Python de Extremo a Extremo

Automatización, Análisis Geoespacial y Visión Artificial

Tomás Valderrama

Índice

1 Python de Extremo a Extremo

Automatización, Análisis Geoespacial y Visión Artificial

Manual práctico con ejemplos reales de procesamiento de datos científicos: desde la lectura de archivos hasta la generación automática de informes Word, el análisis espectral de series temporales, la digitalización de cartografía con OCR y el manejo de datos geoespaciales.

1.1 Contenidos

- **Parte I** — Entorno Spyder, sintaxis y fundamentos del lenguaje
- **Parte II** — Manejo de datos con NumPy, Pandas y lectura de archivos
- **Parte III** — Visualización con Matplotlib, rosas de viento y figuras para informes
- **Parte IV** — Análisis de señales: estadísticas circulares, FFT y filtros
- **Parte V** — Automatización de informes Word con python-docx
- **Parte VI** — Datos geoespaciales con Rasterio y GeoPandas
- **Parte VII** — Visión artificial: OCR de cartas batimétricas con OpenCV y Tesseract
- **Parte VIII** — Modelos numéricos oceánicos: outputs NetCDF de CROCO con xarray
- **Parte IX** — Datos satelitales: Sentinel-2 vía API STAC de Copernicus y análisis de tiles MGRS

1.2 Requisitos

```
pip install numpy pandas matplotlib scipy openpyxl python-docx windrose rasterio
↪  geopandas pyproj contextily opencv-python pytesseract pillow xarray netcdf4
↪  cartopy pystac-client shapely
```

1.3 Cómo usar este manual

Cada capítulo incluye explicaciones y fragmentos de código aplicados a problemas reales de procesamiento de datos.

2 Entorno de desarrollo

Python se puede escribir y ejecutar desde varios entornos. Cada uno tiene un perfil distinto — la elección depende del tipo de tarea y del momento del proyecto.

2.1 Comparativa de entornos

Entorno	Mejor para	Instalación
Spyder	Análisis interactivo, explorar datos, perfil científico	Anaconda
VSCode	Proyectos multi-archivo, Git, scripts de producción	Descarga directa
JupyterLab	Reportes con código + texto, compartir resultados	`pip install jupyterlab`
Google Colab	Sin instalación, acceso a GPU, colaboración	Solo navegador

Este capítulo se enfoca en **Spyder**, que es el más adecuado para empezar: tiene Variable Explorer integrado, permite ejecutar código por secciones y está diseñado específicamente para análisis de datos. Las secciones al final del capítulo describen cuándo conviene cambiar a VSCode o Jupyter.

2.2 Spyder

Spyder es el entorno de desarrollo recomendado para análisis científico en Python. A diferencia de otros editores, está diseñado específicamente para trabajar con datos: tiene un explorador de variables integrado, una consola interactiva y permite ejecutar código por secciones.

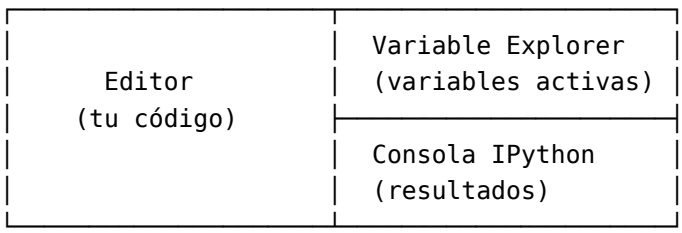
2.3 Instalación

La forma más sencilla es instalar **Anaconda**, que incluye Python, Spyder y las librerías científicas principales:

https://www.anaconda.com/download

2.4 La interfaz de Spyder

Spyder tiene tres paneles principales:



- **Editor**: donde escribes y guardas tu script `.py`
- **Consola IPython**: donde se ejecuta el código y se muestran resultados
- **Variable Explorer**: muestra todas las variables activas, sus tipos y valores — muy útil para inspeccionar DataFrames y arrays

2.5 Ejecutar código

Acción	Atajo
Ejecutar el script completo	`F5`
Ejecutar la línea actual	`F9`
Ejecutar una selección	Seleccionar + `F9`
Ejecutar una celda	`Ctrl + Enter`
Ejecutar celda y avanzar	`Shift + Enter`

2.6 Atajos de edición esenciales

Acción	Atajo
Comentar / descomentar selección	`Ctrl + 1`
Insertar separador de celda `# %%`	`Ctrl + 2`
Duplicar línea	`Ctrl + D`
Mover línea arriba / abajo	`Alt + ↑ / ↓`
Buscar en el archivo	`Ctrl + F`
Ir a definición de función	`Ctrl + G`

`Ctrl + 1` es especialmente útil para desactivar temporalmente un bloque de código sin borrarlo. `Ctrl + 2` inserta un `# %%` en la posición del cursor, creando una nueva celda ejecutable en ese punto.

2.7 Celdas de código

Las celdas permiten dividir el script en bloques ejecutables de forma independiente, similar a un notebook pero dentro de un archivo `.py` normal.

Se definen con `# %%`:

```
# %% Cargar datos
import pandas as pd
df = pd.read_csv('corrientes.csv')

# %% Graficar
import matplotlib.pyplot as plt
plt.plot(df['velocidad'])
plt.show()
```

Cada bloque `# %%` se puede ejecutar por separado con `Ctrl + Enter`. Esto es muy útil para procesar datos paso a paso sin reejecutar todo el script.

2.8 Variable Explorer

El explorador de variables muestra en tiempo real:

- Nombre y tipo de cada variable
- Dimensiones de arrays y DataFrames
- Vista previa de los valores

Se puede hacer doble clic en un DataFrame para abrirlo en una tabla interactiva, o en un array para ver su contenido completo.

