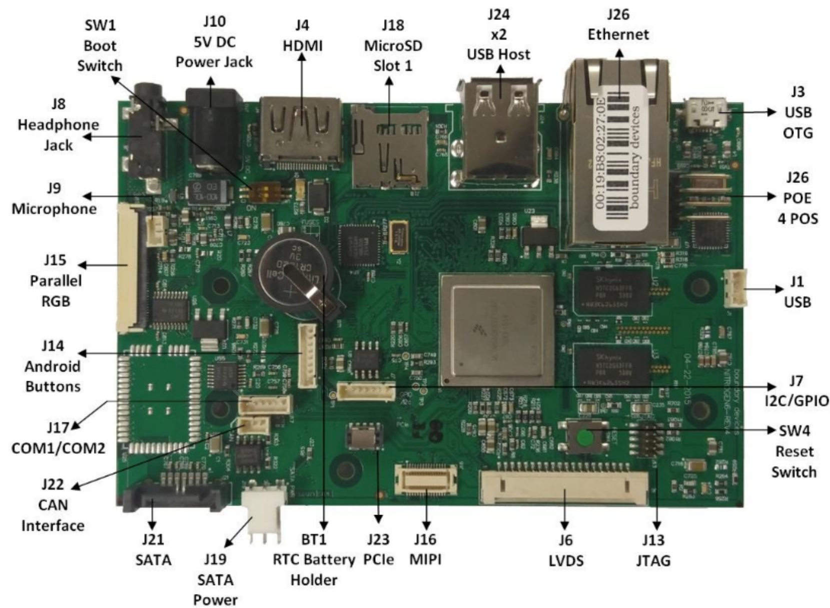


Figura 2.8: Microcomputadora de a bordo Nitrogen6x - (https://boundarydevices.com)

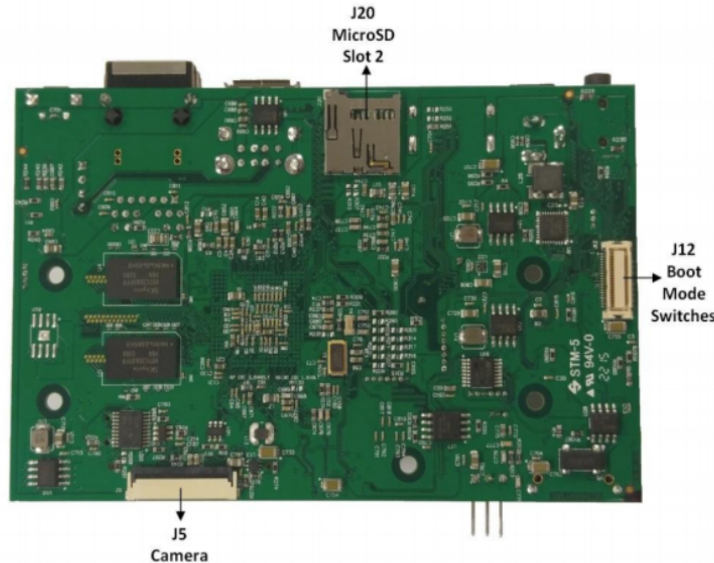


Figura 2.9: Microcomputadora de a bordo Nitrogen6x - (https://boundarydevices.com)

El voltaje de alimentación va en el conector J10 (5V DC).

Para acceder a la consola y realizar pruebas en laboratorio se puede conectar un monitor en el conector J4 (HDMI) mientras que en los puertos USB del conector J24 (USB Host) se pueden conectar un ratón y un teclado.

A través del conector J17 (COM1/COM2) se puede acceder al puerto serial de consola (COM1) para realizar modificaciones a la configuración y actualizar los programas en campo. Por otro lado, al utilizar el puerto COM2 se puede acceder al puerto serial RS232 nativo de la computadora y habilitar la comunicación serial con otro dispositivo.

En el conector J18 (MicroSD slot1) se aloja una memoria tipo microSD de 8GB que contiene tanto al kernel como a la distribución de Linux y los programas de adquisición de datos. En el conector J20 (MicroSD slot2) va insertada una memoria tipo microSD de 32 GB para almacenamiento primario de los datos.

Los conectores J19 y J21 se utilizan para suministrar voltaje y conectar el ducto de datos tipo SATA (respectivamente) de un disco duro de estado sólido de al menos 128 GB.

Los sensores y dispositivos externos de la boya se conectan a la computadora de a bordo a través de puertos USB utilizando convertidores USB-Serial con chipset de tecnología tipo FTDI.

2.4. Estimación del espectro direccional del oleaje (EDO)

Para estimar el espectro direccional del oleaje se utilizan las series de datos de la elevación de la superficie del mar que se obtienen a partir del conjunto de alambres de capacitancia.

Estimar el EDO consiste en obtener por una parte la matriz de densidad de energía $|W|^2$ en función de la frecuencia y el tiempo y por otra parte la matriz de direcciones Θ. La secuencia en el análisis de los datos con el fin de estimar el espectro direccional es la siguiente:

Control de calidad de los datos.

Para asegurar la confiabilidad en la estimación del espectro direccional se debe realizar al menos un control de calidad en cada conjunto de series de tiempo, que consiste en descartar las series temporales que no cuenten con al menos el 70 % de los datos esperados además eliminar los valores atípicos mediante el uso del criterio de Chauvenet (umbral de 2 desviaciones estándar).

Por otro lado, debido a que se presentan diferencias de precisión entre el reloj de la

computadora de a bordo y el reloj de cada uno de los diferentes sensores, es necesario realizar una interpolación de los datos para homogeneizar la tasa de muestreo a 20Hz aunque el sensor de movimiento tiene tasa de muestreo de 100 Hz.

Corrección de los datos por el movimiento de la boya.

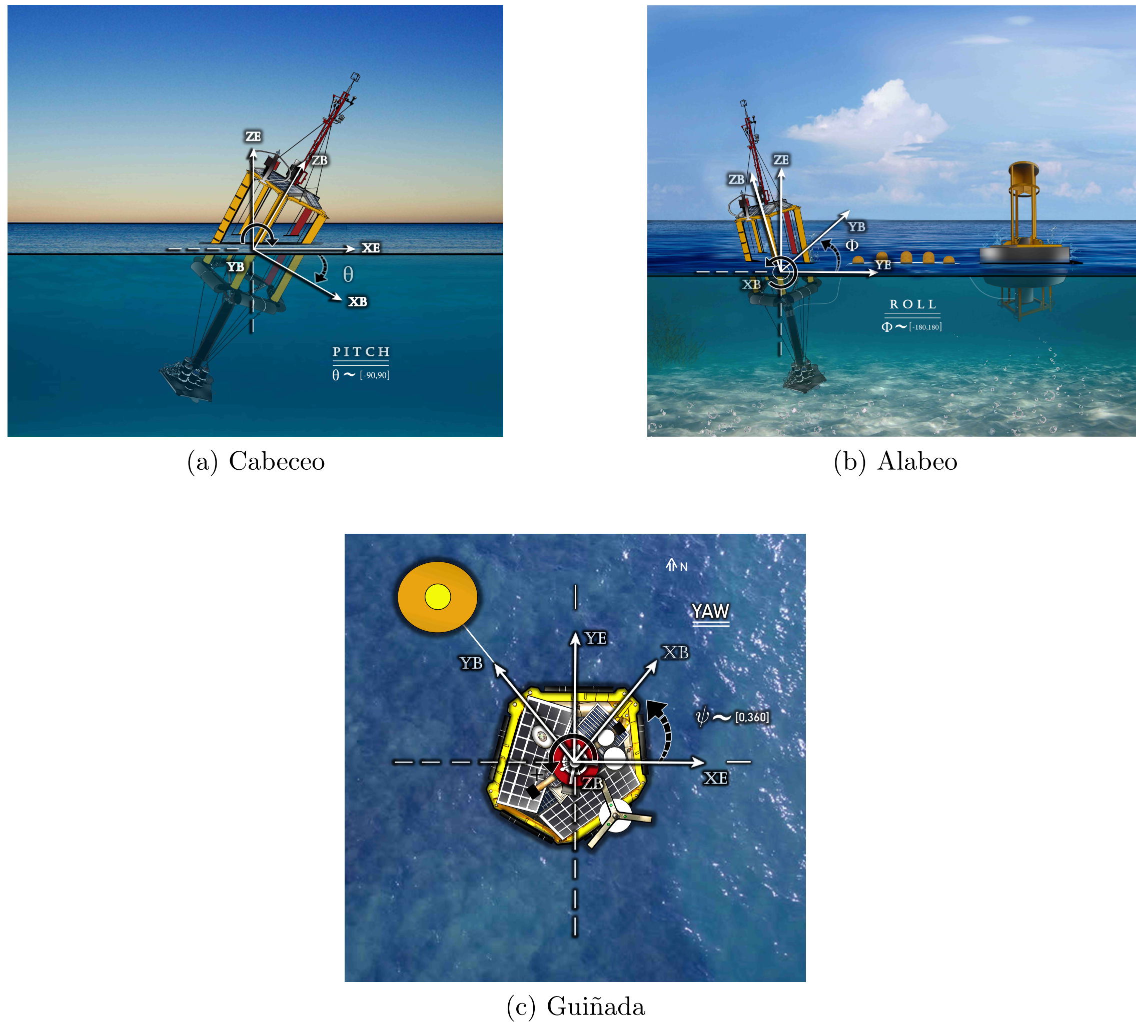
Los movimientos angulares de la boya alrededor de los ejes x y y inducen una señal incorrecta en la medición de la elevación de la superficie del mar, de la misma forma que el movimiento angular alrededor del eje z lo hace en el registro de la orientación de la boya, afectando directamente la estimación de la direccionalidad del campo de olas (ver fig. 2.10). Por esta razón es necesario llevar a cabo una corrección y transportar las mediciones a un sistema de referencia inercial. Los ángulos de rotación θ, ϕ y ψ corresponden con los movimientos de la boya denominados cabeceo, alabeo y guiñada respectivamente.

Un ***sistema de referencia inercial*** es un sistema en el cual un objeto en reposo permanece en reposo a menos que sobre él actúe una fuerza externa neta, es decir es un sistema que cumple con las leyes de movimiento de Newton. Cualquier sistema de referencia que se mueva con velocidad constante relativa a un sistema de referencia inercial es también un sistema de referencia inercial.

El vector de posición de la elevación de la superficie del mar en un alambre de capacitancia se define como $\mathbf{E} = (x, y, \eta)$, donde $x(t)$, $y(t)$ representan la posición del alambre y $\eta(t)$ representa la elevación de la superficie libre del mar. Este vector en el sistema de referencia inercial está dado por (Anctil et al., 1994; Drennan et al., 1994; Miller et al., 2008; García-Nava, 2011; Peláez-Zapata, 2018):

$$\mathbf{E} = \mathbf{T}\mathbf{E}_o + \mathbf{T}\iint \mathbf{a}\,dt\,dt + \mathbf{T}\int \mathbf{\Omega}\mathrm{x}\mathbf{L}\,dt \tag{2.2}$$

donde $\mathbf{T}$ es la matriz de rotación o de transformación de coordenadas, $\mathbf{E}_o$ contiene



(a) Cabeceo

(b) Alabeo

(c) Guiñada

Figura 2.10: Descripción de los ángulos de rotación provocados por los movimientos de cabeceo, alabeo y guiñada.
Las figuras son ilustrativas; las dimensiones de las boyas, la distancia de separación y las inclinaciones no corresponden con la escala real.

la información de la elevación de la superficie libre del mar observada en el sistema en movimiento, $\mathbf{a}$ es la matriz de aceleración registrada por el sensor de movimiento, $\boldsymbol{\Omega}$ representa las tasas de cambio de los movimientos angulares también registradas por el sensor de movimiento y $\mathbf{L}$ es la distancia entre el alambre de capacitancia y el sensor de movimiento.

Dado que el sensor de movimiento mide la aceleración y la tasa de cambio de los movimientos angulares, resulta necesario realizar una doble integral de $\mathbf{a}$ y una inte-

gral simple de $\mathbf{\Omega}$ para obtener los desplazamientos lineales y las velocidades angulares de la boya, respectivamente.

La matriz de rotación $\mathbf{T}$ está dada por:

$$\mathbf{T}= \begin{bmatrix} \cos\theta \cos\psi & \sin\phi \sin\theta \cos\psi - \cos\phi \sin\psi & \cos\phi \sin\theta \cos\psi + \sin\phi \sin\psi \\ \cos\theta \sin\psi & \sin\phi \sin\theta \sin\psi + \cos\phi \cos\psi & \cos\phi \sin\theta \sin\psi - \sin\phi \cos\psi \\ -\sin\theta & \sin\phi \cos\theta & \cos\phi \cos\theta \end{bmatrix}$$

donde $\theta(t)$, $\phi(t)$ y $\psi(t)$ representan los movimientos de alabeo, cabeceo y guiñada previamente descritos como los movimientos de rotación alrededor de los ejes x, y y z, respectivamente. De igual forma se definen los desplazamientos avance, arfada y deriva en las mismas direcciones.

Por último, la matriz de velocidades angulares $\mathbf{\Omega}$ está dada por:

$$\mathbf{\Omega} = \begin{bmatrix} -\frac{\partial\theta}{\partial t}\sin\psi + \frac{\partial\phi}{\partial t}\cos\theta\cos\psi \\ \frac{\partial\theta}{\partial t}\cos\psi + \frac{\partial\phi}{\partial t}\cos\theta\sin\psi \\ \frac{\partial\psi}{\partial t} - \frac{\partial\psi}{\partial t}\sin\theta \end{bmatrix}$$

Implementación del WDM

a) Transformada continua de Wavelet (CWT; por sus siglas en inglés)

El primer paso consiste en obtener la CWT de cada una de las series de tiempo correspondientes a la elevación de la superficie del mar $\eta(t)$ obtenidas con los alambres de capacitancia.

La CWT se define como la convolución entre una señal en el tiempo y una función que actúa como el núcleo de la transformada, la cual es trasladada un tiempo τ y

escalada un factor σ. Esta función es conocida como wavelet madre y se define como $W(\tau,\ \sigma)$. Donelan et al. (1996) proponen utilizar la wavelet madre de Morlet que consiste en una onda plana modulada por una campana de Gauss.

Debido a que las escalas de tiempo τ y de forma σ son directamente comparables con el tiempo t y la frecuencia f, podemos calcular la densidad de energía para cada frecuencia y para cada tiempo mediante:

$$P(f,t) = |W|^2 \ , \tag{2.3}$$

y la fase como:

$$\varphi(f,t) = \tan^{-1}\left(\frac{-\mathrm{Im}\{W\}}{\mathrm{Re}\{W\}}\right). \tag{2.4}$$

b) Número de onda $k = \frac{2pi}{L}$

El ***número de onda*** es una magnitud que indica el número de veces que vibra una onda en una unidad de distancia y es también conocido como el inverso de la longitud de onda. La superficie del mar, por ejemplo, puede representarse con la suma de un gran número de ondas, cada una con una frecuencia y un número de onda.

Mientras que la amplitud de una onda debe ser igual en un par de puntos m y n (con vectores posición x_m y x_n) del arreglo de sensores del nivel del mar, las diferencias que se detecta en los dos puntos corresponden con las diferencias entre las fases establecidas con el número de onda (k) correspondiente y están dadas por:

$$\Delta\varphi_{mn} = k{\cdot}x_m - k{\cdot}x_n \tag{2.5}$$

Usando el método de mínimos cuadrados se puede estimar el vector número de onda de la siguiente forma:

$$\boldsymbol{k} = (D^T D)^{-1} D^T \Delta\varphi \qquad (2.6)$$

donde **D** es la matriz de distancias obtenida del arreglo de sensores de la elevación del mar y $\boldsymbol{\Delta\varphi}$ es una matriz que contiene los desfases (diferencia de fases) entre las señales observadas por un par de alambres de capacitancia. Esta matriz de desfases se obtiene a partir de la transformada de wavelet.

c) Matriz de dirección Θ

El ***vector número de onda*** es un vector que apunta en la dirección de propagación de una onda y su magnitud es el número de onda. Utilizando las componentes del vector número de onda se puede estimar la dirección de propagación del oleaje de la siguiente manera:

$$\Theta(t, f) = \tan^{-1}(\frac{ky}{kx}) \qquad (2.7)$$

d) Espectro direccional del Oleaje

Una vez que se obtiene la matriz de densidad de energía $|W|^2$ y la matriz de direcciones $\Theta(t, f)$, el espectro direccional E(f,Θ) se obtiene promediando la energía asociada con una dirección determinada de cada componente espectral.

Para mayores detalles en el procedimiento y el análisis de los datos para estimar el espectro direccional del oleaje utilizando el método WDM, se recomienda revisar las tesis (García-Nava, 2011) y (Peláez-Zapata, 2018).

2.5. Transmisión de datos

Uno de los objetivos de este trabajo consiste en obtener información del estado del mar en tiempo real, con el fin de evaluar las condiciones del océano en el momento

y poder tomar decisiones adecuadas de manera rápida y efectiva.

Debido a la ubicación geográfica en la que se encuentran instaladas las boyas, es complicado (y en ocasiones imposible) utilizar una red de comunicación terrestre (GSM, GPRS, 3G o 4G) o de telemetría por RF, de tal manera que la única opción viable es utilizar telemetría satelital.

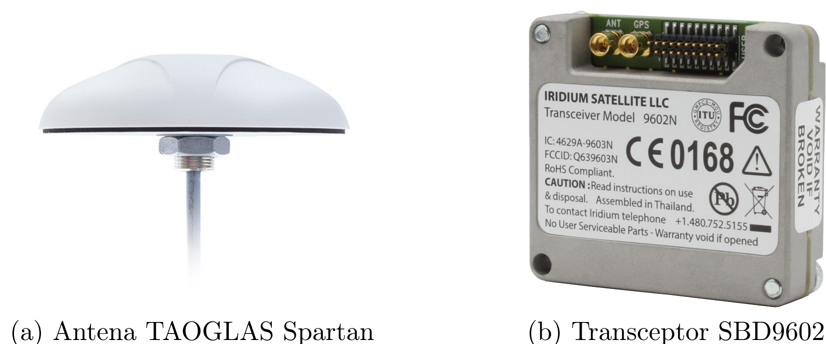
2.5.1. IRIDIUM

El sistema IRIDIUM se basa en una constelación de 66 satélites de órbita terrestre baja (LEO por sus siglas en inglés) a una altura aproximada de 780 km de la tierra (ver fig. 2.11). La constelación se distribuye en 6 órbitas y cada una consta de 11 satélites operativos equidistantes entre sí, así como un satélite de reserva.