2.9 Consola IPython

La consola acepta comandos directos sin necesidad de estar en el editor:

```
# Ver las primeras filas de un DataFrame
df.head()

# Ver el tipo de una variable
type(df)

# Obtener ayuda de una función
help(pd.read_csv)
# o más rápido:
pd.read_csv?
```

2.10 Configuración recomendada

Algunas opciones útiles en **Tools → Preferences**:

- **Editor → mostrar números de línea**: facilita depuración
- **IPython console → Graphics → Backend: Inline**: los gráficos aparecen en la consola (recomendado para análisis)
- **IPython console → Graphics → Backend: Tkinter**: cada gráfico abre en ventana separada, liviana e interactiva (zoom, pan, guardar)
- **IPython console → Graphics → Backend: Qt5**: similar a Tkinter pero más pesado; en algunos equipos es lento o inestable

También se puede cambiar el backend por sesión desde la consola sin tocar las preferencias:

```
%matplotlib inline   # figuras en consola
%matplotlib tk       # ventana Tkinter interactiva
%matplotlib qt5      # ventana Qt5
```

2.10.1 Cuándo usar cada backend

Situación	Backend
Explorar datos, iterar rápido	Inline
Revisar figura con zoom o hacer clics sobre ella	Tkinter
Script automático que solo guarda PNG	`matplotlib.use('Agg')` antes de importar pyplot

2.10.2 Tkinter vs Qt5 — por qué Tkinter es más estable en Spyder

Spyder está construido sobre Qt5 (PyQt5). Cuando se usa el backend Qt5Agg, matplotlib intenta crear ventanas Qt5 dentro del mismo event loop que ya está usando Spyder — pueden producirse conflictos, ventanas lentas o cierres inesperados.

Tkinter tiene su propio event loop completamente independiente de Qt, por eso no interfiere con Spyder.

En **VSCode** este problema no existe: VSCode está hecho en Electron (Chromium), no en Qt. El backend Qt5Agg abre ventanas Qt5 de forma independiente y funciona sin conflictos.

Otra causa de ventanas que ”se congelan”: si el script hace procesamiento pesado (imágenes, OCR, cálculos largos) en el mismo hilo que la ventana gráfica, el event loop no puede responder a clics mientras Python está ocupado. Esto ocurre con cualquier backend, pero Qt5 lo manifiesta más visiblemente.

	Tkinter	Qt5
Integración con Spyder	Sin conflictos	Puede interferir con el event loop
Integración con VSCode	Bien	Bien
Peso	Liviano, incluido en Python	Pesado, requiere PyQt5/PySide2
Widgets interactivos	Básicos (zoom, pan, guardar)	Avanzados (sliders, botones personalizados)
Uso recomendado	Backend interactivo general	Herramientas con interfaz propia

[>] Recomendación — Usar Inline como predeterminado en Spyder. Cambiar a Tkinter cuando se necesita interactividad. Qt5 solo tiene ventaja real si se construye una interfaz con botones o controles propios, y en ese caso es mejor trabajar en VSCode o en un script independiente fuera de Spyder.

2.11 VSCode

Visual Studio Code es un editor de propósito general con soporte de primera clase para Python. No está diseñado específicamente para análisis de datos, pero tiene ventajas claras cuando el proyecto crece.

Instalar: code.visualstudio.com. Luego instalar la extensión **Python** (Microsoft) y opcionalmente **Pylance** para autocompletado avanzado.

2.11.1 Ventajas sobre Spyder

- **Editor más potente**: autocompletado inteligente entre archivos, ir a definición entre módulos, refactoring, buscar y reemplazar en todo el proyecto
- **Git integrado**: diff visual, historial de commits, blame por línea, sin salir del editor
- **Terminal real**: bash/WSL integrado directamente, útil para correr scripts, git, pandoc
- **Multi-archivo**: navegación entre archivos de un paquete Python es mucho más fluida
- **Sin conflicto Qt5**: VSCode usa Electron (no Qt), por lo que el backend Qt5Agg de matplotlib funciona sin interferencias
- **Extensiones**: SSH remoto (trabajar en servidores Linux), Jupyter, Docker, soporte para otros lenguajes en el mismo proyecto

2.11.2 Cuándo conviene migrar a VSCode

- El proyecto tiene varios archivos `.py` que se importan entre sí
- Se trabaja con Git activamente
- Se desarrolla una herramienta con interfaz interactiva (matplotlib + clicks, sliders)
- Se necesita conectarse a un servidor remoto (cluster para CROCO, por ejemplo)

2.11.3 Variable Explorer en VSCode

VSCode no tiene un Variable Explorer tan completo como Spyder. La alternativa es usar el modo **Jupyter** (celdas `# %%` con el kernel interactivo), que muestra variables y permite inspeccionarlas. Para DataFrames, `df.head()` o `df.describe()` en la terminal interactiva cumplen el mismo rol.

2.12 JupyterLab y Jupyter Notebook

Jupyter permite mezclar código Python, texto explicativo (Markdown) y resultados (gráficos, tablas) en un mismo documento interactivo llamado **notebook** (`.ipynb`).

```
pip install jupyterlab
jupyter lab   # abre en el navegador
```

2.12.1 Cuándo usar Jupyter

- **Exploración inicial de datos nuevos**: ver los datos, graficar, entender la estructura
- **Reportes reproducibles**: compartir análisis donde el lector puede ver el código y los resultados juntos
- **Tutoriales y documentación**: la mayoría de tutoriales de xarray, pandas, geopandas están en notebooks
- **Colaboración**: Google Colab permite compartir y ejecutar notebooks sin instalar nada

2.12.2 Limitaciones de Jupyter para producción

Los notebooks tienen desventajas cuando el código madura:

- El orden de ejecución de celdas puede ser arbitrario — un notebook con celdas ejecutadas fuera de orden produce resultados incorrectos sin avisar
- Difíciles de versionar con Git (el formato `.ipynb` incluye outputs, lo que genera diffs enormes)
- No se pueden importar como módulos desde otros scripts
- La refactorización y navegación entre archivos es más limitada

Regla práctica: usar Jupyter para explorar y comunicar resultados; convertir el código útil a scripts `.py` para producción.

2.13 PyCharm

PyCharm (JetBrains) es un IDE profesional orientado al desarrollo de software en Python. Tiene herramientas avanzadas de refactoring, depuración y gestión de proyectos que lo hacen muy popular en desarrollo de aplicaciones y backend.

Para análisis de datos y scripting científico no es la primera opción: no tiene Variable Explorer, su curva de aprendizaje es mayor, y la versión completa es de pago (existe una Community Edition gratuita). Se menciona aquí porque es frecuente encontrarlo en tutoriales y comparativas, pero no se cubre en este manual — para el tipo de trabajo que se describe, Spyder y VSCode cubren mejor el espectro.

2.14 Google Colab

Jupyter en la nube de Google. No requiere instalar nada — solo un navegador y cuenta de Google.

- URL: colab.research.google.com
- Acceso gratuito a GPU (útil para modelos de deep learning o procesamiento de imágenes pesado)
- Se integra con Google Drive para leer y guardar archivos

- Ideal para compartir análisis con alguien que no tiene Python instalado

La limitación principal: la sesión se desconecta después de un tiempo de inactividad y los datos temporales se pierden. Para trabajo continuo es mejor un entorno local.

3 Spyder vs VSCode

Spyder es el mejor punto de partida para análisis de datos: es simple, tiene Variable Explorer y está pensado para trabajar con datos desde el primer día. VSCode es más potente como editor y se adapta mejor cuando el código crece en proyectos multi-archivo.

Este capítulo compara ambos entornos y explica cómo hacer la transición cuando conviene.

3.1 Por qué este manual usa Spyder y no Jupyter

Jupyter es muy popular en ciencia de datos, y es probable que lo encuentres en tutoriales y papers. Sin embargo, para el tipo de trabajo que cubre este manual — pipelines de procesamiento, automatización de informes, análisis reproducibles — Spyder con scripts `.py` tiene ventajas concretas sobre notebooks `.ipynb`.

El problema de estado de los notebooks

En Jupyter, las celdas se pueden ejecutar en cualquier orden. Esto crea "estado oculto": variables que existen en memoria de corridas anteriores, resultados que dependen del orden en que ejecutaste las celdas, comportamiento que no se puede reproducir corriendo el notebook de arriba a abajo. En un análisis corto de exploración no importa; en un pipeline de producción es una fuente de errores difíciles de detectar.

```
# Celda 1
df = pd.read_csv('datos.csv')

# Celda 3 (ejecutada antes que la 2)
df = df[df['velocidad'] > 0]   # modifica df

# Celda 2 (ejecutada después)
print(len(df))   # resultado depende del orden de ejecución
```

En un script `.py` de Spyder el código siempre corre de arriba a abajo. El estado es predecible.

Los scripts `.py` son importables, los notebooks no

Cuando el código madura, se organiza en funciones que se reutilizan entre proyectos. Un script `.py` se puede importar directamente:

```
from utils import calcular_media_vectorial
```

Un notebook `.ipynb` no se puede importar sin conversión previa. Esto hace que el código de un notebook quede "atrapado" en él.

Git funciona bien con `.py`, mal con `.ipynb`

Los archivos `.ipynb` incluyen los outputs (gráficos, tablas) dentro del archivo JSON. Un `git diff` de un notebook modificado es ilegible. Con scripts `.py`, los diffs son limpios y el historial de cambios es útil.

Spyder da lo mismo que Jupyter en lo útil

Lo que hace atractivo a Jupyter para análisis interactivo — ejecutar código por secciones, ver resultados inmediatamente, inspeccionar variables — también lo tiene Spyder:

Jupyter	Spyder equivalente
Celdas de código	Celdas `# %%`
Output inline	Consola IPython
Variable inspector	Variable Explorer (más completo)
Narrativa en Markdown	Comentarios en el código

La diferencia es que en Spyder el código vive en un archivo `.py` limpio, no en un JSON con outputs embebidos.

Cuándo sí usar Jupyter

Jupyter tiene ventajas reales para comunicar resultados: mezclar texto explicativo, código y gráficos en un documento que otros pueden leer y re-ejecutar. Es el formato estándar para papers reproducibles y tutoriales. Si el objetivo es compartir un análisis con alguien que no va a modificar el código, un notebook bien hecho es ideal. Si el objetivo es construir un pipeline que procese datos y se mantenga en el tiempo, un script `.py` es mejor.

3.2 Comparativa directa

	Spyder	VSCode
Variable Explorer	Integrado, visual, con doble clic en DataFrames	Requiere modo Jupyter interactivo
Celdas `# %%`	Nativo, `Ctrl+Enter` / `Shift+Enter`	Con extensión Python, funciona igual
Autocompletado	Básico	Avanzado (Pylance: tipos, docstrings, imports)
Navegación entre archivos	Limitada	Ir a definición entre módulos, búsqueda global
Git	No integrado	Diff visual, historial, blame por línea
Terminal	Consola IPython	Terminal bash/WSL real
Backend Qt5	Conflicto de event loop	Sin conflicto (Electron ≠ Qt)
Refactoring	Básico	Renombrar en todo el proyecto, mover código
Curva de aprendizaje	Baja	Media
Instalación	Anaconda	Descarga + extensión Python

3.3 La distinción más importante: pipeline interactivo vs estático

La diferencia real entre Spyder y VSCode no es nivel de experiencia — es el tipo de pipeline:

Pipeline interactivo: el procesamiento requiere decisiones humanas en el camino. Cargas los datos, inspeccionas, decides dónde cortar la serie, identificas un offset, limpias anomalías, vuelves a graficar. En este flujo, el Variable Explorer de Spyder no es una rueda de entrenamiento — es una herramienta de trabajo. Ver el DataFrame en tiempo real, hacer doble clic para abrirlo como tabla, comparar shapes antes y después de un filtro: todo eso acelera el trabajo de procesamiento interactivo.