Figura 2.11: Constelación LEO de Iridium

2.5.2. Tranceptor satelital - TAOGLAS Spartan STS.01

El sistema de telemetría de las boyas se basa en un transceptor Taoglas Spartan STS.01 que consiste en una avanzada antena de parche y un módem IRIDIUM de banda L modelo SBD9602 con interfaz de comunicación tipo RS-232. Todo esto dentro de un robusto encapsulado con grado de protección IP67 (ver fig. 2.12).



(a) Antena TAOGLAS Spartan (b) Transceptor SBD9602

Figura 2.12: Transceptor satelital para transmisión de los datos a través de la red IRIDIUM.

Este transceptor tiene la capacidad de enviar un mensaje de hasta 340 bytes cada 4 segundos (frecuencia máxima de envío de mensajes) y de recibir mensajes de hasta 270 bytes en la banda de frecuencias de 1616 MHz a 1626.5 MHz, con una potencia promedio de transmisión de 1 Watt.

Para comunicarse remotamente con el dispositivo, es necesario enviar un correo electrónico a la dirección de correo data@sbd.iridium.com que en el 'asunto' lleve el número de IMEI (International Mobile Equipment Identity) del transceptor destino y que además, contenga un archivo atado con extensión .sbd incluyendo la información deseada o comandos que queremos hacer llegar a la boya.

Los datos se reciben a través de una dirección IP fija (servidor principal) y además por correo electrónico y algunas otras direcciones de IP fijas (servidores redundantes) para asegurar el respaldo de la información y posteriormente los datos son procesados para su almacenamiento en el servidor y despliegue en un sitio web.

2.6. Sitio y campaña de mediciones

El grupo de oleaje del CICESE mantiene un sitio para pruebas del funcionamiento de las BOMM, localizado al noroeste de la Isla Todos B.C., México con coordenadas 31°49.332'N, 116°50.178'W (ver fig. 2.13).

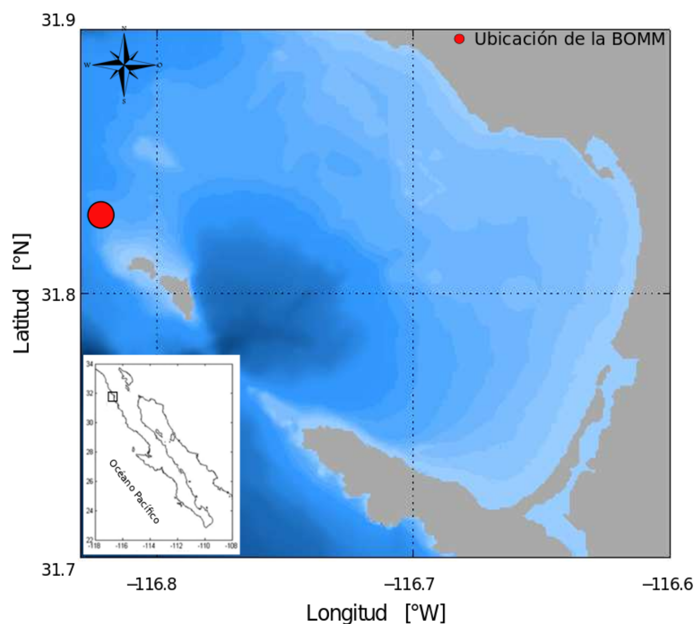


Figura 2.13: Localización del sitio para pruebas de flotabilidad y funcionamiento de las BOMM, al noroeste de la Isla Todos Santos, B.C.

Este sitio fue escenario de la instalación, pruebas y recuperación de la BOMM1 en el periodo del 17 de noviembre de 2017 al 02 de febrero de 2018. Los datos recabados en este periodo se utilizaron para el desarrollo del presente trabajo, particularmente los archivos de los días 17 y 18 de noviembre de 2017.

Capítulo 3

Implementación del método WDM y Resultados

3.1. Configuración de la computadora de a bordo

La computadora de a bordo Nitrogen6x soporta una variedad de opciones de sistema operativo frecuentemente solicitados en el espacio de diseño embebido. El sistema operativo elegido para realizar este trabajo es Ubuntu Trusty versión 14.04.4, debido a su robustez y con el fin de establecer compatibilidad con la configuración actual del sistema de adquisición de datos de las BOMM.

Para instalar y ejecutar el sistema operativo se requiere una memoria tipo micro SD de al menos 4GB de capacidad. Se utilizó una memoria de 8GB con formato ext4 y dos particiones de 4GB cada una. En la primer partición se ejecuta el sistema operativo mientras que en la segunda partición se aloja el ambiente de trabajo Miniconda, el lenguaje de programación Python y los programas de procesamiento y transmisión de datos.

Los datos de los diferentes sensores se almacenan en un disco duro externo de estado sólido con 4TB de capacidad. Los programas de adquisición de los datos que provienen de los diversos sensores no se desarrollaron en este trabajo, pero todos los

detalles pueden revisarse en Valenzuela et al. (2019).

- Ubuntu Trusty 14.04.4.

La instalación del sistema operativo Trusty se logra mediante la ejecución de los siguientes comandos:

```
ubuntu@nitrogen:~$ sudo umount /dev/sdX*
ubuntu@nitrogen:~$ zcat dir_imagen/20160330-nitrogen-3.14.xx.....armhf.img.gz
                   | sudo dd of=/dev/sdX bs=1M
ubuntu@nitrogen:~$ sync
```

Donde X corresponde a la letra de la unidad de almacenamiento (a, b, c...) en que se aloja la memoria dentro del kernel de la computadora de a bordo y dir_imagen corresponde al directorio en que se encuentra el archivo de la imagen de Linux con extensión .img.gz.

- Anaconda, Conda y Miniconda

Anaconda es un conjunto de paquetes que incluyen Conda, numpy, scipy, ipython notebook, etc. **Conda** es un gestor de paquetes (de código abierto) y a su vez, un sistema de gestión de entornos que se ejecuta en Windows, MacOS y Linux. Este gestor instala, ejecuta y actualiza rápidamente los distintos paquetes y sus dependencias. Además, crea, guarda, carga y conmuta fácilmente entre entornos dentro de una computadora local. Fue creado para programas Python, pero puede empaquetar y distribuir software en cualquier lenguaje de programación.

Por su parte, **Miniconda** contiene la versión mínima del gestor, incluyendo únicamente Conda y sus dependencias. Esto resulta una gran solución para trabajar en servidores, o incluso en máquinas menos poderosas en las que no se puede permitir

la utilización innecesaria de recursos.

- **Instalación de Miniconda**.

Existe una versión de Miniconda compatible con la arquitectura ARM del procesador de la computadora de abordo. Incluye la versión de Python 3.4.3-1 y se instala de la siguiente manera:

```
ubuntu@nitrogen:~$ bash miniconda3-latest-Linux-armv7l.sh
```

La implementación del software para estimación del espectro direccional del oleaje demanda ciertos módulos de Python (cuyas versiones sean compatibles con la arquitectura ARM) para realizar operaciones matemáticas avanzadas. A continuación se enlistan los módulos necesarios:

- **Blas**

  Los Subprogramas de Álgebra Lineal Básica (BLAS por sus siglas en inglés) son un conjunto de rutinas de bajo nivel para realizar operaciones de álgebra lineal comunes tales como suma de vectores, multiplicación escalar, productos punto, combinaciones lineales y multiplicación de matrices.

- **SciPy = 0.16.0**

  SciPy es un entorno de software de código abierto para matemáticas, ciencias e ingeniería basado en Python y es considerado como el ambiente (es decir, un módulo que contiene otros módulos) científico más completo.

- **Pandas = 0.16.2**

Pandas es una paquetería de código abierto que proporciona estructuras de datos de alto rendimiento que son fáciles de usar y herramientas de análisis de datos para el lenguaje de programación Python.

- **NumPy = 1.9.2**

  NumPy es el paquete fundamental para la computación científica con Python. Contiene, entre otras cosas:

  - Un poderoso objeto de matriz N-dimensional.
  - Funciones sofisticadas.
  - Herramientas para integrar código C/C++ y Fortran.
  - Gran capacidad de álgebra lineal, transformada de Fourier y números aleatorios.

  NumPy también puede ser utilizado como un eficiente contenedor multidimensional de datos genéricos. Se pueden definir tipos de datos arbitrarios. Esto permite que NumPy se integre sin problemas y rápidamente con una amplia variedad de bases de datos.

Una búsqueda exhaustiva con la ayuda del cliente-conda (conda-client) para localizar paquetes en anaconda.org identificó al usuario RPI como la opción viable para descarga de los módulos con versiones adecuadas.

La instalación de los módulos se lleva a cabo de la siguiente manera:

```
ubuntu@nitrogen:~$ dir/miniconda3/bin/conda install -c rpi blas
ubuntu@nitrogen:~$ dir/miniconda3/bin/conda install -c rpi scipy
ubuntu@nitrogen:~$ dir/miniconda3/bin/conda install -c rpi pandas
ubuntu@nitrogen:~$ dir/miniconda3/bin/conda install -c rpi numpy
```

Donde dir corresponde al directorio en donde se encuentra instalado el gestor Miniconda.

3.2. Estimación del Espectro Direccional del Oleaje

Los programas para estimar el EDO se desarrollaron en lenguaje de programación Python y se dividen en 4 módulos (ver fig. 3.1):

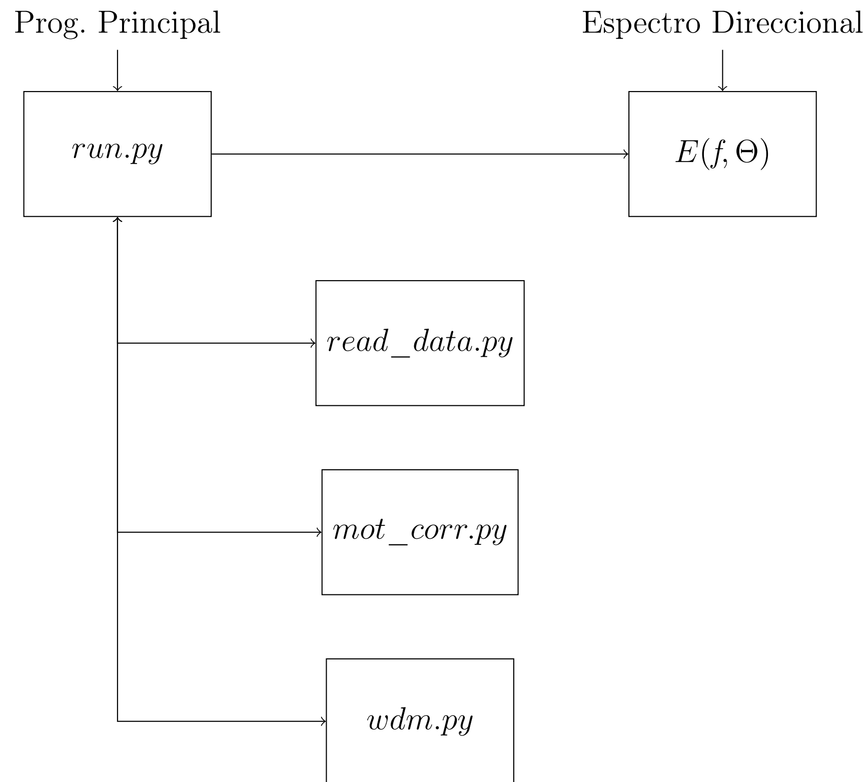


Figura 3.1: Diagrama a bloques de la programación para obtener el espectro direccional del oleaje.

En la figura 3.2 se presenta el diagrama de flujo del programa principal (run.py) que se utiliza para implementar el método WDM y obtener el espectro direccional del oleaje. En este mismo programa se ordena la ejecución de otros tres programas (read_data.py, mot_corr.py y edm.py) cuyos diagramas de flujo se muestran en la figura 3.3.

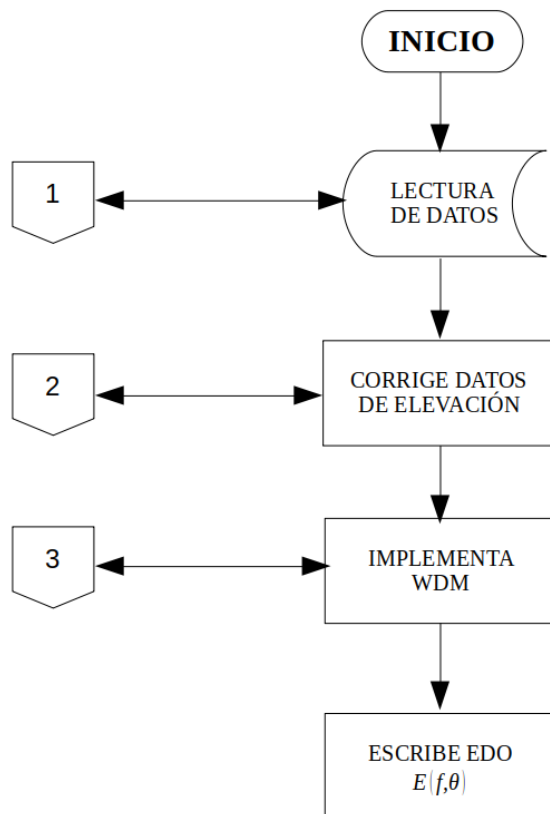


Figura 3.2: Diagrama de flujo del programa principal para obtener el EDO.

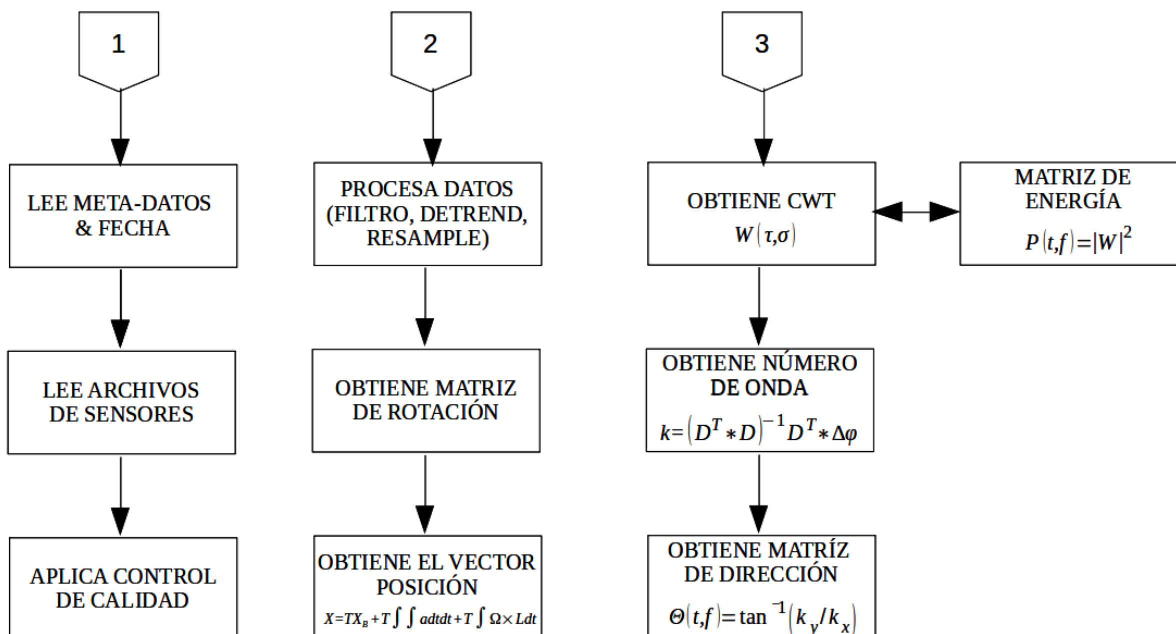


Figura 3.3: Diagrama de flujo de los programas secundarios para obtener el EDO.

A continuación se presenta una breve descripción de cada uno de los programas:

- **read_data.py**.

Programa para lectura de datos.
Inicia leyendo el archivo de meta-datos bomm1_its.yml que corresponde a la BOMM1 instalada en la Bahía Todos Santos durante el periodo de pruebas y contiene información de los diferentes sensores, por ejemplo la frecuencia de muestreo, número de serie, descripción y variables de medición. Posteriormente, se leen los archivos de datos de los distintos sensores y se define si cumplen con el criterio de calidad propuesto en la sección 2.4 del capítulo 2.

Nota: Los meta-datos son un conjunto de datos que describen el contenido informativo de un recurso, de archivos o de información de los mismos.

- **mot_corr.py**.

Este programa realiza la corrección adecuada a las mediciones del sensor de la elevación de la superficie del mar utilizando los datos del sensor de movimiento para transformarlas a un sistema de referencia inercial.

- **wdm.py**.

Programa para implementación del método WDM como se menciona en la sección 2.4 del capitulo 2.

- **run.py**.

Programa principal en el que se ejecutan los programas anteriores y se obtiene la matriz correspondiente al espectro direccional del oleaje. En este programa también se ejecutan subrutinas de adecuación de la matriz de datos para enviarla a través del satélite.

Al ejecutar el programa principal, se debe dar como argumento la fecha inicial del

periodo de datos que se va a procesar. Por defecto y para este trabajo, el programa automáticamente obtendrá el espectro direccional de los 10 minutos posteriores al minuto ingresado en el quinto argumento. Un ejemplo se muestra a continuación.

ubuntu@nitrogen:~$ dir/miniconda3/bin/python dir_programas/run.py 2017 11 17 00 00

Donde

- 2017 es el año.
- 11 es el mes.
- 17 es el día.
- 00 es la hora (múltiplo de 10).
- 00 es el minuto (múltiplo de 10).
- dir es el directorio en donde se encuentra el gestor Miniconda.
- dir_programas es el directorio en donde se encuentra el programa principal.