Pipeline estático: el código corre de principio a fin sin intervención. Leer archivos, procesar, generar figuras, exportar resultados. Aquí Spyder no agrega valor — un script ejecutado desde la terminal de VSCode hace exactamente lo mismo con mejor soporte de proyecto y Git.

3.4 Cuándo quedarse en Spyder

- El procesamiento requiere inspeccionar y tomar decisiones sobre los datos (offset, recortes, limpieza manual)
- Usas el Variable Explorer activamente para ver el estado intermedio de arrays y DataFrames
- El flujo es exploratorio: cargar, graficar, ajustar, volver a graficar
- Trabajas con un dataset por vez y el script cabe en un archivo

3.5 Cuándo usar VSCode

- El pipeline corre completo sin intervención (genera figuras, exporta, automatiza)
- El proyecto tiene varios módulos que se importan entre sí
- Usas Git activamente y quieres diffs sin salir del editor
- Desarrollas herramientas con interfaz (matplotlib interactivo, Qt5) sin conflictos de event loop
- Necesitas conectarte a un servidor remoto (extensión Remote-SSH)
- El código ya está maduro y entra en fase de mantenimiento

3.6 Configurar VSCode para análisis de datos

3.6.1 Instalación mínima

1. Descargar VSCode: code.visualstudio.com
2. Instalar extensiones:
    - **Python** (Microsoft) — soporte base
    - **Pylance** — autocompletado avanzado con inferencia de tipos

3.6.2 Seleccionar el intérprete de Python

`Ctrl+Shift+P` → ”Python: Select Interpreter” → elegir el Python de Anaconda (`conda base` o el entorno que uses).

3.6.3 Celdas `# %%` en VSCode

Las celdas funcionan exactamente igual que en Spyder. Con la extensión Python instalada:

```
# %% Cargar datos
import pandas as pd
df = pd.read_csv('corrientes.csv')

# %% Graficar
import matplotlib.pyplot as plt
plt.plot(df['velocidad'])
plt.show()
```

- `Shift+Enter` ejecuta la celda y avanza
- `Ctrl+Enter` ejecuta la celda sin avanzar
- Se abre un panel "Jupyter Interactive" donde aparecen los outputs

3.6.4 Variable Explorer equivalente

El panel Jupyter Interactive muestra las variables activas. También se puede:

```
# En la celda, inspeccionar directamente
df.head()        # muestra las primeras filas como tabla
df.dtypes        # tipos de columnas
df.describe()    # estadísticas
```

Para abrir un DataFrame como tabla visual: instalar la extensión **Data Wrangler** (Microsoft).

3.6.5 Configurar el backend de matplotlib

En VSCode, el backend inline se activa automáticamente en el modo interactivo. Para ventanas separadas:

```
%matplotlib tk    # Tkinter — funciona bien
%matplotlib qt5   # Qt5 — también funciona bien (sin conflicto con VSCode)
```

3.6.6 Atajos equivalentes a Spyder

Ejecución y edición

Acción	Spyder	VSCode
Ejecutar celda	`Ctrl+Enter`	`Shift+Enter`
Ejecutar celda y avanzar	`Shift+Enter`	`Shift+Enter`
Ejecutar selección	`F9`	`Shift+Enter` (con selección)
Comentar / descomentar	`Ctrl+1`	`Ctrl+/`

Navegación y búsqueda

Acción	Spyder	VSCode
Ir a definición	`Ctrl+G`	`F12`
Buscar en archivo	`Ctrl+F`	`Ctrl+F`
Buscar en todo el proyecto	—	`Ctrl+Shift+F`
Paleta de comandos	—	`Ctrl+Shift+P`

3.6.7 Terminal integrado

`Ctrl+ñ` (o 'Ctrl+") abre una terminal bash/WSL real dentro de VSCode. Útil para correr scripts completos, git, o herramientas de línea de comandos:

```
python generar_informe.py
bash publicar.sh
git log --oneline -10
```

3.7 Jupyter en VSCode

VSCode soporta notebooks `.ipynb` nativamente. Para abrir o crear uno: `Ctrl+Shift+P` → "Create new Jupyter Notebook".

La ventaja sobre JupyterLab clásico: el editor de VSCode (autocompletado, Pylance) funciona dentro de las celdas del notebook.

3.8 Flujo mixto recomendado

No es necesario elegir uno y abandonar el otro. La combinación más práctica:

1. **Procesamiento interactivo en Spyder**: inspeccionar datos, identificar problemas, tomar decisiones (offset, recortes, limpieza). El Variable Explorer hace este trabajo más rápido.
2. **Pipeline automático en VSCode + terminal**: cuando el procesamiento ya está definido y corre sin intervención. Ejecutar desde terminal, gestionar Git, mantener el proyecto multi-archivo.
3. **Git en VSCode**: commits, diffs visuales, historial — independientemente de dónde se escribió el código.

El código funciona igual en ambos entornos. Lo que cambia es qué tan cómodo es cada flujo de trabajo según la etapa del proyecto.