El resultado es una matriz con 72 líneas que corresponden a las direcciones del oleaje, 49 columnas que corresponden a las frecuencias y la información contenida en la matriz corresponde a la energía del oleaje. Dicha matriz se almacena en formato ASCII en un archivo con extensión .txt y además se genera el archivo binario correspondiente con extensión .npy.

El transceptor satelital tiene la capacidad de enviar hasta 340 bytes en cada transmisión. Dado que la matriz del EDO abarca aproximadamente 7.2kB, el archivo en binario se parte en "n" archivos de 330 Bytes cada uno con el fin de transmitirlos en secuencia. Para partir el archivo se ejecuta la siguiente línea:

>split -d -b 330 –additional-suffix=.npy test.spectrum.npy ../wdm/EDO_

Donde

- split es un comando del sistema que se utiliza para dividir un archivo en "n" partes.
- "-b 330" indica el número de bytes para cada archivo de salida (330 bytes).
- "-d" indica sufijos numéricos (iniciando en 0) posterior al nombre "EDO_"
- –additional-suffix=.npy agrega el sufijo .npy al final del nombre del archivo. - test.spectrum.npy es el archivo que contiene el EDO en binario.

De esta manera se generan 22 archivos de los cuales 21 abarcan un espacio de 330B y el último abarca los bytes restantes.

3.3. Algoritmo de comunicación bidireccional

3.3.1. Estación terrena

La ejecución de programas para estimar el espectro direccional del oleaje en la boya se realiza previa solicitud del usuario desde una estación terrena. Esto consiste en enviar un correo electrónico al proveedor de servicios *data@sbd.iridium.com*, que en la línea de asunto contenga el número de IMEI (International Mobile Equipment Identity) del transceptor destino y que además, lleve atado un archivo con extensión .sbd en el que se incluya la información que se quiere hacer llegar a la boya, es decir, la fecha y hora del periodo de datos del cuál se quiere obtener el EDO.

La figura 3.4 muestra un ejemplo del correo electrónico enviado al transceptor con IMEI cuya terminación es 2650.

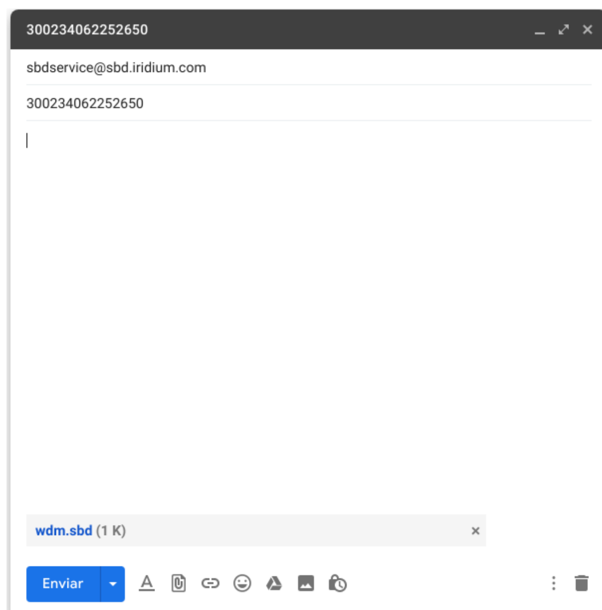


Figura 3.4: Información del correo electrónico.

Se observa en atado el archivo wdm.sbd que incluye la siguiente información:

17,11,17,00,00

Esto corresponde a obtener el EDO de los 10 minutos posteriores al día 17 de noviembre de 2017 a las 00:00.

3.3.2. Boya

La información enviada en el correo electrónico debe ser recibida e interpretada por la computadora de a bordo. Posteriormente se deben ejecutar los programas adecuados para obtener el EDO y finalmente transmitir los resultados desde la boya hasta el usuario a través del satélite.

La secuencia para configuración del transceptor, descarga e interpretación de instrucciones, estimación del espectro direccional y transmisión del EDO es la siguiente:

- Configuración del transceptor satelital

La configuración del módem se basa en el diagrama de flujo mostrado en la figura 3.5. Al inicio del programa se configuran los registros del módem (baudrate, tipo de respuesta, etc.) y se almacenan los cambios en la localidad 0 para que se mantengan fijos cada reinicio del transceptor.

Posteriormente se habilita la función "alertas de anillo" (ring alert) al ejecutar el comando AT+SBDMTA (Short Burst Data: Mobile-Terminated Alert) = 1. Esto permite obtener una alerta en la BOMM cuando haya un nuevo mensaje de la estación terrena en el satélite.
También es necesario registrar el módem con la ayuda del comando AT+SBDREG (Short Burst Data: Network Registration).
El Registro en la red realiza dos funciones:
1. Notificar a la puerta de enlace de IRIDIUM que el módem está configurado y listo para recibir alertas de anillo.
2. Proporcionar las coordenadas de geolocalización necesarias para que la puerta de enlace de IRIDIUM sepa a dónde enrutar la(s) alerta(s) de anillo. Una vez configurado el módem, se procede a la rutina para estimar el espectro direccional del oleaje.

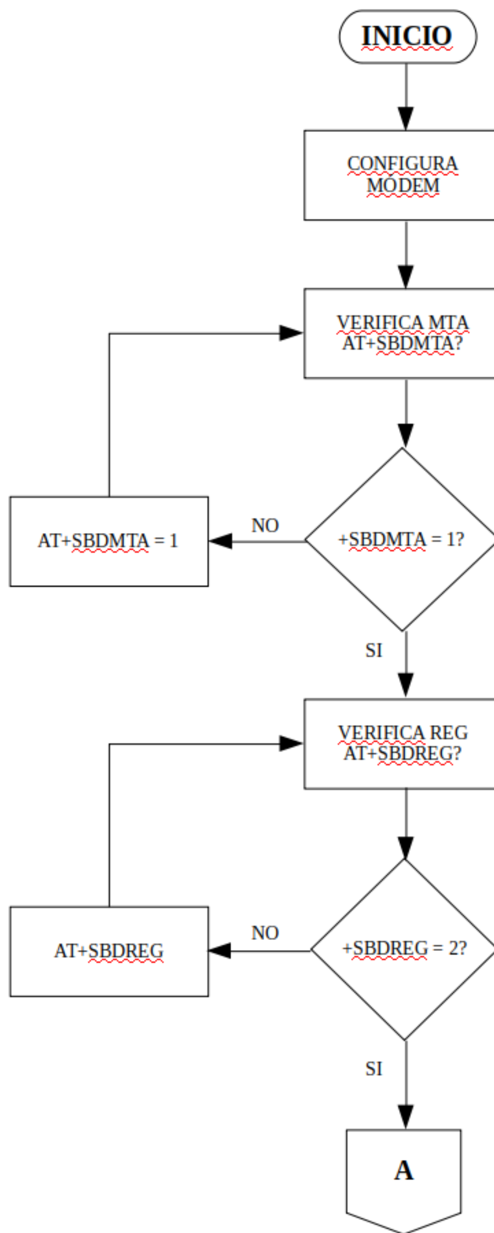


Figura 3.5: Diagrama de flujo para la configuración del transceptor.

- Estimación del espectro direccional del oleaje

La figura 3.6 corresponde al diagrama de flujo para realizar la estimación del EDO. Una vez configurado el transceptor, la computadora de a bordo se mantiene en espera de una solicitud por parte del usuario monitoreando periódicamente el estado del sa-

télite con la ayuda del comando AT+SBDSX (Short Burst Data: Status Extended). Si el sexto caracter en la línea de respuesta es igual a 1, indica que existe un nuevo mensaje en la puerta de enlace de IRIDIUM e inicia la descarga de la información del satélite al transceptor mediante el comando AT+SBDI (Short Burst Data: Initiate an SBD Session), seguido del comando AT+SBDRT (Short Burst Data: Read a Text Message from the Module) para descargar la información del transceptor a la computadora de a bordo. Si no existe un nuevo mensaje, el programa espera un minuto e intentará nuevamente.

Una vez que se haya recibido y descifrado la información, la computadora de a bordo ejecuta el programa de lectura (read_data.py) de los datos que correspondan al periodo solicitado, realiza el control de calidad necesario y ejecuta los programas mot_corr.py y wdm.py para obtener la matriz correspondiente al EDO, convertirla a binario y dividirla en 22 segmentos como se mencionó en la sección 3.2 de este capítulo. Ya que se tienen los 22 segmentos, el programa pasa al modo de transmisión, cuyo diagrama de flujo se muestra en la figura 3.7.

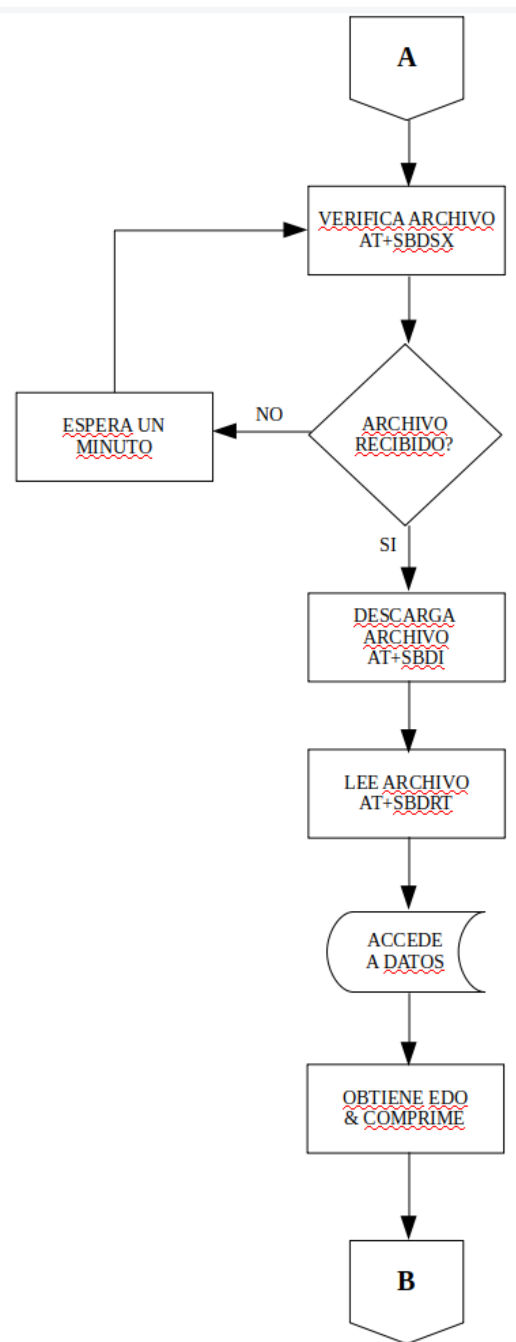


Figura 3.6: Diagrama de flujo para estimar el EDO.

- Transmisión del EDO a través de la red de satélites de IRIDIUM

Cada segmento de la matriz se almacena secuencialmente en la memoria del módem con ayuda del comando AT+SBDWB (Short Burst Data: Write Binary Data to the ISU) y se envía hacia la estación terrena al ejecutar el comando AT+SBDI

(Short Burst Data: Initiate an SBD Session) siempre que la intensidad de la señal de comunicación entre el transceptor y el satélite sea igual o mayor a 3* (+CSQ: 3). Este dato se obtiene mediante el comando AT+CSQ (Signal Quality).

Una vez enviados todos los mensajes, se limpia la memoria del módem ejecutando el comando AT+SBDD1 (Short burst data: Clear SBD message buffers) e inicia el ciclo de espera de un nuevo mensaje.

* Cada número representa una mejora de aproximadamente 2 dB. en el margen del enlace con respecto al valor anterior. Una lectura de 0 es igual o inferior al nivel mínimo de sensibilidad del receptor. Una lectura de 1 indica aproximadamente 2 db de margen de enlace. Una lectura de 5 indica un margen de enlace de 10 dB o más.

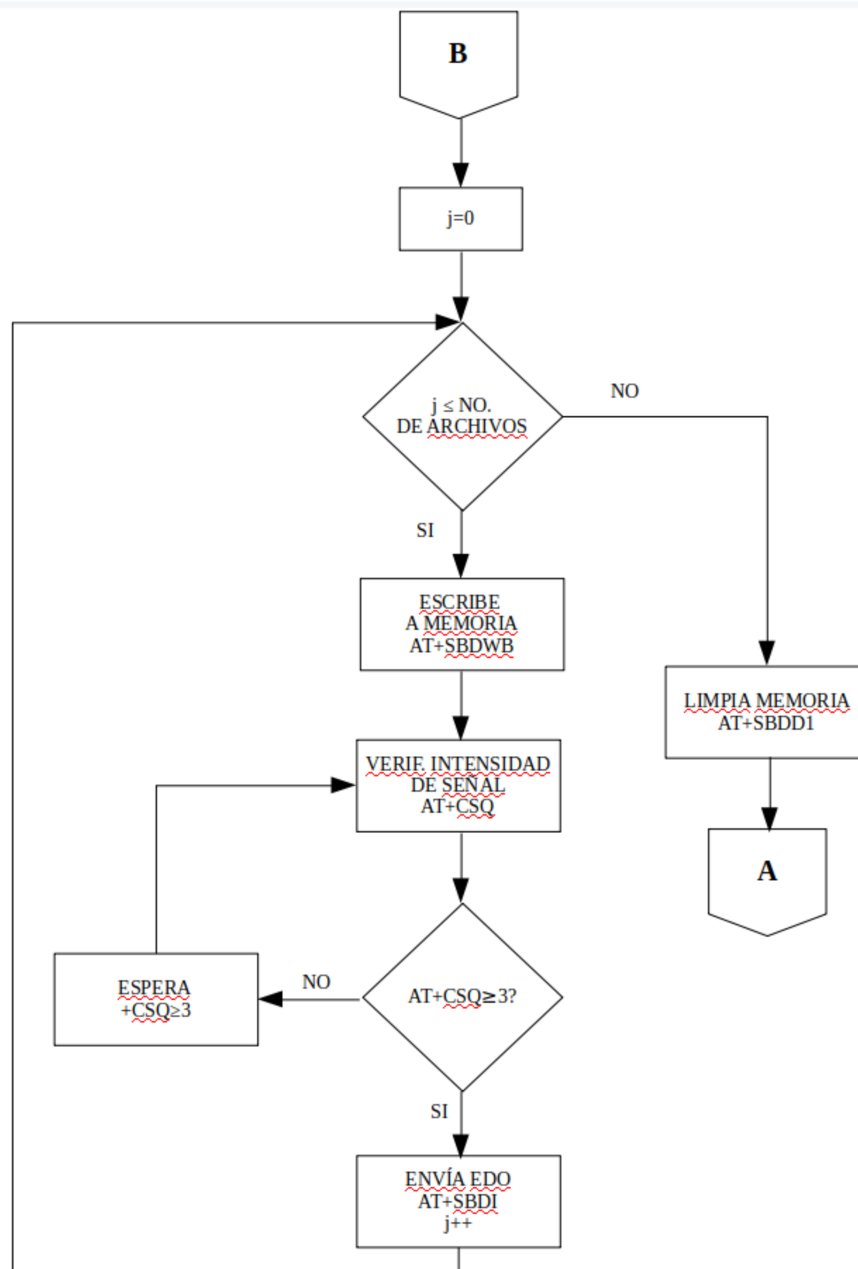


Figura 3.7: Diagrama de flujo para la transmisión del EDO.

3.4. Recuperación y despliegue del EDO

El procedimiento para descarga de mensajes en la computadora de la estación terrena consiste en ejecutar un programa que escucha permanentemente el puerto TCP 8100 y agrega cada línea de mensaje recibida a un archivo con nombre EDO_xx.npy.

Donde xx incrementa de 0 al número de líneas recibidas y corresponde a la posición del segmento dentro de la matriz del EDO.

Para recuperar la matriz del espectro direccional se unen todos los archivos anteriores con ayuda del comando cat, de la siguiente manera:

```
$ cat EDO* > EDO.npy
```

El nuevo archivo EDO.npy es una copia fiel del archivo original generado en la computadora de a bordo y se utiliza para recuperar la matriz en formato ASCII con la ayuda de la subrutina np.load() de NumPy.

Para el despliegue del espectro direccional se desarrolló un programa en MatLab que grafica el espectro al transformar la matriz del EDO de coordenadas polares a coordenadas cartesianas y además muestra en colores la energía y su distribución en las diferentes frecuencias y direcciones.

En la figura 3.8 se presenta el espectro direccional del oleaje de dos fechas particulares, una del día 2017-11-17 a las 10:00 (izquierda), y la otra el día 2017-11-18 a las 10:00 (derecha) obtenidos con el programa previamente mencionado. En ambos casos se observa energía distribuida en dos sistemas de oleaje. Uno poco energético y de periodo corto ($\sim$3 seg.), considerado como oleaje generado localmente, que va evolucionando del día 17 al 18 de noviembre y otro grupo más energético y de periodo largo ($\sim$10 seg.) considerado como oleaje generado por tormentas lejanas. El oleaje de ambos sistemas se propaga hacia el noroeste.