3.8.1 Cuándo salir del IDE y correr desde terminal

Hay casos en que Spyder y VSCode se interponen en vez de ayudar:

- **Script con menú de procesamiento** (`input()` en un loop): Spyder ejecuta en un kernel IPython que no maneja bien la entrada interactiva en modo batch.
- **GUI propia con matplotlib o Tkinter**: el IDE ya tiene un event loop corriendo; abrir otro genera conflictos. TkAgg suele funcionar en Spyder con `%matplotlib tk`, pero Qt5Agg frecuentemente no.
- **Combinación de ambos** (menú + ventana gráfica): ningún backend funciona de forma confiable dentro del IDE.

En esos casos la solución es correr el script directamente desde una terminal:

```
# PowerShell o cmd (usa el Python de Windows, TkAgg nativo)
python mi_script.py

# Terminal WSL (requiere WSLg en Windows 11, o un servidor X en Windows 10)
python mi_script.py
```

El script toma control del proceso completo: puede pedir input, abrir ventanas y cerrarlas sin conflicto con el IDE. Ver cap. 10b para el patrón `plt.ion()` + `plt.pause()` que permite combinar un menú de texto con figuras interactivas.

4 Entornos virtuales con conda

4.1 ¿Qué es un entorno virtual?

Un entorno virtual es una instalación aislada de Python con sus propios paquetes. Sin entornos, todos los proyectos comparten los mismos paquetes instalados en el Python global — lo que inevitablemente genera conflictos: el proyecto A necesita numpy 1.24, el proyecto B necesita numpy 2.0, y no pueden coexistir en la misma instalación.

Con un entorno por proyecto, cada uno tiene sus propias versiones y los conflictos desaparecen. También permite compartir exactamente qué versiones se usaron para producir un análisis — lo que hace el trabajo reproducible.

4.2 conda vs pip

Anaconda incluye dos gestores de paquetes:

	conda	pip
Qué instala	Python + librerías C/Fortran	Paquetes Python
Resuelve dependencias binarias	Sí	No
Crea entornos virtuales	`conda create`	`venv`
Fuente de paquetes	Anaconda / conda-forge	PyPI

Para paquetes científicos — NumPy, SciPy, GDAL, rasterio — conda maneja mejor las dependencias compiladas. Para paquetes de Python puro que no están en conda, pip funciona igual.

Regla práctica: instalar con conda lo que esté disponible; usar pip para el resto. Evitar mezclar en el mismo entorno si es posible.

4.3 Crear y activar un entorno

```
# Crear un entorno con Python 3.11
conda create -n oceanografia python=3.11

# Activar
conda activate oceanografia

# El prompt cambia para indicar qué entorno está activo:
# (oceanografia) $
```

Una vez activado, todos los `conda install` y `pip install` afectan solo a ese entorno — el sistema global queda intacto.

```
# Desactivar (volver al entorno base)
conda deactivate
```

4.4 Instalar paquetes

```
# Instalar varios paquetes a la vez
conda install numpy pandas matplotlib scipy

# conda-forge tiene versiones más actualizadas de paquetes geoespaciales
conda install -c conda-forge xarray rasterio geopandas

# pip para lo que no está en conda
pip install python-docx pytesseract tqdm pyyaml

# Ver qué hay instalado en el entorno activo
conda list
```

4.5 Listar y eliminar entornos

```
conda env list                          # ver todos los entornos
conda env remove -n nombre_entorno      # eliminar un entorno completo
```

4.6 Exportar e importar entornos

La forma de hacer un análisis reproducible y compartible es exportar el entorno:

```
# Exportar el entorno activo
conda env export > environment.yml

# Recrear el entorno exacto en otro equipo
conda env create -f environment.yml
conda activate oceanografia
```

El archivo `environment.yml` resultante se ve así:

```
name: oceanografia
channels:
  - conda-forge
  - defaults
dependencies:
  - python=3.11
  - numpy=2.0.1
  - pandas=2.2.2
  - matplotlib=3.9.1
  - scipy=1.13.1
  - xarray=2024.7.0
```

```
  - rasterio=1.3.10
  - pip:
    - python-docx==1.1.2
    - tqdm==4.66.4
    - pyyaml==6.0.1
```

Compartir este archivo con un colaborador garantiza que tiene exactamente las mismas versiones. Es más confiable que una lista de `pip install` en un README.

4.7 El entorno base — por qué no instalar todo ahí

Anaconda viene con un entorno `base` preinstalado con Python y los paquetes científicos más comunes. Es tentador instalar todo ahí, pero tiene una desventaja: cada paquete nuevo puede romper la compatibilidad con los que ya estaban. Después de varios meses de instalaciones, el entorno base se vuelve inconsistente y difícil de reparar.

```
# Mal: instalar todo en base
conda install nuevo_paquete   # puede romper numpy o pandas que ya estaban

# Bien: crear un entorno separado
conda create -n mi_proyecto python=3.11
conda activate mi_proyecto
conda install nuevo_paquete   # solo afecta este entorno
```

Recomendación: reservar `base` solo para conda mismo. Un entorno para el trabajo de este manual, otro para proyectos distintos si tienen dependencias incompatibles.

4.8 Actualizar paquetes

```
# Actualizar un paquete específico
conda update numpy

# Actualizar todos los paquetes del entorno activo
conda update --all

# Actualizar conda mismo (desde el entorno base)
conda update conda
```

4.9 Conectar el entorno a Spyder

Spyder usa su propio intérprete por defecto. Para que use el entorno `oceanografia`:

1. Con el entorno activo, instalar el kernel que Spyder necesita:

```
conda activate oceanografia
conda install spyder-kernels
```

2. En Spyder: **Tools → Preferences → Python interpreter → Use the following interpreter**
3. Pegar la ruta del Python del entorno:
   - Windows: `C:\Users\Usuario\anaconda3\envs\oceanografia\python.exe`
   - Linux/WSL: `/home/usuario/anaconda3/envs/oceanografia/bin/python`
4. Reiniciar el kernel de Spyder (`Ctrl+.`).