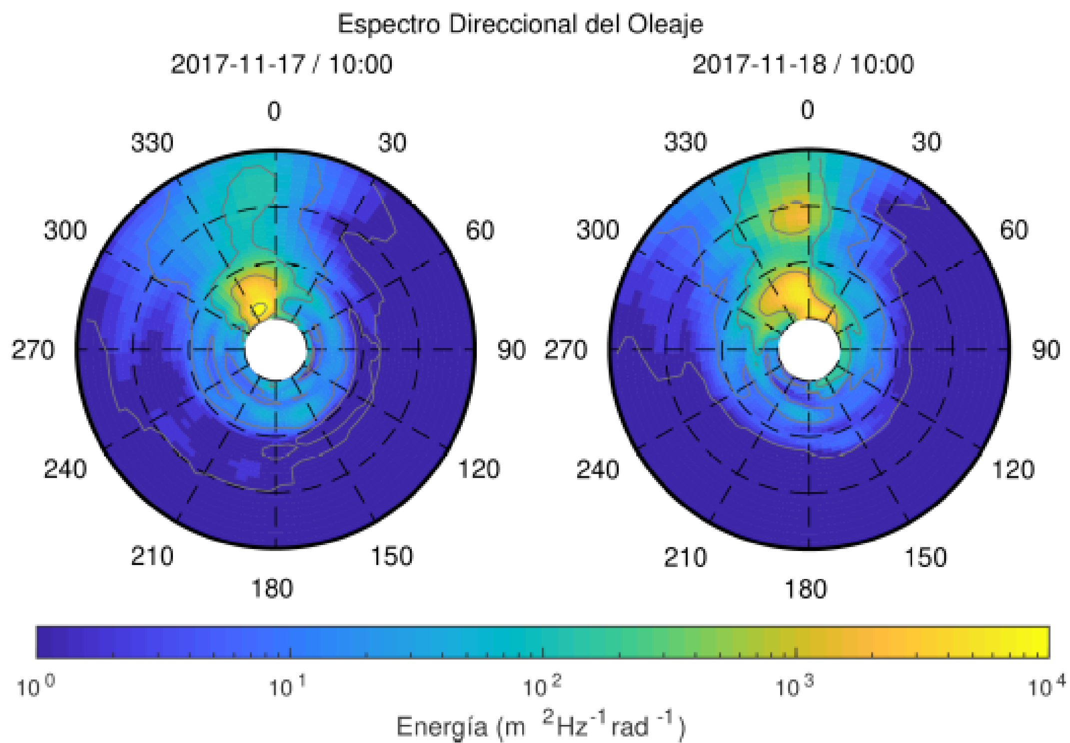


Figura 3.8: Espectros direccionales del oleaje del día 2017-11-17 10:00 (izquierda) y el día 2017-11-17 10:00 (derecha). Los círculos representan a las frecuencias .1Hz, .2Hz, .3Hz y .4Hz. La barra de colores indica la densidad de energía y sus unidades son: $\frac{m^2}{Hz\ rad}$

Capítulo 4

Conclusiones

En este trabajo se presenta un sistema para estimar el espectro direccional del oleaje (EDO) implementando el método direccional de wavelets (WDM) a partir de las mediciones realizadas con diversos sensores instalados en una Boya Oceanográfica y de Meteorología Marina. Se desarrollaron programas en lenguaje de programación Python para procesamiento de datos e implementación del método, los cuales fueron ejecutados en una computadora de a bordo. Todas las pruebas fueron realizadas en laboratorio, sin embargo, los datos utilizados para calcular el espectro direccional del oleaje fueron datos reales que se obtuvieron de una BOMM desplegada en el campo. La implementación del método in situ permite identificar la direccionalidad de las olas al momento, así como la distribución de la energía del oleaje en diferentes frecuencias y direcciones.

Los resultados obtenidos se transmitieron en tiempo real a través de telemetría satelital desde una computadora de bordo hasta una estación terrena. Para ello se diseñó un algoritmo bidireccional de transmisión de datos con la ayuda de un transceptor satelital de IRIDIUM. El algoritmo se puede adaptar fácilmente a otras aplicaciones de acceso remoto mediante la adecuación necesaria que conlleva algunos cambios mínimos. La transmisión de datos se logra satisfactoriamente con las características técnicas del transceptor elegido, sin embargo, se sugiere el uso de un transceptor que permita el envío de una mayor cantidad de bytes por transmisión y

que además, cuente con un mayor ancho de banda. Los programas de transmisión y recepción se desarrollaron en lenguaje de programación C++ y se ejecutan dentro de la computadora de a bordo. Debido a la posible falta de sincronía entre la transmisión y recepción de mensajes, se propone agregar un indicador de posición así como una suma de verificación (checksum) al final de cada línea de mensaje transmitido y recuperar la información original en la etapa de recepción. La transmisión se realiza a través de satélite debido a que la estimación del EDO se lleva a cabo en plataformas boyantes instaladas en sitios remotos, en donde el uso de comunicaciones convencionales como telefonía celular, internet o radiofrecuencia no está disponible.

Se desarrolló un programa en lenguaje de programación M (Matlab) para despliegue del EDO y se obtuvieron resultados a partir de los datos de los días 17 y 18 de noviembre de 2017 obtenidos cerca de la Isla Todos Santos, Ensenada, B.C. La ejecución de este programa es manual por lo que se sugiere el diseño de un ambiente gráfico que permita automatizar todas las tareas de recepción y graficado. Los espectros mostrados en los resultados de esta tesis fueron obtenidos y graficados con los programas realizados en este trabajo y ejecutados dentro de las computadoras de a bordo y de la estación terrena respectivamente.

El desarrollo de este trabajo permite comprobar que los avances tecnológicos en la actualidad posibilitan realizar estudios científicos que requieren gran capacidad de almacenamiento y procesamiento de datos en sitios remotos. Esto es muy importante para proveer de información valiosa a la comunidad científica e ingenieril que actualmente demanda una gran cantidad de datos para entender los distintos procesos físicos de la naturaleza, en particular de la oceanografía. De igual manera, el uso de telemetría satelital facilita el acceso a sitios lejanos de tal manera que se puede obtener información en tiempo real para realizar análisis detallados al momento y por periodos largos.

En síntesis, el objetivo general fue cubierto casi en su totalidad, faltando únicamen-

te realizar pruebas in situ implementando el sistema en una BOMM. Los objetivos específicos se lograron en su totalidad, al construir un sistema de adquisición de datos en laboratorio, ejecutar programas para estimar el espectro direccional del oleaje dentro de la computadora de a bordo, transmitir el EDO a través de la constelación de satélites de IRIDIUM y desplegarlo en una estación terrena.

Como actividades a futuro, se sugiere instalar los programas desarrollados en este trabajo en una BOMM, con el fin de realizar pruebas ya sea en la vecindad de Isla de Todos Santos, Ensenada B.C. o en el Golfo de México y transmitir los datos en tiempo real. También se sugiere el reemplazo del módem por uno con mayor capacidad de transmisión y ancho de banda y además, se propone agregar un indicador de posición así como una suma de verificación (checksum) al final de cada línea de mensaje transmitido y recuperar la información original en la etapa de recepción. Se sugiere implementar un control de calidad de los datos más robusto que permita evaluar condiciones particulares de la boya, por ejemplo, la ausencia de sensores de elevación del nivel del mar por mal funcionamiento y se propone el diseño de un ambiente gráfico que permita automatizar todas las tareas de recepción y graficado del EDO en la estación terrena.

Bibliografía

Anctil, F., Donelan, M. A., Drennan, W. M., and Graber, H. C. (1994). Eddy-correlation measurements of air-sea fluxes from a discus buoy. *Journal of Atmospheric and Oceanic Technology*, 11(4):1144–1150.

Capon, J. (1979). Maximum-likelihood spectral estimation. *In Nonlinear methods of spectral analysis*, 34:155–179.

Castro, A., Martínez-Osuna, J., Michel, R., Escoto-Rodriguez, M., Bullock, S. H., Cueva, A., López-Reyes, E., j. Reimer, Salazar, M., Villarreal, S., and Vargas, R. (2017). A low-cost modular data-acquisition system for monitoring biometeorological variables. *Esevier: Computers and Electronics in Agriculture*, 141:357–371.

Davis, R. E. and Regier, L. (1977). Methods for estimating direc- tional wave spectra from multi-element arrays. *J. Mar. Res.*, 35:453–477.

Donelan, M., Babanin, A., Sanina, E., and Chalikov, D. (2015). A comparison of methods for estimating directional spectra of surface waves. *JGR: Oceans*, 120:5040–5053.

Donelan, M. A., Drennan, W. M., and Magnusson, A. K. (1996). Nonstationary analysis of the directional properties of propagating waves. *J. Phys. Oceanogr.*, 120:5040–5053.

Drennan, W. M., Donelan, M. A., Madsen, N., Katsaros, K. B., Terray, E. A., and Flagg, C. N. (1994). Directional wave spectra from a swath ship at sea. *Journal of Atmospheric and Oceanic Technology*, 11:1109–1116.

Fleisch, D. and Kinnaman, L. (2015). *A Student's Guide to Waves.* Cambridge University Press, University Printing House, Cambridge CB2 8BS, UK.

García-Nava, H. (2006). Evaluación del flujo de momento entre la atmósfera y el océano bajo diferentes condiciones de oleaje. Master's thesis, CICESE, Ensenada, México.

García-Nava, H. (2011). *Efecto del oleaje en la capa límite atmosférica marina.* PhD thesis, CICESE, Ensenada, México.

Graber, H. C., Terray, E. A., Donelan, M. A., Drennan, W. M., Leer, J., and Peters, D. B. (2000). Asis—a new air–sea interaction spar buoy: 0design and performance at sea. *Journal of Atmospheric and Oceanic Technology*, 17:708–720.

Hanson, K. A., Hara, T., Bock, E. J., and Karachintsev, A. B. (1997). Estimation of directional surface wave spectra from a towed research catamaran. *Journal of Atmospheric And Oceanic Technology*, 14:1467–1482.

Holthuijsen, L. H. (2007). *Waves in Oceanic and Coastal Waters.* Cambridge University Press, The Edinburgh Building, Cambridge CB2 8RU, UK.

Longuet-Higgins, M. S., Cartwright, D. E., and Smith, N. D. (1963). Observations of the directional spectrum of sea waves using the motions of a floating buoy. *In Ocean Wave Spectra, proceedings of a conference*, pages 111–136.

Lygre, A. and Krogstad, H. E. (1986). Maximum entropy estimation of the directional distribution in ocean wave spectra. *J. Phys. Oceanogr.*, 16:2052–2060.

Miller, S. D., Hristov, T. S., Edson, J. B., and Friehe, C. A. (2008). Platform motion effects on measurements of turbulence and air–sea exchange over the open ocean. *Journal of Atmospheric and Oceanic Technology*, 25(9):1683–1694.

Ocampo-Torres, F. J., García-Nava, H., Durazo, R., Osuna, P., Méndez, G. M., and Graber, H. C. (2010). The intoa experiment: A study of ocean-atmosphere interactions under moderate to strong offshore winds and opposing swell conditions in the gulf of tehuantepec, méxico. *Boundary-Layer Meteorology*, 138:433–451.

Peláez-Zapata, D. S. (2018). Efecto del oleaje en la capa límite superficial del océano. Master's thesis, CICESE, Ensenada, México.

Takahashi, N., Ishihara, Y., Ochi, H., Fukuda, T., Tahara, J., Maeda, Y., Kido, M., Ohta, Y., Mutoh, K., Hashimoto, G., Kogure, S., and Kaneda, Y. (2014). New buoy observation system for tsunami and crustal deformation. *Mar Geophys Res*, 35:243–253.

Torrence, C. and Compo, G. P. (1998). A practical guide to wavelet analysis. *Bulletin of the American Meteorological Society*, 79:61–78.

Valenzuela, E., Orozco, O., Ortiz, E., Ojeda, R., Rodríguez, C. E., Ceseña, J. A., and Ocampo-Torres, F. J. (2019). *Descripción del software para la adquisición, almacenamiento y transmisión satelital de las boyas oceanográficas*. CICESE, Carretera Ensenada - Tijuana No. 3918. Zona Playitas.

Vlachos, D. S. and Tsabaris, C. (2007). The use of vertical and horizontal accelerations of a floating buoy for the determination of directional wave spectra in coastal zones. *Elsevier: Mathematical and Computer Modelling*, 48:1949–1956.

ANEXO 1.

- - run.py - -

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
# vim:fenc=utf-8
#
# Copyright © 2019 Daniel Santiago <dpelaez@cicese.edu.mx>
# Copyright © 2019 Juan Mtz-Osuna <jmartine@cicese.mx>
#
# Distributed under terms of the GNU/GPL license.

"""

"""
import numpy as np
import scipy.signal as signal
import datetime as dt
import yaml
import sys
import os
import os.path
from os import path
#
import wdm as wdm
import motion_correction as motcor
from read_data import ReadRawData


class ProcessingData(object):

    """
    This class contains funcion to get the directional wave spectra from the
    BOMM SSD using the Wavelet Directional Method (WDM)

    Usage:

        # define the path of the metadata file
        >> metafile = "../metadata/bomm1_its.yml"


        # define the specific date
        >> date = dt.datetime(int(sys.argv[1]), int(sys.argv[2]), int(sys.argv[3]))
        #>> date = dt.datetime(2017,11,17,0,0)

        # create instance of the main class
        >> p = ProcessingData(metafile, date)

        # load the data, perform motion_correction and get_spectrum
        >> p.read_data()
        >> p.get_directional_spectrum()
        >> p.write_directional_spectrum()

        # [optional] get the wave parameters
        >> p.get_wave_parameters()
    """

    # private methods {{{
    def __init__(self, metafile, date):
        """Function to initialize the class.

        Args:
            metafile (str): filename containing the metadata
```

```
        date (datetime): datetime object
    """

    self.metafile = metafile
    self.date = date

    # load metadata
    with open(metafile, "r") as f:
        self.metadata = yaml.load(f)


# check for nans in input variables
def _check_nans(self, dic, limit=0.3):
    """Check if some of our dictionaries has more nans than wanted."""

    valid = True
    for k,v in dic.items():
        try:
            number_of_nans = len(np.nonzero(v.mask)[0])
            if (number_of_nans / len(v)) >= limit:
                valid = False
                return valid
        except AttributeError as e:
            pass
    return valid

# }}}

# read data {{{
def read_data(self):
    """Function to read the raw data from the BOMM SDD."""

    # create instance of ReadRawData
    r = ReadRawData(self.metafile)

    # load data as dictionaries
    self.ekx = r.read("ekinox", self.date)
    self.wav = r.read("wstaff", self.date)
    self.sig = r.read("signature", self.date)
    self.met = r.read("maximet", self.date)
    self.gps = r.read("gps", self.date)

    self.results = {}
# }}}


# wavenumber {{{
def wavenumber(self, f, d=100, mode="hunt"):
    """
    mode = "exact"
    --------------
    Calculo del numero de onda usando la relacion de dispersion de la teoria
    lineal resuelta con un metodo iterativo (punto fijo) sin tener en cuenta
    el efecto doppler de las corrientes en la frecuencia de las olas.
          2
        w    =   g * k * tanh(k*h)    --->    w  = 2*pi*f

    mode = "hunt"  (default)
    ------------------------
    Calculo del numero de onda usando la aproximacion empirica propuesta por Hunt 1979
              2           2            y
        (kh)    =     y   +  --------------------
                                   6
                                  ----     n
                           1 + \    d   y
```

```
                        /     n
                       ----
                       n = 1
              2
        y = w  h  / g

        d0 = [0.666, 0.355, 0.161, 0.0632, 0.0218, 0.0065]
       """

    if d < 0:
        raise ValueError("Depth must be positive")

    if mode == "exact":
        #
        tol = 1e-9
        maxiter = 1000000
        g = 9.8
        w = 2.* np.pi * f
        k0 = (w**2.)/g
        for cnt in range(maxiter):
            k = (w**2)/(g*np.tanh(k0*d))
            k0 = k
            if all(abs(k - k0) >= tol):
                return k0
        return k

    elif mode == "hunt":
        #
        d0 = [0.666, 0.355, 0.161, 0.0632, 0.0218, 0.0065]
        g = 9.8
        w = 2.* np.pi * f
        y = (w**2)*d/g
        #
        poly = np.zeros_like(f)
        for n, dn in enumerate(d0):
            poly = poly + dn * y**(n+1)
        #
        k = np.sqrt(y**2 + y/(1 + poly))/d

        return k
        #
    else:
        raise ValueError("`mode` must br `hunt` o `exact`")
# }}}

# fourier spectrum {{{
def welch(self, x, fs, nfft=512, overlap=128):
    """Computes the Fourier periodograms ignoring segments with NaNs."""

    # check how if all data is nan
    n = len(x)
    nans = len(np.where(np.isnan(x))[0])
    if n == nans:
        raise Exception("Array is full of NaNs.")

    # loop for each segment
    S = []
    for j in np.arange(0, n-nfft+overlap, overlap):
        arr = x[j:j+nfft]
        nans = len(np.where(np.isnan(arr))[0])
        if nans == 0 and len(arr) == nfft:
            f, S0 = signal.welch(arr, fs, window="hann", nperseg=nfft)
            S += [S0]

    return f, np.mean(S, axis=0)
```

```
# }}}

# compute wave parameters {{{
def wave_parameters(self, frqs, dirs, E):
    """"Return basic bulk wave parameters from the directional wave spectrum."""

    # TODO: check for nans

    # integrate directional wave specrtum
    dirs = np.radians(dirs)
    S_int = np.trapz(E, x=dirs, axis=0)
    D_int = np.trapz(E, x=frqs, axis=1)

    # computes m,n oder moments of the spectrum
    # indices <<mn>> represents the exponents of f and S respectivaly
    m = lambda n, p: np.trapz((frqs**n)*(S_int**p), x=frqs)

    # compute basic parameters
    Hm0 = 4.0 * np.sqrt(m(0,1))
    Tp1 = m(0,4) / m(1,4)
    #
    # compute directional params
    m = lambda n, p: np.trapz((dirs**n)*(D_int**p), x=dirs)
    # pDir = np.mod(dirs[np.argmax(D_int)], 2*np.pi)
    pDir = m(1,4) / m(0,4)
    mDir = m(1,1) / m(0,1)

    return Hm0, Tp1, np.degrees(pDir), np.degrees(mDir)

# }}}

# compute stokes drift profile {{{
def stokes_drift(self, f, S, z=-np.logspace(-5,2,50)):
    """Compute stokes drift profile as Breivik et al 2016 eq5."""

    # angular frequency and spectrum in right units
    g = 9.8
    k = self.wavenumber(f, 100)
    w = 2*np.pi * f
    Sw = S / (2*np.pi)

    fac = 2 / g
    if isinstance(z, float) or isinstance(z, int):
        dummy = w**3 * Sw * np.exp(2*k*z)
    else:
        dummy = w[None,:]**3 * Sw[None,:] * np.exp(2*k[None,:]*z[:,None])
    return np.trapz(fac*dummy, w)

# }}}


# compute buoy heading {{{
def compute_heading(self):
    """Compute the heading from different sources. Return heading in deg."""