El Variable Explorer y la consola IPython de Spyder ahora usan los paquetes del entorno, no los del sistema.

4.10 Conectar el entorno a VSCode

`Ctrl+Shift+P` → **Python: Select Interpreter** → elegir `oceanografia` de la lista. VSCode detecta automáticamente los entornos conda instalados.

4.11 Entorno recomendado para este manual

Para seguir todos los capítulos del manual sin problemas de dependencias:

```
conda create -n oceanografia python=3.11
conda activate oceanografia

conda install -c conda-forge numpy pandas matplotlib scipy xarray rasterio geopandas

pip install python-docx pytesseract tqdm pyyaml
```

```
# Guardar para compartir o reproducir
conda env export > environment.yml
```

[>] Reinstalación limpia — Si el entorno queda inconsistente después de muchas instalaciones y desinstalaciones, la solución más rápida es eliminarlo y recrearlo desde el environment.yml: conda env remove -n oceanografia y luego conda env create -f environment.yml. Tarda unos minutos pero garantiza un entorno limpio.

5 Sintaxis básica

Python es un lenguaje de tipado dinámico e indentado. No necesita declarar tipos de variables ni usar llaves `{}` para delimitar bloques — la estructura del código se define por la indentación.

5.1 Variables y tipos

Python detecta el tipo de una variable automáticamente al asignarla. No es necesario declararlo. Los tipos fundamentales son:

5.1.1 float — número decimal

Representa cualquier número con parte decimal. Internamente usa 64 bits (doble precisión), lo que da ~15 dígitos significativos de precisión.

```
velocidad   = 3.5       # float
temperatura = -1.8      # float (puede ser negativo)
proporcion  = 0.73      # float entre 0 y 1

type(velocidad)   # <class 'float'>
```

Se usa para mediciones físicas: velocidades, temperaturas, coordenadas, profundidades. Cualquier número que pueda tener decimales debe ser float.

[!] Precisión de float — Los floats tienen un error de representación inherente. 0.1 + 0.2 en Python da 0.30000000000000004, no 0.3. Esto rara vez afecta el análisis de datos, pero sí puede causar sorpresas en comparaciones exactas. Usar round() o np.isclose() en vez de == para comparar floats.

5.1.2 int — número entero

Número sin parte decimal. En Python 3 no tiene límite de tamaño.

```
n_muestras  = 186       # int
profundidad = 7         # int (metros exactos)
indice      = 0         # int — los índices siempre son int

type(n_muestras)   # <class 'int'>
```

Se usa para contadores, índices, cantidades discretas (número de archivos, de celdas, de meses).

```
# La división entre ints en Python 3 siempre da float
7 / 2      # 3.5   (float)
7 // 2     # 3     (int — división entera)
7 % 2      # 1     (int — resto)
```

5.1.3 str — texto

Secuencia de caracteres. Se define con comillas simples o dobles (equivalentes).

```
nombre_proyecto = "Los Vilos Oct 2025"
unidad          = 'm/s'
ruta            = r'C:\Users\Tomas\datos.csv'   # r"..." ignora las barras invertidas
```

5.1.4 bool — booleano

Solo dos valores posibles: `True` o `False`. Es un caso especial de int (`True == 1`, `False == 0`).

```
datos_validos = True
es_negativo   = velocidad < 0   # bool resultado de una comparación

# Se usa en condiciones
if datos_validos:
    procesar(df)
```

5.1.5 None — ausencia de valor

Representa ”sin valor”. Equivalente a `NULL` en otras lenguas.

```
resultado = None   # aún no calculado

# Verificar si algo es None
if resultado is None:
    print("Aún no hay resultado")
```

5.1.6 Resumen y conversión entre tipos

```
# Verificar tipo
type(3.5)     # <class 'float'>
type(23)      # <class 'int'>
type("texto") # <class 'str'>
type(True)    # <class 'bool'>

# Convertir
int(3.9)      # 3     — trunca, no redondea
float(23)     # 23.0
```

```
str(186)      # '186'
int("23")     # 23    — solo si el string es un número válido
bool(0)       # False — 0 es falso, cualquier otro número es True
bool("")      # False — string vacío es falso
```

5.1.7 ¿Cuándo importa el tipo?

```
# Concatenar string con número da error
nombre = "Profundidad: " + 7          # TypeError
nombre = "Profundidad: " + str(7)     # OK: "Profundidad: 7"
nombre = f"Profundidad: {7}"          # OK con f-string (convierte automático)

# Operaciones entre int y float dan float
3 + 1.5    # 4.5 (float)
7 * 2.0    # 14.0 (float)
```

5.2 El punto (.) en Python

El punto es el operador de acceso en Python. Aparece constantemente pero tiene tres usos distintos que conviene distinguir desde el principio.

1. Acceso a módulo — llama una función o clase que vive dentro de un módulo importado:

```
import numpy as np
import pandas as pd

np.array([1, 2, 3])      # función array del módulo numpy
pd.read_csv('datos.csv') # función read_csv del módulo pandas
```

2. Método de objeto — ejecuta una función que pertenece a un objeto específico (los paréntesis indican que es una llamada):

```
velocidades = [0.3, 0.8, 0.1, 0.5]
velocidades.append(0.6)    # método append de la lista
velocidades.sort()         # método sort de la lista

df['vel'].mean()           # método mean de la Series de pandas
df.dropna()                # método dropna del DataFrame
```

3. Atributo de objeto — accede a una propiedad del objeto, sin llamarla (sin paréntesis):

```
df.shape       # (filas, columnas) — es un dato, no una función
df.columns     # lista de nombres de columnas
arr.dtype      # tipo de dato del array NumPy
```

La diferencia entre método y atributo: si tiene paréntesis `()` es una llamada que ejecuta algo; si no los tiene es una propiedad que ya existe.

Encadenamiento — se pueden combinar varios niveles en una sola línea:

```
doc.paragraphs[2].runs[0].text   # atributo del objeto dentro de una lista dentro de
↪  otro objeto
df['vel'].dropna().mean()        # método sobre el resultado de otro método
```

En MATLAB no existe este patrón — las funciones son independientes (`mean(vel)`, `size(df)`). En Python los objetos llevan sus propias funciones consigo.

5.3 Strings (texto)

```
empresa = "Compas Marine"
centro  = "Los Vilos"

# Concatenación
titulo = empresa + " — " + centro

# f-strings (la forma moderna y más legible)
titulo = f"{empresa} — {centro}"

# Formateo con decimales
promedio = 3.93
texto = f"Velocidad promedio: {promedio:.2f} m/s"
# → "Velocidad promedio: 3.93 m/s"

# Métodos útiles
"Los Vilos".upper()       # 'LOS VILOS'
"  texto  ".strip()       # 'texto'
"a,b,c".split(",")        # ['a', 'b', 'c']
"corrientes".replace("e", "a")  # 'corriantes'
```

5.4 Listas

Las listas almacenan secuencias de elementos de cualquier tipo:

```
profundidades = [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]
meses = ["sep", "oct", "nov", "dic", "ene", "feb", "mar"]

# Acceso por índice (empieza en 0, no en 1 como MATLAB)
profundidades[0]    # 3   — primer elemento
profundidades[-1]   # 23  — último elemento
profundidades[-2]   # 21  — penúltimo

# Operaciones
len(profundidades)          # 11
profundidades.append(25)    # agrega al final
profundidades.sort()        # ordena en lugar
sum(profundidades)          # suma
```

5.4.1 Slicing — seleccionar rangos

La sintaxis es `[inicio:fin:paso]`. El índice `fin` **no se incluye**:

```
p = [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]
#    0  1  2  3   4   5   6   7   8   9  10

p[2:5]    # [7, 9, 11]      — índices 2, 3, 4 (el 5 no entra)
p[:4]     # [3, 5, 7, 9]    — desde el inicio hasta el índice 3
p[7:]     # [17, 19, 21, 23] — desde el índice 7 hasta el final
p[::2]    # [3, 7, 11, 15, 19, 23] — uno de cada dos (paso 2)
p[::-1]   # [23, 21, ..., 3]       — invertir la lista
```

En NumPy y pandas el slicing funciona igual y se puede aplicar a filas y columnas:

```
import numpy as np

A = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])

A[0, :]      # [1, 2, 3, 4]   — primera fila, todas las columnas
A[:, 1]      # [2, 6, 10]     — todas las filas, columna 1
A[0:2, 1:3]  # [[2,3],[6,7]]  — filas 0-1, columnas 1-2
```

El equivalente en MATLAB sería `A(1,:)`, `A(:,2)`, `A(1:2, 2:3)` — la diferencia es que Python empieza en 0 y el índice final no se incluye.

5.5 Tuplas

Las tuplas son secuencias **inmutables**: se definen igual que una lista pero con paréntesis, y no se pueden modificar después de creadas.

```
coordenadas = (-31.9, -71.5)          # (lat, lon)
rango       = (0, 23)                 # profundidad mínima y máxima
rgb         = (30, 144, 255)          # color fijo

# Acceso — igual que lista
coordenadas[0]    # -31.9
coordenadas[-1]   # -71.5
```

¿Por qué no simplemente usar una lista? La inmutabilidad comunica intención: si algo es una tupla, el lector del código sabe que ese valor no va a cambiar. También tienen otro uso frecuente: las funciones que devuelven múltiples valores en Python devuelven una tupla.

```
def estadisticas(datos):
    return np.mean(datos), np.std(datos)   # devuelve tupla

media, std = estadisticas(vel)   # desempaquetado automático
```

```
# Diferencia clave con lista
lista = [1, 2, 3]
lista[0] = 99       # OK — las listas son mutables

tupla = (1, 2, 3)
tupla[0] = 99       # TypeError: 'tuple' object does not support item assignment
```

Se usa para: coordenadas geográficas, rangos fijos de profundidad o tiempo, pares clave-valor, y cualquier conjunto de valores relacionados que no deba cambiar.

5.6 Diccionarios

Los diccionarios almacenan pares clave-valor. Son muy usados para configuración:

```
config = {
    "empresa": "Compas Marine",
    "centro":  "Los Vilos",
    "lat":     -31.9,
    "lon":     -71.5,
    "prof_max": 23
}

# Acceso
config["empresa"]          # "Compas Marine"
config.get("lat", None)    # -31.9 (con valor por defecto si no existe)

# Modificar
config["prof_max"] = 25

# Iterar
```

```python
for clave, valor in config.items():
    print(f"{clave}: {valor}")
```

5.7 Indentación

Python usa la indentación (4 espacios) para delimitar bloques. No hay llaves ni `end`:

```python
# Correcto
if velocidad > 0.5:
    print("Velocidad alta")
    print("Revisar datos")

# Error — la indentación inconsistente produce IndentationError
if velocidad > 0.5:
print("Velocidad alta")   # ← falta indentación
```

5.8 Comentarios

```python
# Comentario de una línea

velocidad_max = 0.6  # m/s — comentario al final de línea

"""
Comentario de
múltiples líneas
(también se usa como docstring de funciones)
"""
```