    # TODO: when magnetometre will be available choose it as default
    if hasattr(self, "mag"):
        pass

    # heading signature
    heading_sig = (self.sig["heading"]/100) % 360

    # heading maximet
    maximet_angle = self.metadata["sensors"]["maximet"]["maximet_angle"]
```

```
    true_wnd, rel_wnd = self.met["true_wind_dir"], self.met["relative_wind_dir"]
    heading_met = (true_wnd - rel_wnd + maximet_angle) % 360

    # the low frequency heading means the angle between new BOMM y-axis and
    # true north. Magnetic deviation is taken from GPS measurements. All in
    # degrees, The mag deviation or declination is added to the current
    # magnetic mesurement ---> (check this, im not pretty sure)
    if np.isnan(heading_sig).all():
        heading = heading_met - self.gps["mag_var"][0] * 0
    else:
        heading = heading_sig - self.gps["mag_var"][0] * 0

    return heading % 360
# }}}

# motion matrices {{{
def motion_matrices(self):
    """Matrices of the accelerometer, gyroscope and euler angles"""

    # check for anomalous data
    for k, v in self.ekx.items():
        if k not in ["time"]:
            self.ekx[k][abs(v) > 1E5] = np.nan

    # fill nans when possible
    for k, v in self.ekx.items():
        if k not in ["time"]:
            number_of_nans = np.isnan(v).sum()
            if (number_of_nans / len(v)) < 0.1:
                self.ekx[k][np.isnan(v)] = np.nanmean(v)

    # construct accelerometer and gyroscope tuples
    # apply a rotation to an ENU frame of reference
    R = (np.pi, 0, np.pi/2)
    self.Acc = motcor.vector_rotation((self.ekx["accel_x"],
        self.ekx["accel_y"], self.ekx["accel_z"]), R)
    #
    self.Gyr = motcor.vector_rotation((self.ekx["gyro_x"],
        self.ekx["gyro_y"],  self.ekx["gyro_z"]),  R)

    # integrate accel and gyro to obtain euler angles
    ax, ay, ax = self.Acc
    gx, gy, gz = self.Gyr
    phi, theta = motcor.pitch_and_roll(
            ax, ay, ax, gx, gy, ax, fs=100, fc=0.04, fm=1)
    #
    # compute bomm heading and the merge with ekinox
    heading = np.radians((90 - self.compute_heading()) % 360)
    psi = motcor.yaw_from_magnetometer(self.Gyr[2], heading,
            fs=100, fc=0.04, fm=0.04)
    #
    # TODO: from BOMM3 the eknox was updated to output euler angles
    #       so we need choose phi and theta directly and psi from the
    #       combination between the magnetometre and the ekinox
    self.Eul = (phi, theta, psi)
# }}}

# get directional spectrum {{{
def get_directional_spectrum(self):
    """Compute directional wave spectrum and other wave parameters."""

    # check for nans: if both are valid do nothing, else raise exception
    if (self._check_nans(self.ekx) and self._check_nans(self.wav)):
        pass
    else:
```

```
    raise Exception("Number of nans is more than 30%")

# # if heading is greater than 90 degress return an error
# std_yaw = np.nanstd(self.Eul[2]) * 180/np.pi
# if std_yaw > 15:
    # raise Exception(f"BOMM has veered too much: {std_yaw:.2f} deg")

# TODO: check wavestaff standar deviation

# get motion matrices
self.motion_matrices()

# check waestaffs in use
valid_wires = self.metadata["sensors"]["wstaff"]["valid_wires"]
valid_wires_index = [w - 1 for w in valid_wires]

# check dimensions
nfft = 1024
npoint = 6
ntime = len(self.wav["time"])

# determinte position of the wavestaffs
# TODO: since offset depends on each bomm, they should be passed
#       as an input argument.
x_offset, y_offset, z_offset = -0.339, 0.413, 4.45
xx, yy = wdm.reg_array(N=5, R=0.866, theta_0=180)
# xx, yy = xx + x_offset, yy + y_offset

# get the sampling frequency and the resampling factor
fs = self.metadata["sensors"]["wstaff"]["sampling_frequency"]
q = int(100/fs)

# allocate variables
S = np.zeros((int(nfft/2+1), npoint))
X, Y, Z = (np.zeros((ntime, npoint)) for _ in range(3))
#
# apply the correction to the surface elevation and compute fourier spc
for i, (x, y), in enumerate(zip(xx, yy)):
    #
    # get suface elevation at each point
    z = self.wav["ws%d" % (i+1)] * 3.5/4095 + z_offset
    #
    # apply motion correction
    # fs, q = 20, 5 # BOMM1
    X[:,i], Y[:,i], Z[:,i] = motcor.position_correction((x,y,z),
            self.Acc, self.Eul, fs=fs, fc=0.04, q=q)
    #
    # compute fourier spectrum
    # TODO: compute spectrum with homemade pwelch, it will allow to
    # discard the blocks containing nan data.
    ffrq, S[:,i] = self.welch(Z[:,i], fs=fs, nfft=nfft, overlap=int(nfft/4))

# limit to the half of the nfft to remove high frequency noise
S = np.mean(S[1:int(nfft/4)+1,valid_wires_index], axis=1)
ffrq = ffrq[1:int(nfft/4)+1]

# compute directional wave spectrum
# TODO: create a anti-aliasing filter to decimate the time series
dfac = 2
d = lambda x: x[::dfac,1:]
wfrq, dirs, E, D = wdm.fdir_spectrum(d(Z), d(X), d(Y), fs=int(fs/dfac),
        omin=-4, omax=-1, nvoice=16, ws=(30, 4))

# save data in the output dictionary
list_of_variables = {
```

```
            "ffrq" : "fourier_frequencies",
            "S"    : "frequency_spectrum",
            "wfrq" : "wavelet_frequencies",
            "dirs" : "wavelet_directions",
            "E"    : "directional_wave_spectrum",
            }
        #
        for k, v in list_of_variables.items():
            self.results[k] = eval(k)
    # }}}

    # get wave parameters {{{
    def get_wave_parameters(self):
        """Compute wave paramemters for a given directional wave spectrum."""

        # get data from results dictionary
        # TODO check if self.get_directional_spectrum was called
        E = self.results["E"]
        S = self.results["S"]
        ffrq = self.results["ffrq"]
        wfrq = self.results["wfrq"]
        dirs = self.results["dirs"]

        # compute bulk wave parameters and stokes drift magnitude
        Hm0, Tp, pDir, mDir = self.wave_parameters(wfrq, dirs, E)
        Us0 = self.stokes_drift(ffrq, S, z=0.0)

        # save data in the output dictionary
        list_of_variables = {
            "Hm0"  : "significant_wave_height",
            "Tp"   : "peak_wave_period",
            "pDir" : "peak_wave_direction",
            "mDir" : "average_wave_direction",
            "Us0"  : "surface_stokes_drift"
            }
        #
        for k, v in list_of_variables.items():
            self.results[k] = eval(k)
    # }}}

    # write directional specrtum {{{
    def write_directional_spectrum(self):
        """Write directional spectrum to a file in the disk"""

        E = np.uint16(self.results["E"][::5,:] * 1E4)
        # np.savetxt("test.spectrum.txt", E, fmt='%d')
        np.savetxt("test.spectrum.txt", E, fmt='%4d')
        np.save("test.spectrum", E)
        np.savez_compressed("test.spectrum", E)

        if path.exists("test.spectrum.npy") == True:
            os.system('split -d -b 330 --additional-suffix=.npy test.spectrum.npy ../wdm/EDO_')
            print ("Archivo existente \n")
            os.system("./TaoglasTX")
        else:
            print ("Archivo inexistente \n")

    # }}}



if __name__ == "__main__":

    # define the path of the metadata file
```

```
# Las siguientes 3 lineas son para obtener argumentos
#conc = '../metadata/' + sys.argv[1]
#metafile = conc
#print ("Metafile es:" + metafile)

metafile = "../metadata/bomm1_its.yml"
#conc = '../metadata/' + 'sys.argv[1]'

# define the specific date
date = dt.datetime(int(sys.argv[1]), int(sys.argv[2]), int(sys.argv[3]))
#date = dt.datetime(2017,11,17,0,0)

# create instance of the main class
p = ProcessingData(metafile, date)

# load the data, perform motion_correction and get_spectrum
p.read_data()
p.get_directional_spectrum()
p.write_directional_spectrum()

# [optional] get the wave parameters
p.get_wave_parameters()
```

- - read_data.py - -

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
# vim:fenc=utf-8
# vim:fdm=marker
#
# ================================================================
#  Copyright © 2018 Daniel Santiago <dpelaez@cicese.edu.mx>
# Copyright © 2019 Juan Mtz-Osuna <jmartine@cicese.mx>
#  Distributed under terms of the GNU/GPL license.
# ================================================================

""
""

# --- import libs ---
import numpy as np
import datetime as dt
import pandas as pd
import csv
import sys
import yaml


# === Read Data Class ===
class ReadRawData(object):

    ""
    This class contains methods to read the csv files written in the SSD for the
    CICESE-BOMM (Oceanographic and Marine Meteorology Buoys).
    ""

    __slots__ = ["metadata", "basepath", "bomm_name"]

    # private methods to read data {{{
    def __init__(self, metafile):
        """Function to initialize the class.


        Args:
            metafile (string): Name of the YAML file containg BOMM metadata.

        Usage:
            import bomm
            metafile = "bomm.yml"
            b = bomm.ReadRowData(metafile)
            wav = b.read(sensor="wstaff", date=dt.datetime(2017,11,17,0,0))

        Sensors:
            ekinox, sonic, gps, maximet, proceanus,
            rbr, signature, vector, wstaff


        TODO:
            - [ ] create function to read UCMR.

        ""

        self.metadata = self._readmetadata(metafile)
        self.basepath = self.metadata["basepath"]
        self.bomm_name = self.metadata["name"]


    # parse hour line into floating
```

```
def _parsedate(self, string):
    """Parse the date HH:MM:SS.ffffff YYYY mm dd."""

    hour, dd, mm, yy = string.split()
    H, M, sec = hour.split(":")
    try:
        S, f = sec.split(".")
    except ValueError:
        S, f = sec, '0'
    return  dt.datetime(int(yy), int(mm), int(dd), int(H), int(M), int(S), int(f))



# get file name from a given date
def _getfilename(self, sensor, date):
    """Returns filename based on date, sensor, and path."""

    if sensor in ["maximet"]:                   # <--- one-hour files
        fmt = "/{0}/%Y/%m/%d/{0}-%y%m%d%H.csv".format(sensor)
    #
    elif sensor in ["rbr", "proceanus"]:        # <--- one-day files
        fmt = "/{0}/%Y/%m/{0}-%y%m%d.csv".format(sensor)
    #
    else:                                       # <--- one-minute files
        fmt = "/{0}/%Y/%m/%d/%H/{0}-%y%m%d%H%M.csv".format(sensor)

    filename = self.basepath + self.bomm_name + \
            "/data" + dt.datetime.strftime(date, fmt)
    return filename


# read metadada
def _readmetadata(self, metafile):
    """Read the YAML file containg metadata."""

    with open(metafile, "r") as f:
        return yaml.load(f)


# read sampling frequency
def _getsampfreq(self, sensor):
    """Returns the sampling frequency for specific sensor."""

    sensors = self.metadata["sensors"][sensor]
    try:
        fs = float(sensors["sampling_frequency"])
    except ValueError as e:
        num, den = sensors["sampling_frequency"].split("/")
        fs = float(num) / float(den)

    return fs



# read seconds per file
def _getsecperfile(self, sensor):
    """Returns the seconds per file for specific sensor."""

    sensors = self.metadata["sensors"][sensor]
    return int(sensors["seconds_per_file"])



# read columns
def _getcolumns(self, sensor):
    """Returns the columns to be read for specific sensor."""
```

```
        sensors = self.metadata["sensors"][sensor]
        return {k:v["column"] for k,v in sensors["variables"].items()}

    # convert to float if you can
    def _float(self, x):
        """Convert string to float even if it has double decimal point."""

        try:
            return float(x)
        except ValueError:
            #
            # if value has double decimal
            if len(x.split(".")) >= 2:
                return np.nan
            #
            # if value is a letter
            elif x.isalpha():
                return x
            #
            # finally, return nan
            else:
                return np.nan

    # read file
    def _readfile(self, sensor, date, columns):
        """This function reads the data of the original files."""

        # get filename
        filename = self._getfilename(sensor, date)

        # pre-define variables
        time = []
        obs = {v: [] for v in columns.keys()}

        # TODO: check if file has null bytes

        # read file
        with open(filename, 'r') as f:
            data = csv.reader(f, delimiter=',')
            for irow, row in enumerate(data):
                #
                # parse date and append it to a list
                time.append(self._parsedate(" ".join(row[:4])))
                #
                # loop for each column
                for k, v in columns.items():
                    #
                    # check if data is a 2d array
                    if isinstance(v, list):
                        a, b = v
                        obs[k].append([self._float(s) for s in row[a:b+1]])
                    #
                    # if not, then it is a 1d array
                    else:
                        try:
                            #
                            # try to convert each value into float
                            obs[k].append(self._float(row[v]))
                        except IndexError:
                            #
                            # if line is empty then fill with nans
                            obs[k].append(np.nan)
                        # except ValueError:
                            # #
```

```
                    # # if conversion fails then leave as string
                    # obs[k].append(row[v])
                except:
                    raise Exception("Problems reading data")

        # convert to numpy array and add time array
        obs["time"] = time
        for k, v in obs.items():
            obs[k] = np.asarray(v)

        return obs


    # new time array
    def _getnewtime(self, date, fs, N):
        """Returns list of datetimes of N seconds at fs sampling frequency."""

        seconds = np.arange(0, N, 1/fs)
        return np.array([date + dt.timedelta(seconds=s) for s in seconds])


    # return invalid file
    def _returninvalid(self, sensor, date):
        """Returns NAN with the same structure as the valid data."""

        N = self._getsecperfile(sensor)
        fs = self._getsampfreq(sensor)
        columns = self._getcolumns(sensor)

        data = {"time": self._getnewtime(date, fs, N)}
        for k, v in columns.items():
            if isinstance(v, list):
                data[k] = np.full((len(data["time"]), v[1]-v[0]+1), np.nan)
            else:
                data[k] = np.full(len(data["time"]), np.nan)

        return data


    # interpolate the data
    # TODO: I need to improve this code function
    def _resample(self, dic, date, fs, N=600):
        """This function uses pandas for an accurate resample"""

        # check minimum sampling frequency
        # fs = np.max((fs, 1./600.)) # force to be one data each ten minutes

        # create new time array
        time_new = self._getnewtime(date, fs, N)

        # create pandas dataframe using the data in the dictionary
        t = dic["time"]
        df = pd.DataFrame({k:v for k,v in dic.items()
                           if k not in ["time"]}, index=t)

        # check time discontinuities
        max_gap = np.diff(dic["time"]).max().total_seconds()
        if max_gap > 5 / fs:
            pass

        # check the number of nans
        n = len(t)
        n_nans = df.isnull().sum().max()
        if n_nans / n > 0.1:
```

```
            raise Exception("Number of NaNs is %d out of %d." % (n_nans, n))

        # drop missing values only if are less than 10 percent of the data
        # TODO: limit to a number of consecutive nans of one second
        df = df.dropna(how="any")

        # remove duplicate indices if they exist
        # df = df[~df.index.duplicated(keep='first')]

        # sort data in ascending and reindex to the new time
        # perform the reindex twice, one backward and one foreward
        # limit to a number of consecutive nans of one second
        # l = int(fs) if fs>=1 else 1
        l = 1
        df = df.sort_index().reindex(time_new, limit=l, method="bfill").ffill()

        # crate new dictionary for output
        outdic = {c:df[c].values for c in df}
        outdic["time"] = time_new
        return outdic


    # remove NULL bytes
    def _remove_nullbytes(self, filename):
        """Remove NULL bytes in file represented by '\x00'."""

        with open(filename, "r") as f:
            data = f.read()

            if data.find("\x00") != -1:
                f.write(data.replace("\x00", ""))

            return f
    # }}}

    # read ekinox {{{
    def ekinox(self, date):

        # parameters
        fs = self._getsampfreq("ekinox")
        N = self._getsecperfile("ekinox")
        columns = self._getcolumns("ekinox")

        # load observed data
        obs = self._readfile("acelerometro", date, columns)

        # return data
        return self._resample(obs, date, fs, N)
    # }}}

    # read sonic {{{
    def sonic(self, date):

        # parameters
        fs = self._getsampfreq("sonic")
        N = self._getsecperfile("sonic")
        columns = self._getcolumns("sonic")

        # load observed data
        obs = self._readfile("anemometro", date, columns)

        # return data
        return self._resample(obs, date, fs, N)
    # }}}
```

```
# read gps {{{
def gps(self, date):

    # parameters
    fs = self._getsampfreq("gps")
    N = self._getsecperfile("gps")
    columns = self._getcolumns("gps")

    # load observed data
    obs = self._readfile("gps", date, columns)

    # assign sign to latitud or longitude
    sign = np.vectorize(lambda s: -1 if s in ["S", "W"] else 1)
    obs["latitude"] *= sign(obs["lat_sign"])
    obs["longitude"] *= sign(obs["lon_sign"])

    # remove items
    for a in ["status","lon_sign", "lat_sign"]:
        obs.pop(a)

    # return data
    return self._resample(obs, date, fs, N)
# }}}

# read marvi {{{
def marvi(self, date):

    # parameters
    fs = self._getsampfreq("marvi")
    N = self._getsecperfile("marvi")
    columns = self._getcolumns("marvi")

    # load observed data
    obs = self._readfile("marvi", date, columns)

    # return data
    return self._resample(obs, date, fs, N)
# }}}

# read maximet {{{
def maximet(self, date):

    # parameters
    date = date.replace(minute=0)
    fs = self._getsampfreq("maximet")
    N = self._getsecperfile("maximet")
    columns = self._getcolumns("maximet")

    # load observed data
    obs = self._readfile("maximet", date, columns)

    # return data
    return self._resample(obs, date, fs, N)
# }}}

# read proceanus {{{
def proceanus(self, date):

    # parameters
    date = date.replace(hour=0, minute=0)
    fs = self._getsampfreq("proceanus")
    N = self._getsecperfile("proceanus")
    columns = self._getcolumns("proceanus")
```

```
    # load observed data
    obs = self._readfile("proceanus", date, columns)

    # return data
    return self._resample(obs, date, fs, N)
# }}}

# read rbr {{{
def rbr(self, date):

    # parameters
    date = date.replace(hour=0, minute=0)
    fs = self._getsampfreq("rbr")
    N = self._getsecperfile("rbr")
    columns = self._getcolumns("rbr")

    # load observed data
    obs = self._readfile("rbr", date, columns)

    # return data
    return self._resample(obs, date, fs, N)
# }}}

# read signature {{{
def signature(self, date):

    # parameters
    fs = self._getsampfreq("signature")
    N = self._getsecperfile("signature")
    columns = self._getcolumns("signature")

    # load observed data
    obs = self._readfile("signature", date, columns)

    # get active beams and number of cells
    beams = self.metadata["sensors"]["signature"]["beams"]
    ncell = self.metadata["sensors"]["signature"]["ncell"]

    # resample velocity, amplitude and correlation
    for beam in beams:
        for letter in ["vel", "amp", "cor"]:
            variable = "{0}_b{1}".format(letter, beam)
            arr = obs[variable]
            for i in range(ncell):
                obs["%s_c%02d" % (variable, i+1)] = arr[:,i]
            #
            # remove arr
            obs.pop("%s" % (variable))

    # resample data
    dic = self._resample(obs, date, fs, N)

    # return to two-dimensional arrays
    for beam in beams:
        for letter in ["vel", "amp", "cor"]:
            variable = "{0}_b{1}".format(letter, beam)
            arr = np.full((len(dic["time"]), ncell), np.nan)
            for i in range(ncell):
                arr[:,i] = dic["%s_c%02d" % (variable, i+1)]
                dic.pop("%s_c%02d" % (variable, i+1))
            #
            dic[variable] = arr

    return dic
```

```
    # }}}

    # read vector {{{
    def vector(self, date):

        # parameters
        fs = self._getsampfreq("vector")
        N = self._getsecperfile("vector")
        columns = self._getcolumns("vector")

        # load observed data
        obs = self._readfile("vector", date, columns)

        # return data
        return self._resample(obs, date, fs, N)
    # }}}

    # read wavestaff {{{
    def wstaff(self, date):

        # parameters
        fs = self._getsampfreq("wstaff")
        N = self._getsecperfile("wstaff")
        columns = self._getcolumns("wstaff")

        # load observed data
        obs = self._readfile("wstaff", date, columns)

        # apply correction factor of 3.5/4095
        # for k in columns.values():
            # obs[k] *= 3.5/4095

        # return data
        return self._resample(obs, date, fs, N)
    # }}}

    # read any sensor {{{
    def read(self, sensor, date, logfile=sys.stderr):
        """Wrapper function to handle errors when reading data for each sensor."""

        function = eval("self.%s" % sensor)
        try:
            # read data
            return function(date)

        # if file does not exist or not valid
        except Exception as e:
            error = "%10s ---> {%s} : %s" % (sensor, date, e)
            print(error, file=logfile)
            return self._returninvalid(sensor, date)
    # }}}



if __name__ == "__main__":

    pass

# === end of file ===
```

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
# vim:fenc=utf-8
# vim:fdm=marker
#
# ================================================================
#  Copyright © 2018 Daniel Santiago <dpelaez@cicese.edu.mx>
#  Copyright © 2019 Juan Mtz-Osuna <jmartine@cicese.mx>
#  Distributed under terms of the GNU/GPL license.
# ================================================================

"""
This module contains functions to apply the correction of the
data due to the BOMM inertial motion.
"""


import numpy as np
import scipy.signal as signal
import pandas as pd


# --- helper functions ---
# butterworth lowpass and highpass filter {{{
def butterworth_filter(data, fs=100, fc=None, order=5, kind="low"):
    """Butterworth low- and hihg-pass filter implementation."""

    if fc is not None:
        b, a = signal.butter(order, fc/(0.5*fs), btype=kind, analog=False)
        data_filtered = signal.filtfilt(b, a, data)
        if hasattr(data, "mask"):
            return np.ma.masked_array(data_filtered, mask=data.mask)
        else:
            return data_filtered
    else:
        return data
# }}}

# detrend with no nans {{{
def detrend(y, degree=1):
    """Nice detrend function to handle with NaNs."""


    # remove nans
    if hasattr(y, "mask"):
        if len(np.nonzero(y.mask)[0]) == len(y):
            raise Exception("Array is full of nans")
    else:
        ixnan = np.isnan(y)
        y[ixnan] = np.nanmean(y)

    # fit a polinomial
    x = np.linspace(0, len(y), len(y))
    p = np.polyfit(x, y, degree)

    # remove trend
    return y - np.polyval(p, x)
# }}}

# resample data to the same sampling frequency {{{
def resample(x, y):
    """Interpolates `y` data into `x` size and returns `y_new`
```

```
    This function uses pandas for an accurate resample. The `x` data is the
    fast signal and the `y` data is the slow signal. This function interpolates
    `y` data into the `x` data.

    """

    x_time = np.linspace(0, 100, len(x))
    y_time = np.linspace(0, 100, len(y))

    s = pd.Series(y, index=y_time)

    # check the number of nans
    n = len(y)
    n_nans = s.isnull().sum().max()
    if n_nans / n > 0.1:
        raise Exception("Number of NaNs is %d out of %d" % (n_nans, n))

    # drop missing values only if are less than 10 percent of the data
    # TODO: not more than 100 consecutive missing values
    s = s.dropna(how="any")

    # remove duplicate indices if they exist
    # s = s[~s.index.duplicated(keep='first')]

    # sort data in ascending and reindex to the new time
    # i still dont know what is the difference between ffill/bfill
    s = s.sort_index().reindex(x_time, limit=1, method="bfill").ffill()

    # crate new dictionary for output
    if hasattr(x, "mask"):
        return np.ma.masked_array(s.values, mask=x.mask)
    else:
        return s.values
# }}}

# complementery filter in using a digital filter {{{
def complementary_filter(signal_a, signal_b, fs, fm, filter_order=2):
    """"Returns a merge between two angles signal_a and signal_b in radians."""


    # merge the signals using the merge frequency fm
    s = butterworth_filter(np.exp(1j*signal_a), fs, fm, filter_order, kind="low") + \
        butterworth_filter(np.exp(1j*signal_b), fs, fm, filter_order, kind="high")

    return np.angle(s)

# }}}

# integration in the frequency domain {{{
def fft_integration(signal, fs, fc=None, order=-1):
    """Intergration of a signal in the frequency domain using FFT.


    This function implements the integration in the time domain by means of the
    Fast Fourier Transform. It also performs a band pass filter, removing all
    the unwanted frequencies.

    Args:
        signal (array): Numpy 1d-array with the data.
        fs (float or int): Sampling frequency.
        fc (float or tuple): This is the cut-off frequency. If fc is floating or
            integer a lowpass filter is performed. If fc is a list a band
            pass-pass filter is made between the two given frequencies.
        order (integer): Indicates the order of the integration. Negative number
            indicates integration while positive indicates differentiontion (not
```

```
            implemented yet).

    Return (array): Signal integrated.
    """

    # check for nans if more than 10 percents
    nans = np.isnan(signal)
    if len(nans.nonzero()[0]) / len(signal) < 0.1:
        signal[nans] = np.nanmean(signal[~nans])
    else:
        return signal * np.nan


    # if order == 0 do nothing
    if order == 0:
        return signal

    # if order > 0 raise an error
    if order > 0:
        raise ValueError("Order must be a negative integer")

    # get frequency array
    N = len(signal)
    freqs = np.fft.fftfreq(N, 1/fs)


    # the first element of freqs array is zero, so we have
    # to discard it to avoid division by zero
    # the factor of integration is iw
    factor = 1j*2*np.pi*freqs[1:]


    # compute fft of the signal for the non-zero frequencies
    # and apply integration factor
    fft = np.zeros(len(freqs), 'complex')
    fft[1:] = np.fft.fft(signal)[1:] * factor**order


    # high pass filter
    if fc is None:
        return np.fft.ifft(fft).real
    elif isinstance(fc, float) or isinstance(fc, int):
        ix = abs(freqs) <= fc
        fft[ix] = 0.
    elif isinstance(fc, list) or isinstance(fc, tuple):
        flow, fhigh = fc
        ix = np.logical_and(abs(freqs) >= flow, abs(freqs) <= fhigh)
        fft[~ix] = 0.

    return np.fft.ifft(fft).real
# --- }}}

# rotation matrix in three dimensions {{{
def vector_rotation(U, E, units="rad"):
    """Apply a three dimensional rotation of the vector U given the angles T.


    Args:
        U (tuple): Components of the vector U = (u, v, w)
        E (tuple): Rotation angles corresponding to roll (x), pitch (y) and
            yaw (z), respectively. E = (phi, theta, psi)
        units (str): Flag to choose input angles between degrees or radians.

    Return (float): Components of the rotated vector.
    """

    # unpack tuples
    u, v, w = U
```

```
    phi, theta, psi = E

    # if input angles are in degrees, convert it to radians
    if units == "deg":
        phi, theta, psi = np.radians(phi), np.radians(theta), np.radians(psi)
    elif units == "rad":
        pass
    else:
        raise ValueError("units must be either def or rad")

    # check validity of angles
    # phi and psi must be between -pi and pi
    # theta musr be between -pi/2 and  pi/2

    # compute sins and cosines
    c_phi, c_theta, c_psi = np.cos(phi), np.cos(theta), np.cos(psi)
    s_phi, s_theta, s_psi = np.sin(phi), np.sin(theta), np.sin(psi)

    # apply the rotation of each components
    #
    # first component
    u_rot = (c_theta*c_psi)                       * u + \
            (s_phi*s_theta*c_psi - c_phi*s_psi) * v + \
            (c_phi*s_theta*c_psi + s_phi*s_psi) * w

    # second component
    v_rot = (c_theta*s_psi)                       * u + \
            (s_phi*s_theta*s_psi + c_phi*c_psi) * v + \
            (c_phi*s_theta*s_psi - s_phi*c_psi) * w

    # third component
    w_rot = (-s_theta)      * u + \
            (s_phi*c_theta) * v + \
            (c_phi*c_theta) * w

    return u_rot, v_rot, w_rot


# }}}

# angles_to_quaternions {{{
def angles_to_quaternions(T):
    """Returns the quaterions associatted with the angles T=(phi,theta,psi)."""

    # unpack tuples
    phi, theta, psi = T

    # compute sins and cosines
    c_phi, c_theta, c_psi = np.cos(phi), np.cos(theta), np.cos(psi)
    s_phi, s_theta, s_psi = np.sin(phi), np.sin(theta), np.sin(psi)

    # compute quaternions components
    q0 = 0.5*np.sqrt(1 + c_theta*s_psi + s_phi*s_theta*s_psi + \
            c_theta*c_psi + c_phi*c_theta)
    q1 = (s_phi*c_theta - c_phi*s_theta*s_psi + s_phi*c_psi) / (4*q0)
    q2 = (c_phi*s_theta*c_psi + s_phi*s_psi + s_theta) / (4*q0)
    q3 = (c_theta*s_psi - s_phi*s_theta*c_psi + c_phi*s_psi) / (4*q0)

    return (q1, q2, q3, q4)

# }}}

# split into flow, waves and turbulent part {{{
def split_into_three(data, fs=100, fc=(0.04,2)):
    """Split a signal into flow, waves and turbulent parts."""
```

```
    data_flow = bomm.butterworth_filter(data, fs, fc[0], kind="low")
    data_turb = bomm.butterworth_filter(data, fs, fc[1], kind="high")
    data_wave = bomm.butterworth_filter(data, fs, fc[0], kind="high") - data_turb

    return data_flow, data_wave, data_turb
# }}}

# get an array of a regular distribution of wavestaffs {{{
def wavestaff_coordinates(N, R=0.866, theta_0=-270):
    """This function returns position a regular array

    Function to get the coordinates (x, y) of and an array of wavestaffs formed by
    an regular N-vertices figured centered in (0,0) increasing counterclockwise
    Args:
        N (float): Number of vertices
        R (float): Separation
        theta_0 (float): Starting angle. 270 for the Ekinox reference frame.


    Returns:
        x (1d-array): x-coordinates of the array
        y (1d-array): y-coordinates of the array
    """

    theta = np.arange(0, 360, 360/N) - theta_0
    x, y  = [0], [0]
    x = np.append(x, R * np.cos(theta * np.pi / 180.))
    y = np.append(y, R * np.sin(theta * np.pi / 180.))

    return x, y
# }}}


# --- compute yaw, pitch and roll ---
# compute tilt from accelerations {{{
def tilt_from_accerometer(ax, ay, az):
    """Compute the inclination from the acceleromter signal as complex number."""

    # using the SBG rotation matrix and arctan
    phi = np.arctan(ay / az)
    theta = np.arctan(-ax / (ay*np.sin(phi) + az*np.cos(phi)))

    return phi, theta
# }}}

# compute pitch and roll using a complementary_filter {{{
def pitch_and_roll(ax, ay, az, wx, wy, wz, fs=100, fc=1/25, fm=1):
    """Euler angles using a complementary filter in frequency domain."""

    # compute pitch and roll from acclerometers
    phi_acc, theta_acc = tilt_from_accerometer(ax, ay, az)

    # compute pitch and roll from gyrospcope
    get_angle = lambda x: fft_integration(x, fs=fs, fc=fc, order=-1)
    phi_gyr, theta_gyr = get_angle(wx), get_angle(wy)

    # complementary filter
    phi = complementary_filter(phi_acc, phi_gyr, fs, fm=fm)
    theta = complementary_filter(theta_acc, theta_gyr, fs, fm=fm)

    # return data
    return phi, theta
# }}}

# orientation from magnetic north {{{
```

```
def yaw_from_magnetometer(wz, heading, fs=100, fc=1/25, fm=1):
    """Returns the yaw angle measured clockwise from north.

    This function computes the yaw angle which is equivalent to the buoy
    orientation. The function requires the data from the accelerometer and the
    magnetometer heading. Data are interpolated to the maximum sampling
    frquency. Uses a complementary digital filter to merge the high frequency
    gyroscope data with the low frequency magnetometer.

    Args:
        wz (float): Angular rate of change in rad/s.
        heading (float): Heading angle from magnetometer or signature in rad.
            Fortunately the heading usually follows the nautical convention,
            this means that the angle is measured clockwise which is consistent
            withe the Ekinox convention for the angles (Tait-Bryan or North,
            Easth, Down).

    Returns (float): Orientation respect to magnetic north.
    """

    # interpolate to the `wz` sampling frquency
    heading_fast = resample(wz, np.cos(heading)) + \
        1j* resample(wz, np.sin(heading))

    # compute psi angle from the magnetometre
    psi_mag = np.mod(np.ma.angle(heading_fast), 2*np.pi)
    mean_psi = np.nanmean(psi_mag)

    # compute pitch and roll from gyrospcope
    psi_gyr = fft_integration(wz, fs=fs, fc=fc, order=-1)

    # complementary filter
    psi = complementary_filter(psi_mag-mean_psi, psi_gyr, fs, fm)

    return np.mod(psi + mean_psi, 2*np.pi)
# }}}


# --- compute real position ---
# position correction {{{
def position_correction(X, A, E, fs=20, fc=1/25, q=5, full=False):
    """Correcion of the position and the surface elevation.

    This function applies the correction of the surface elevation measured by
    the wavestaffs due to the buoy inertial motion. The correction is perform
    not only to the surface elevation (z coordinate) but also for the position
    in the x-y-plane. So, the input/output is the uncorrected/corrected
    time-varying position vector of the water surface.

    The equation to perfom such correction is given by


                        //              /
      x_E =   R x_B    +    R || a_B dt   +    R | curl { Om_B, x_B } dt
                        //              /
            ---v---    -------v-------      ------------v--------------
             x_obs          x_acc                     x_rot

    where R is a rotation matrix.

          |  cosp*cosy  sinr*sinp*cosy-cosr*siny  cosr*sinp*cosy+sinr*siny |
      R = |  cosp*siny  sinr*sinp*siny+cosr*cosy  cosr*sinp*siny-sinr*cosy |
          |   -sinp           sinr*cosp               cosr*cosp           |

    in which, `p` is pitch (theta), `r` is roll (phi) and `y` is yaw (psi)
```

```
In the same way, the matrix of angular rate of changes is given by:

        | - dpdt siny + drdt cosp cosy |
 Omega = |   dpdt cosy + drdt cosp siny |
        |      dydt   - drdt sinp      |

Args:
    X (tuple): Contains the elements of X=(x,y,z). Each component is a time
        series given in a numpy 1d array.
    A (tuple): Contains the time series of the accelerations A=(ax, ay, az).
    E (dict): Contains the time series of the euler angles
        given by E=(roll, pitch, yaw).
    fs (float): Sampling frequency of the time series.
    fc (float): Cut-ff frequency for the integration. Could be a tuple.
    q (int): Factor to decimate the accelerometer data into wavestaff data.
        This value must be 5 since the ekinox frequency is 5 times the
        wavestaff frequency. When wavestaff measured at 10 Hz q must be 10.
    full (bool): Return full values or only the corrected ones. Default False.

Returns: Tuple with the X tuple corrected.

Note:
    Arrays must be clean before attempting to apply the correction.

References:
    * Anctil Donelan Drennan Graber 1994, JAOT 11, 1144-1150
    * Drennan Donelan Madsen Katsaros Terray Flagg 1994, JAOT 11, 1109-1116
"""

# substract gravity acceleration effect
G = vector_rotation((0,0,-9.8), E)

# apply double integration in the frequency domain
P = (fft_integration(A[0], fs*q, fc=fc*2.5, order=-2), # TODO: WARNING
     fft_integration(A[1], fs*q, fc=fc*2.5, order=-2), # TODO: WARNING
     fft_integration(A[2], fs*q, fc=fc, order=-2))


# compute the derivative of the euler angles
D = tuple(np.gradient(e, 1/(fs*q)) for e in E)

# decimate the high frequency signals to the given frecuency
# note than in this function i use the following equivalences
#   roll  --> r --> phi   --> E[0]
#   pitch --> p --> theta --> E[1]
#   yaw   --> y --> psi   --> E[2]
decimate = lambda x, q: x[::q]
E_down = tuple(decimate(e, q) for e in E) # <- Euler
P_down = tuple(decimate(p, q) for p in P) # <- Position
D_down = tuple(decimate(d, q) for d in D) # <- Derivative

# compute sines and cosines
roll, pitch, yaw = E_down
cp, sp = np.cos(pitch), np.sin(pitch)
cr, sr = np.cos(roll),  np.sin(roll)
cy, sy = np.cos(yaw), np.sin(yaw)

# convert observations into the inertial frame
x_obs, y_obs, z_obs = vector_rotation(X, (roll, pitch, yaw))


# correction due to translations
x_acc, y_acc, z_acc = vector_rotation(P_down, (roll, pitch, yaw))

# correction due to rotation
```

```
    xB, yB, zB = X
    droll, dpitch, dyaw = D_down
    curl = (fft_integration( dpitch*xB -   dyaw*yB, fs, fc, order=-1),
            fft_integration(-dpitch*zB +   dyaw*xB, fs, fc, order=-1),
            fft_integration(  droll*yB - dpitch*xB, fs, fc, order=-1))
    x_rot, y_rot, z_rot = vector_rotation(curl, (roll, pitch, yaw))

    # compute earth-based water position
    xE = x_obs + x_acc + x_rot
    yE = y_obs + y_acc + y_rot
    zE = z_obs + z_acc + z_rot

    # return data
    if full:
        return (x_obs,y_obs,z_obs), (x_acc,y_acc,z_acc), (x_rot,y_rot,z_rot)
    else:
        return xE, yE, zE

# --- }}}


# --- run as a script ---
if __name__ == "__main__":
    pass


# --- end of file ---
```

- - wdm.py - -

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
# vim:fenc=utf-8
#
# ================================================================
#  Copyright © 2017 Daniel Santiago <dpelaez@cicese.edu.mx>
#  Copyright © 2019 Juan Mtz-Osuna <jmartine@cicese.mx>
#  Distributed under terms of the GNU/GPL license.
# ================================================================

"""
File:     wdm.py
Created:  2017-03-19
Modified: 2019-04-08

Purpose:
    This module contain classes and functions to perform the Wavelet Directional
    Method to an array of WaveStaffs in order to get the wavenumber-direction
    wave spectrum. This method was proposed by Donelan et al. (1996)

    This is a special implemention planned to work on the on-board computer
    Nitrogen5x in the Oceanographic and Marine Meteorology Bouy (BOMM)

Author:
    Daniel S. P. Zapata
    dspelaez@gmail.com

Requirements:
    >> conda install -c alorenzo175 numpy scipy pandas

References:
    Donelan, M. A., Drennan, W. M., & Magnusson, A. K. (1996). Nonstationary
    analysis of the directional properties of propagating waves. Journal of
    Physical Oceanography, 26(9), 1901-1914.

    Donelan, M., Babanin, A., Sanina, E., & Chalikov, D. (2015). A comparison of
    methods for estimating directional spectra of surface waves. Journal of
    Geophysical Research: Oceans, 120(7), 5040-5053.

    Hauser, D., Kahma, K. K., Harald E. Krogstad, Susanne Lehner, Jaak Monbaliu
    and Lucy R. Wyatt (2003). Measuring and analysis the directional spectrum of
    ocean waves. URL:
    http://projects.knmi.nl/geoss/cost/Cost_Action_714_deel_1.pdf

    Hampson, R. W. (2008). Video-based nearshore depth inversion using WDM
    method. URL: http://www1.udel.edu/kirby/papers/hampson-kirby-cacr-08-02.pdf
    """

# --- import libs ---
import numpy as np
import pandas as pd
import scipy.signal as signal
import scipy.interpolate as interpolate
import os

import motion_correction as motcor


# --- wavelet transform ---

# compute wavelet frequencies {{{
```

```
def getfreqs(omin, omax, nvoice):
    """Returns the frequencies arrays for the wavelet spectrum.

    Args:
        omin (int): Minimun octave. It means fmin = 2^omin
        omax (int): Maximum octave. It means fmax = 2^omax
        nvoice (int): Number of voices. It means number of points between each
            order of magnitud. For example between 2^-4 and 2^-3 will be nvoice
            intermediate points.

    Returns: Array of frequencies logarithmic distributed.

    """
    return 2.**np.linspace(omin, omax, nvoice * abs(omin-omax)+1)
# }}}

# define mother wavelet {{{
def morlet(scale, omega, omega0=6.):
    """Returns the Morlet mother wavelet for a given scale array."""

    return (np.pi ** -.25) * np.exp(-0.5 * (scale * omega - omega0) ** 2.)
# }}}

#  continuous wavelet transform as torrence and compo {{{
def cwt(x, fs, freqs, mother=morlet):
    """
    This function compute the continuous wavelet transform


    Args:
        x      : time series
        fs     : sampling frquency [Hz]
        freqs  : array of frequencies
        mother : function to compute the cwt

    Returns:
        freqs, W:
    """

    # compute scales
    if mother == morlet:
        f0 = 6.
        flambda = (4 * np.pi) / (f0 + np.sqrt(2. + f0 ** 2.))
    else:
        raise NotImplementedError("Only Mortet was defined so far.")

    # scale
    scale = 1. / (freqs * flambda)

    # number of times and number of scales
    ntime  = len(x)
    nscale = len(scale)

    # fourier frequencies
    omega  = 2 * np.pi * np.fft.fftfreq(ntime, 1./fs)

    # loop for fill the window and scales of wavelet
    k = 0
    w = np.zeros((nscale, ntime))
    for k in range(nscale):
        w[k,:] = np.sqrt(scale[k] * omega[1] * ntime) * mother(scale[k], omega)

    # fourier transform of signal
    fft = np.fft.fft(x)

    # convolve window and transformed series
```

```
    fac = np.sqrt(2 / fs / flambda)
    return fac * np.fft.ifft(fft[None,:] * w, ntime)
# }}}


# --- geometry  ---

# function to get array of a regular distribution {{{
def reg_array(N, R, theta_0):
    """Thisn function returns position a regular array

    Function to get the coordinates (x, y) of and an array of wavestaffs formed by
    an regular N-vertices figured centered in (0,0) increasing counterclockwise

    Args:
        N (float): Number of vertices
        R (float): Separation
        theta_0 (float): Starting angle


    Returns:
        x (1d-array): x-coordinates of the array
        y (1d-array): y-coordinates of the array
    """

    theta = np.arange(1, 360, 360/N) - theta_0
    x, y  = [0], [0]
    x = np.append(x, R * np.cos(theta * np.pi / 180.))
    y = np.append(y, R * np.sin(theta * np.pi / 180.))

    return x, y
# }}}



# --- spectrum computation ---


# wavelet spectrograms of each wavestaff {{{
def wavelet_spectrogram(A, fs, omin=-6, omax=1, nvoice=16, mode='TC98'):
    """This function computes the wavelet spectrogram of an array of timeseries.

    Args:
        A (NxM Array): Surface elevation for each N wavestaff and M time.
        fs (float): sampling frequency.
        omin (int): Minimun octave. It means fmin = 2^omin
        omax (int): Maximum octave. It means fmax = 2^omax
        nvoice (int): Number of voices. It means number of points between each
            order of magnitud. For example between 2^-4 and 2^-3 will be nvoice
            intermediate points.
        mode (str): String to define if CWT is computing follong Torrence and
            Compo (1998) method (TC98) or Bertrand Chapron's (BC).

    """

    # define scales
    freqs = getfreqs(omin, omax, nvoice)

    # compute length of variables
    ntime, npoints = A.shape
    nfrqs = len(freqs)

    #  compute wavelet coefficients
    W = np.zeros((nfrqs, ntime, npoints), dtype='complex')
    for i in range(npoints):
```

```
        if mode == "TC98":
            W[:,:,i] = cwt(A[:,i], fs, freqs)
        elif mode == "BC":
            W[:,:,i] = cwt_bc(A[:,i], fs, freqs) * np.sqrt(1.03565 / nvoice)
        else:
            raise ValueError("Mode must be TC98 or BC")

    return freqs, W
# }}}

# wavenumber from spectrogram and array geometry {{{
def klcomponents(W, x, y, *args, **kwargs):
    """
    This function computes the directional wave spectrum using the
    Wavelete Directional Method proposed by Donelan et al. (1996)

    Args:
        W    : [2d-array] Wavelet coefficients
        x    : [2d-array] time varying x-coordinate of wavestaffs
        y    : [2d-array] time varying y-coordinate of wavestaffs

    Returns:
        k,l  : [2d-array] least square estimation of kk components

    """

    # --- compute phase and power for each frequency and time
    power = np.abs(W)**2
    power = power.mean(2)
    phase = np.arctan2(-W.imag, W.real) # -> cartesian towards

    # --- dimensions ---
    nfrqs, ntime, npoints = W.shape
    neqs    = int(npoints * (npoints-1) / 2)
    npairs  = int(neqs * (neqs-1) / 2)

    # check if position change in time
    if x.ndim == 2 and y.ndim == 2:
        pass
    else:
        x, y = np.tile(x,(ntime,1)), np.tile(y, (ntime,1))

    # --- loop for N unique pairs of equations ---
    X = np.zeros((ntime, neqs, 2))          # <--- matrix of distances
    Dphi = np.zeros((nfrqs, ntime, neqs))   # <--- vector of phase diffs
    #
    ij = 0
    for i in range(npoints-1):
        for j in range(i+1, npoints):
            #
            # distances between pairs for each time
            for k in range(ntime):
                dx = x[k,j] - x[k,i]
                dy = y[k,j] - y[k,i]
                X[k,ij,:] = [dx, dy]
            #
            # difference of phases
            Dphi[:,:,ij] = phase[:,:,j] - phase[:,:,i]
            #
            # acumulate counter
            ij += 1

    # --- contrains of phase differences ---
    # Dphi[Dphi >  np.pi] -= 2. * np.pi
    # Dphi[Dphi < -np.pi] += 2. * np.pi
```

```
    limit = kwargs.get('limit', np.pi)
    mask  = kwargs.get('mask',  np.nan)
    Dphi[np.abs(Dphi) > limit] = mask

    # --- least square estimation of vector kk=(k, l) ---
    #     the LSR of kk is given by:
    #       kk^LS = (X^T X)^-1 X^T phi
    #
    # function to compute wavenumber vector
    def wavenumber_solver(X, Dphi):
        return np.linalg.inv(X.T.dot(X)).dot(X.T).dot(Dphi)

    # TODO: define function to solve point to point
    #
    # evaluate function at each f-t point.
    kx = np.zeros((nfrqs, ntime))
    ky = np.zeros((nfrqs, ntime))
    for i in range(nfrqs):
        for j in range(ntime):
            kx[i,j], ky[i,j] = wavenumber_solver(X[j], Dphi[i,j,:])


    # return k and l
    return kx, ky
# }}}

# directional spreading function {{{
def directional_spreading(frqs, power, kx, ky):
    """
    Input:
        frqs        [1d-array] : Frequencies in Hz.
        power(f,t) [2d-array] : Power wavelet spectrum.
        k(f,t)      [2d-array] : X-component of wavenumber array.
        l(f,t)      [2d-array] : Y-component of wavenumber array.


    Output:
        D(f,dirs) [2d-array] : Directional spreading function in radians
    """

    # array of directions each degree
    dirs = np.arange(0, 360, 1)

    # compute magnitude and direction of wavenumber
    kappa = np.abs(kx + 1j*ky)
    theta = np.arctan2(ky, kx) # -> angle points the direction towards waves goes

    # round angles to a resolution of 1 degree and correct
    # angle to be measured counterclockwise from east
    theta_degrees = np.round(theta * 180./np.pi) % 360

    # length of arrays
    nfrqs, ntime = power.shape
    ndirs = len(dirs)

    # loop for each frequency
    D = np.zeros((ndirs,nfrqs), dtype='float')
    for j in range(nfrqs):


        # loop for each direction
        for i in range(360):
            ix = theta_degrees[j,:] == i
            dd = len(ix.nonzero()[0])
            weight = np.mean(power[j,ix]) if dd != 0 else 0.
            D[i,j] = dd * weight
```

```
    # normalize to satisfy int(D) = 1 for each direction
    m0 = np.trapz(D, x=dirs*np.pi/180, axis=0)
    D = D / m0[None,:]
    return D
# }}}

# smooth 2d arrays {{{
def smooth(F, ws=(5,1)):
    """
    This function takes as an argument the directional spectrum or
    another 2d function and apply a simple moving average with
    dimensions given by winsize.

    For example, if a we have a directional spectrum E(360,64) and
    ws=(5,2) the filter acts averging 10 directiona and 2 frequencies.

    Input:
        F  : Input function
        ws : Windows size

    Output:
        F_smoothed : Function smoothed

    """

    # define window
    nd, nf = ws
    frqwin = np.ones(nf)
    dirwin = signal.hamming(nd)
    window = frqwin[None,:] * dirwin[:,None]
    window = window / window.sum()


    # permorm convolution and return output
    return signal.convolve2d(F, window, mode='same', boundary='wrap')
# }}}

# interpolate spectrogram at specific frequencies {{{
def interpfrqs(S, frqs, new_frqs):
    """
    This function remap the log-spaced frequencies into linear frequencies


    Input:
        W       : Wavelet coefficiets. Dimensions W(nfrqs, ntimes, npoints)
        frqs     : log-spaced frequencies
        new_frqs : linear-spaced frequencies


    """
#    return interpolate.interp1d(frqs, S, fill_value='extrapolate')(new_frqs)
    return interpolate.interp1d(frqs, S)(new_frqs)
# }}}

# frequency - direction spectrum {{{
def fdir_spectrum(A, x, y, fs=10, omin=-6, omax=2, nvoice=16, ws=(30,1)):
    """Simple and ugly implementation of Wavelet Directional Method.

    Args:
        A (array): Surface elevation in the array.
        x (array): Time-varying x position of each probe.
        y (array): Time-varying y position of each probe.
        fs (float): Sampling frequency.
        omin (float): Min octave.
        omax (float): Max octave.
        nvoice (float): Number of voices.
        ws (tuple): Number of directions and frequencies to smooth.
```

```
    Returns:
        Frequceny-direction wave spectrum.

    """

    ntime, nprobes = A.shape
    nfft = int(2**np.floor(np.log2(ntime)))
    nperseg = int(nfft / 4)

    # obtain wavelet spectrogram for each gauge
    frqs, coefs = wavelet_spectrogram(A, fs, omin, omax, nvoice, mode='TC98')

    # compute components of wavenumber
    k, l = klcomponents(coefs, x, y, limit=np.pi)

    # compute power density from wavelets coefficients
    dirs = np.arange(0, 360, 1)
    power = np.mean(np.abs(coefs) ** 2, axis=2)

    # compute fourier spectrum and interpolate to wavelet frequencies
    Pxx = np.zeros((int(nperseg/2+1), nprobes))
    for j in range(nprobes):
        f, Pxx[:,j] = signal.welch(A[:,j], fs, "hann", nperseg)
    S = interpfrqs(Pxx.mean(1)[1:], f[1:], frqs)

    # compute directional spreading function and frequency direction spectrum
    D = directional_spreading(frqs, power, k, l)
    E = S[None,:] * D

    # smooth
    D_smoothed = smooth(D, ws)
    E_smoothed = smooth(E, ws)

    return frqs, dirs, E_smoothed, D_smoothed
# --- }}}



if __name__ == "__main__":
    pass


# --- end of file ----
```

```
------------------------------------------------------------------------------------------------
//    Program for tx/rx of data through TAOGLAS satellite  módem            //
//                          Ing. Juan Francisco Martínez Osuna                          //
//                          Last modification:        Sept / 2019                       //
------------------------------------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <termios.h>
#include <time.h>
#include <iostream>   // std::cout
#include <fstream>    // std::ifstream
#include <string>     // std::string, std::stoi
using namespace std;


#define MaxBuf 1000
#define MY_MASK 0777

struct termios oldtio, newtio;

time_t rawtime;
tm * ptm;

char Data[ 400], Mensaje[ 450];
char Respuesta[ MaxBuf], Nombre[ MaxBuf], Argumento[ 100];
char error[ 81], COM[ 15], path[ 100];
char car,caracter;

int   j, Baud_Rate, bandera=0;
int   yy, mm, dd, hh, nn, ss, anio, mes, dia, hora, minu, seg_ini, seg_fin, min_ini, min_fin;
int   puerto, inisec, finsec;
long  BAUD;

FILE *Archivo;


//---------------------Surbrutinas de configuración General---------------------//
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void ReadConfiguration()                    // Lee la configuración del hardware.
{
            printf("\n\nConfiguración:\n");
            printf("=================================================================\n");
            sprintf(Nombre,"Taoglas_RX.cfg");

            Archivo=fopen(Nombre,"r");
            if(Archivo == NULL)
            {
                        printf("%s\n", Nombre); perror(error); exit(-1);
            }

            fscanf(Archivo, "%s", COM);
            printf("Puerto:                                        %s\n", COM);
            fscanf(Archivo, "%d", &Baud_Rate);
            printf("Baudrate:                              %d\n", Baud_Rate);

            fscanf(Archivo, "%s", path);
            printf("Trayectoria:                           %s\n", path);

            fclose(Archivo);
```

```
}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void SelBaudRate()                                                    //Selecciona BaudRate de puertos.
{
 switch (Baud_Rate)
     {
       case 115200:
         BAUD = B115200;
         break;
       case 57600:
         BAUD = B57600;
         break;
       case 38400:
       default:
         BAUD = B38400;
         break;
       case 19200:
         BAUD = B19200;
         break;
       case 9600:
         BAUD = B9600;
         break;
       case 4800:
         BAUD = B4800;
         break;
       case 2400:
         BAUD = B2400;
         break;
       case 1200:
         BAUD = B1200;
         break;
       case 600:
         BAUD = B600;
         break;
       case 300:
         BAUD = B300;
         break;
     }  //fin del switch baud_rate
}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void OpenPort()                                                            // Inicializa puerto serial.
{
// r/w sin esperar a que llegue algo.
   //BAUD[portsel] = B9600;
   SelBaudRate();
   puerto  = open(COM, O_RDWR | O_NOCTTY);
   if (puerto < 0)
           { perror(COM); exit (-1); }
           else
           {
                   fcntl(puerto, F_SETFL, 0);
                   //printf("Puerto:                        %s abierto. \n", COM);
           }

// Guardar condiciones de puerto actuales.
   tcgetattr (puerto, &oldtio);
   bzero (&newtio, sizeof(newtio));

// Baudrate=Externo, CS8=8n1(8 bits, sin paridad y 1 stop bit)
// CLOCAL=conexionn local, sin control de modem
// CREAD=habilitar caract. recepcion.
   newtio.c_cflag = BAUD | CS8 | CLOCAL | CREAD;

// IGNPAR= Ignorar bytes con error de paridad.
   newtio.c_iflag = IGNPAR;
```

```
// Salida sin formato.
   newtio.c_oflag = 0;
   newtio.c_lflag = 0;
   newtio.c_cc[VTIME] = 0x00;
   newtio.c_cc[VMIN] = 0x00;
   tcflush   (puerto, TCIFLUSH);
   tcsetattr (puerto, TCSANOW, &newtio);
}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void GetDate()                                                                                          // Obtiene Fecha.
{
time(&rawtime);
ptm = localtime(&rawtime);

            yy=ptm->tm_year+1900;
            mm=ptm->tm_mon+1;
            dd=ptm->tm_mday;
            hh=ptm->tm_hour;
            nn=ptm->tm_min;
            ss=ptm->tm_sec;

            anio=yy; mes=mm; dia=dd;
}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void WaitMinute()                                                    // Espera nuevo segundo
{
int min_ini, min_fin;

   time (&rawtime);
   ptm = localtime(&rawtime);
   min_ini = ptm->tm_min;

   do
   {
            time (&rawtime);
            ptm = localtime(&rawtime);
            min_fin = ptm->tm_min;
   } while (min_fin == min_ini);
}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
int ValidChars(char h)                                         // Verifica caractéres validos.
{
            if(h=='0'||h=='1'||h=='2'||h=='3'||h=='4'||h=='5'||h=='6'||h=='7'||h=='8'||
                        h=='9'||h=='.'||h==','||h=='-'||h==' '||h=='\n'||h=='\r'||h==13||h=='\0')
                        return (1);
            else
                        return (0);
}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void ReadPort(int crlf)                                                                   // Lee puerto serial.
{
int rflag,resp,flag, min_fin;
strncpy(Respuesta,"",MaxBuf);
strcpy(Respuesta,"");
tcflush   (puerto, TCIOFLUSH);

GetDate();
//printf("Minuto inicial = %d \n", nn);

            j=0; rflag=0, flag=0;

do
{
```

```
		do
		{
			rflag = read(puerto, &car, 1);
			resp  = ValidChars(car);
			if (rflag > 0 && resp>0)
			{
				Respuesta[j] = car; j++;
			}
			time(&rawtime);
			ptm = localtime(&rawtime);
			min_fin=ptm->tm_min;

		}
		while ((car != '\n') && (nn == min_fin));
		car = (char)NULL;
		flag++;
		//printf("Minuto final = %d \n", min_fin);
		printf("Cadena en el puerto %s",Respuesta);

} while (flag <= crlf);

}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void WaitSecond()							// Espera nuevo segundo.
{
   time (&rawtime);
   ptm = localtime(&rawtime);
   inisec = ptm->tm_sec;

   do
   {
		time (&rawtime);
		ptm = localtime(&rawtime);
		finsec = ptm->tm_sec;
   } while (finsec == inisec);
}


//----------------------Surbrutinas de recepción----------------------//
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void VerifyMTA()				// Verifica función "Ring Alert".
{
do
		{
		char IDdisp[13]= {"AT+SBDMTA?\r"};
		write(puerto,&IDdisp, 11);
		sleep(.5);
		ReadPort(0);
		caracter=Respuesta[0];

					if(caracter != '1')
					{
						printf("Tratando de activar Ring Alert\n");
						char IDdisp[14]= {"AT+SBDMTA=1\r"};
						write(puerto,&IDdisp, 12);
						sleep(2);
						char IDdispA[13]= {"AT+SBDMTA?\r"};
						write(puerto,&IDdispA, 11);
						sleep(.5);
						ReadPort(0);
						caracter=Respuesta[8];
					}

					if(caracter == '1')
```

```
					{
						printf("Ring Alert Activado: %s",Respuesta);
					}



		} while(caracter != '1');
}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void VerifyREG()				// Registra el módem en la RED.
{
do
		{
		char IDdisp[13]= {"AT+SBDREG?\r"};
		write(puerto,&IDdisp, 11);
		sleep(.5);
		ReadPort(0);
		caracter=Respuesta[0];


					if (caracter != '2')
					{
						printf("Tratando de registrar el modem\n");
						char IDdisp[14]= {"AT+SBDREG\r"};
						write(puerto,&IDdisp, 12);
						sleep(2);
						char IDdispA[13]= {"AT+SBDREG?\r"};
						write(puerto,&IDdispA, 11);
						sleep(.5);
						ReadPort(0);
						caracter=Respuesta[0];
					}

					if(caracter == '2')
					{
						printf("Modem correctamente registrado en la RED: %s\n",Respuesta);
					}

		} while(caracter != '2');

}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void ReadModemStatus()		// Verifica nuevo archivo en el satélite.
{
char IDdisp[11]= {"AT+SBDSX\r"};
printf("Escrbire codigo %s para leer estatus del modem\n",IDdisp);
write(puerto,&IDdisp, 9);
sleep(.5);
ReadPort(0);
caracter=Respuesta[15];
//caracter=Respuesta[18];
}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void DownloadFile()			// Descarga nuevo archivo del satélite.
{
char IDdisp[10]= {"AT+SBDI\r"};
write(puerto,&IDdisp, 8);
sleep(1);
ReadPort(0);
caracter=Respuesta[10];
}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void ReadFile()			// Descarga la información del archivo.
{
```

```
char IDdisp[11]= {"AT+SBDRT\r"};
write(puerto,&IDdisp, 9);
sleep(.5);
ReadPort(1);
}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void Decode()			// Decodifica la información del archivo.
{
//		int dia, mes, anio, hora, min;
		char c[ 10], d[ 10];

		sscanf(Respuesta,"%d,%d,%d,%d,%d",&dia,&mes,&anio,&hora,&minu);

		/*printf("String1 = %s\n",c);
		printf("día = %02d\n",dia);
		printf("mes = %02d\n",mes);
		printf("anio = 20%02d\n",anio);
		printf("hora = %02d\n",hora);
		printf("minu = %02d\n",minu);*/

		sprintf(Argumento,"20%02d %02d %02d %02d %02d",dia, mes, anio, hora, minu);
		printf("Argumento para programa WDM: %s\n",Argumento);

}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void ClearBuf()				// Limpia el Buffer del módem.
{
char IDdisp[11]= {"AT+SBDD1\r"};
write(puerto,&IDdisp, 9);
sleep(.5);
}

//---------------------Surbrutinas de transmisión---------------------//

//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
bool FileExists(const std::string& name) {
  struct stat buffer;
  return (stat (name.c_str(), &buffer) == 0);
}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
int VerifyFile()
{
   std::string filename = "/home/tito-dof/Documents/2019/Maestria/wdm/wdm-wdm_nitrogen5x/wdm/test.spectrum.npy";
   //std::cout << "Does file_name.txt Exists ? " << FileExists(filename) << std::endl;
   printf("Valor=%d\n",FileExists(filename));
   return FileExists(filename);
}
//---------------------------------------------------------------------
int printSize(const std::string& address) {
  std::fstream motd(address.c_str(), std::ios::binary|std::ios::in|std::ios::ate);
  if(motd) {
    std::fstream::pos_type size = motd.tellg();
    std::cout << address << " " << size << "\n";
    return size;
  } else {
    perror(address.c_str());
  }

}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
int ReadCheckSum(char Info[ 400])
{
unsigned int checksum = 0; /*Unsigned 16 bit integer*/
int i;
```

```
unsigned char e;
//char* dta = "hello";
int length; /* Number of characters in data */

        length = strlen(Info);
        printf("largo de la variable length es: %d\n",length);

        for (i=0;i<length;i++)
        {
                e = Info[i];
                //putchar(c);
                checksum += e;
        }

//      putch(checksum%256);

        //printf("\n");
        printf("\nEl valor del checksum es: %02x %02x\n", checksum/256,checksum%256);

        return checksum;
}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void ReadData(int z)                                    // Lee información del archivo.
{
int sz;

        sprintf(Nombre,"%s/wdm/wdm-wdm_nitrogen5x/wdm/EDO_%02d.npy",path,z);
        //sprintf(Nombre,"/home/tito/Dropbox/2018/Maestria/2018-2/Iridium.txt");
        //sprintf(Nombre,"/home/root/Iridium.txt");

        Archivo=fopen(Nombre,"r");

        if(Archivo == NULL)                             // Si no existe el archivo lo crea.
        {
                // Genera un archivo con tres caractéres (NFI - NoFileIridium)
                printf("%s\n", Nombre); perror(error);

                printf("Creando archivo log \n");
                system("echo NFI  > /home/tito-dof/Documents/2019/Maestria/wdm/wdm-wdm_nitrogen5x/wdm/Iridium.txt");

                system("chmod 777 /home/tito-dof/Documents/2019/Maestria/wdm/wdm-wdm_nitrogen5x/wdm/Iridium.txt");

                Archivo=fopen(Nombre,"r");
        }

        ifstream myFile (Nombre, ios::in | ios::binary);
        myFile.read (Data, sz);
        //fscanf(Archivo, "%s\n", Data);
        //printf("Datos               %s\n", Data);

        fclose(Archivo);
}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void ReadStatus()                                               // Lee estatus del módem.
{
        char IDdisp[4]= {"AT\r"};
        write(puerto,&IDdisp, 3);

        //printf("\n Identificacion del dispositivo: %s",IDdisp);
        ReadPort(0);
}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void WriteMessageText()                                 // Escribe mensaje en modo texto.
```

```
{
	strcpy(Respuesta,"");
	char IDdisp[10]= {"AT+SBDWT="};
	//sprintf(datos,"f10v1.1");
	sprintf(Mensaje,"%s%s\n\r",IDdisp,Data);
	write(puerto,&Mensaje,65);

	//printf("largo de mensaje %u \n",strlen(Mensaje));
	//printf("Mensaje escrito: %s\n",Mensaje);
}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void WriteMessageBinary()				// Escribe mensaje en modo binario.
{
	strcpy(Respuesta,"");
	char IDdisp[10]= {"AT+SBDWB="};


	sprintf(Mensaje,"%s330\n\r",IDdisp);
	write(puerto,&Mensaje,12);
	sprintf(Mensaje,"%s",Data);
	write(puerto,&Mensaje,330);

	//printf("largo de mensaje %u \n",strlen(Mensaje));
	//printf("Mensaje escrito: %s\n",Mensaje);
}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void ReadNetwork()		// Lee el estado actual de la red.
{
	char IDdisp[9]= {"AT+CSQ\r"};

	write(puerto,&IDdisp, 7);
	ReadPort(0);
	//printf("\n Estado de la red: %s",IDdisp);
}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void SendMessage()			// Envía mensaje al satélite.
{
	strcpy(Respuesta,"");
	char IDdisp[10]= {"AT+SBDI\r"};

	write(puerto,&IDdisp, 8);
	ReadPort(0);
}
//*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
void FlushModem()			// Limpia la memoria del módem.
{
	char IDdisp[7]= {"AT*F\r"};
	write(puerto,&IDdisp, 5);
}

//-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*- PRINCIPAL -*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
int main (void)								// Función principal.
{
int intervalo_seg = 10, intervalo_min = 5, estatus = 0, arch, k, m, borrar;
unsigned int chksm = 0; /*Unsigned 16 bit integer*/
char Nombre1[ 50], Nombre2[ 50], Almacenar[ 100], cmd[1000];
//---------------------------------------------------------------------------------------------------------


					// Configuración general del sistema.
	ReadConfiguration();
	OpenPort();
	printf("===============================================================\n");
```

```
//------------------------------------------------------------------------------------------------------------
            // Inicia subrutina principal
do
{
            GetDate();
            sleep(1);
            //printf("modulus:%d, hh: %d, nn: %d, ss:%d \n",hh % 2,hh,nn,ss);
            //if (hh % 2 == 0 && nn % 2 == 0 && ss == 00)
            //if (hh % 2 == 0 && nn == 30 && ss == 00)
            //sleep(54); // Revisa cada 55 segundos.
            VerifyMTA();
            sleep(1);
            VerifyREG();
            sleep(1);

            ReadModemStatus();
            printf("Caracter es: %c\n",caracter);
            if(caracter == '1')
            {
                        printf("Nuevo archivo en el satélite: %s\n",Respuesta);
                        DownloadFile();
                        sleep(1);
                        ReadFile();
                        sleep(1);
                        Decode();

                        //-------------- Ejecuta software para WDM --------------//
                        //printf("Argumento es: %s",Argumento);
                        //sprintf(cmd, "/home/tito-dof/miniconda3/bin/python %s/WDM/wdm-wdm_nitrogen5x/wdm/run.py 2017 11 17 00
00\n",path);
                        //printf("comando es: %s",cmd);
                        sprintf(cmd, "/home/tito-dof/miniconda3/bin/python %s/wdm/wdm-wdm_nitrogen5x/wdm/run.py
%s",path,Argumento);
                        system (cmd);
                        printf("ejecute instrucción\n");
                        estatus=1;
                                    //-------------------------------------------------------//

//                                  ClearBuf();
            }
            else
            {
                        printf("Satelite sin nueva información \n");
            }

            if(estatus == 1)
            {
                        arch=VerifyFile();

                        if (arch == 1)
                        {
                                    printf("Archivo existente\n");
                                    for (k=0; k<22; k++)
                                    {

                                                ReadData(k);
                                                WriteMessageBinary();
                                                ReadNetwork();
                                                printf("numero de caracteres: %u \n",strlen(Respuesta));

                                                caracter=Respuesta[5];
                                                printf("Estado de la red: %s",Respuesta);
                                                printf("\n caracter de mensaje es: %c \n",caracter);
//------------------------------------------------------------------------------------------------------------
```

```
                                                    // Envia mensaje.
                    if(caracter >= '3')
                    {

                        printf("entre a subrutina para envío del mensaje \n");
                        SendMessage();
                        caracter=Respuesta[8];

                        printf("Respuesta del envio %s\n",Respuesta);
                        //printf("largo de cadena de respuesta de envio: %u \n",strlen(Respuesta));
                        //printf("\n caracter de envio es: %c \n",caracter);

                            if(caracter == '1')
                            {
                                printf("Mensaje enviado exitosamente: %s\n",Respuesta);
                                printf("----------------------------------------------------\n\n");

                            }
                    }
                    sleep(120);
                    GetDate();
                }



                    // Genera archivos a partir del espectro.



            }
                sprintf(Almacenar,"mv test.spectrum.npy ../wdm/data/spectrum_%02d%02d_%02d
%02d.npy",mes,dia,hora,minu);
                system(Almacenar);
                estatus=0;
        }
            else
            {
                printf("No existe el archivo\n");
            }

} while (1);

        WaitMinute();
        WaitMinute();
        WaitMinute();

}
```

```
% -----------------------------------------------------------
% Programa para despliegue del espectro direccional del oleaje.
% Ing. Juan Francisco Martínez Osuna.
% Octubre2019
% -----------------------------------------------------------

clear all; close all;
s1 = load('test.spectrum_1710.txt' );
%s2 = load('test.spectrum_1000.txt' );
s2 = load('test.spectrum_1810.txt' );

% Define resolución del EDO.
omin = -4;
omax = -1;
nvoice = 16;
frec = 2.^linspace(omin, omax, nvoice * abs(omin-omax)+1);
frecc = linspace(1/16,.4,4)
% dir = 0:5:360;

dir = linspace(0,360,72);

frecn=frec./max(frecc);
freccn=frecc./max(frecc);

% Transporta de coordenadas polares a cartesianas.

for j=1:length(dir)
for i=1:length(frecn)
        x(j,i)= sin(dir(j)*(pi/180))*frecn(i);
        y(j,i)= cos(dir(j)*(pi/180))*frecn(i);
end
end

xc=freccn(end)*sin([-pi:pi/500:pi]);
yc=freccn(end)*cos([-pi:pi/500:pi]);

ind=find(inpolygon(x,y,xc(end,:),yc(end,:)) == 0);
for j=1:3
  evalc(['s' num2str(j) '(ind)=NaN']);
end
s(ind)=NaN;



% plotdspec(s,frec,dir,parula(64),linspace(1/16,.4,4),.1)
% for j=0:4;contour(x,y,s,[eval(['[1e' num2str(j) ' 1e' num2str(j) ']'])],'color',[1 1 1]*.4); hold on; end
% cb = colorbar;
% caxis([1e0 1e4])
% ylabel(cb, 'Energía (m^{2}Hz^{-1}rad^{-1})')
% xlabel('Espectro Direccional del Oleaje')

gap=.05;
h=.05;
w=.05;

% Grafica Espectro direccional del oleaje.

for j=1:2


  subtightplot(1,2,j,gap,h,w);
  plotdspec(eval(['s' num2str(j) ]),frec,dir,parula(64),linspace(1/16,.4,4),.2); hold on
  caxis([1e0 1e4])
  for k=0:4;contour(x,y,eval(['s' num2str(j) ]),[eval(['[1e' num2str(k) ' 1e' num2str(k) ']'])],'color',[1 1 1]*.4); hold on; end
```

```
end

cb = colorbar;
set(cb,'location','southoutside')
set(cb,'position',[0.05 0.12 0.9 0.04])
caxis([1e0 1e4])
xlabel(cb, 'Energía (m^{2}Hz^{-1}rad^{-1})','fontsize',10)

axes('position',[0 0 1 1]);
text(.5,.9,'Espectro Direccional del Oleaje','horizontalalignment','center');
set(gca,'visible','off');

text(.17,.85,'2017-11-17 10:00','horizontalalignment','left');
text(.86,.85,'2017-11-18 10:00','horizontalalignment','right');
%title('Espectro Direccional del Oleaje')

print -depsc2 -r400 -painters ejemplo.eps
return

% subtightplot(2,2,2,gap,h,w);
% plotdspec(s,frec,dir,parula(64),linspace(1/16,.4,4),.2); hold on
% for j=0:4;contour(x,y,s,[eval(['[1e' num2str(j) ' 1e' num2str(j) ']'])],'color',[1 1 1]*.4); hold on; end
% print -depsc2 -r400 -painters ejemplo.eps
```