5.9 Operadores

```python
# Aritméticos
3 + 2     # 5
10 / 3    # 3.333...
10 // 3   # 3 (división entera)
10 % 3    # 1 (módulo / resto)
2 ** 3    # 8 (potencia)

# Comparación
5 > 3     # True
5 == 5    # True (igualdad, no asignación)
5 != 4    # True

# Lógicos
True and False   # False
True or False    # True
```

```python
not True         # False

# Útil con pandas: operadores por elemento
velocidad > 0.1          # Series de booleanos
(vel > 0.1) & (dir < 90) # AND elemento a elemento
```

5.10 Referencias y copias

En MATLAB, asignar una variable siempre crea una copia independiente:

```python
% MATLAB
b = a(:, 1:3);   % b es una copia — modificar b no afecta a
```

En Python, asignar una variable **no copia** los objetos mutables (listas, arrays, DataFrames) — crea una segunda referencia al mismo objeto en memoria:

```python
a = [1, 2, 3, 4, 5]
b = a            # b apunta al mismo objeto que a
b[0] = 99
print(a)         # [99, 2, 3, 4, 5] — a también cambió
```

Para obtener una copia real, hay que pedirla explícitamente:

```python
# Listas
b = a.copy()
b = a[:]         # slice completo también copia

# NumPy
import numpy as np
A = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8]])

B = A.copy()            # copia completa — equivalente a b = a en MATLAB
B = A[:, 0:3].copy()    # equivalente a b = A(:, 1:3) en MATLAB

# Sin .copy(), un slice de NumPy es una vista del original:
B = A[:, 0:3]    # B comparte memoria con A
B[0, 0] = 99     # modifica A también

# pandas
df2 = df.copy()
```

[!] Vistas en NumPy — Un slice de NumPy sin .copy() es una vista: ocupa cero memoria extra y cualquier modificación afecta al array original. Útil para eficiencia, pero puede generar bugs si no se tiene en cuenta.

5.11 Conversión de tipos

```
int("23")        # 23
float("3.5")     # 3.5
str(186)         # "186"
list((1, 2, 3))  # [1, 2, 3]
```

[!] Error frecuente — En Python 3, dividir dos enteros siempre devuelve float: 7 / 2 = 3.5. Para división entera usa //.

6 Control de flujo

El control de flujo determina qué código se ejecuta y cuándo. En procesamiento oceanográfico se usa constantemente: para filtrar datos, iterar sobre profundidades, manejar errores de lectura, etc.

6.1 Condicionales: if / elif / else

```
velocidad = 0.35

if velocidad >= 0.5:
    categoria = "alta"
elif velocidad >= 0.2:
    categoria = "moderada"
else:
    categoria = "baja"

print(f"Velocidad {categoria}: {velocidad} m/s")
```

6.1.1 Condicional en una línea (ternario)

```
etiqueta = "válido" if velocidad > 0 else "calma"
```

6.2 Bucles: for

El bucle `for` itera sobre cualquier secuencia: listas, rangos, columnas de un DataFrame, archivos, etc.

```
profundidades = [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]

for prof in profundidades:
    print(f"Procesando capa a {prof} m")
```

6.2.1 range()

```
# range(inicio, fin, paso) — fin no se incluye
for i in range(0, 12):        # 0 a 11
    print(i)

for i in range(0, 24, 2):     # 0, 2, 4, ..., 22
    print(i)
```

6.2.2 enumerate() — índice + valor

```python
meses = ["sep", "oct", "nov", "dic", "ene", "feb", "mar"]

for i, mes in enumerate(meses):
    print(f"Mes {i+1}: {mes}")
```

6.2.3 Iterar sobre un diccionario

```python
stats = {"media": 3.93, "maxima": 18.2, "std": 2.8}

for nombre, valor in stats.items():
    print(f"{nombre}: {valor:.2f}")
```

6.3 Bucles: while

```python
intentos = 0
while intentos < 3:
    print(f"Intento {intentos + 1}")
    intentos += 1
```

6.4 break y continue

```python
for prof in profundidades:
    if prof > 15:
        break       # sale del bucle al llegar a 17 m

for prof in profundidades:
    if prof == 9:
        continue    # salta esta iteración, continúa con la siguiente
    print(prof)
```

6.5 List comprehensions

Una forma compacta de construir listas a partir de otra secuencia. Muy usadas en el procesamiento de datos para transformar o filtrar colecciones:

```python
profundidades = [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]

# Sin comprehension
dobles = []
for p in profundidades:
    dobles.append(p * 2)

# Con comprehension (equivalente, más conciso)
dobles = [p * 2 for p in profundidades]
```

```python
# Con filtro
superficiales = [p for p in profundidades if p <= 7]
# → [3, 5, 7]
```

6.5.1 Ejemplo real del pipeline

```python
# Buscar todos los archivos Excel de corrientes en un directorio
import os
archivos = [f for f in os.listdir(carpeta) if f.endswith('.xlsx')]
```

6.5.2 zip() — recorrer dos listas en paralelo

`zip()` empareja dos o más listas elemento a elemento. Devuelve pares `(a, b)` mientras duren ambas listas:

```python
profundidades = [3, 5, 7, 9]
velocidades   = [0.08, 0.12, 0.60, 0.07]

for prof, vel in zip(profundidades, velocidades):
    print(f"{prof} m → {vel:.2f} m/s")

# También funciona para construir un diccionario
prof_vel = dict(zip(profundidades, velocidades))
# {3: 0.08, 5: 0.12, 7: 0.6, 9: 0.07}
```

Si las listas tienen distinto largo, `zip` se detiene en la más corta — sin error, sin aviso.

6.6 Dict comprehensions

Las comprehensions también funcionan para diccionarios:

```python
profundidades = [3, 5, 7, 9, 11]
velocidades   = [0.08, 0.12, 0.60, 0.07, 0.09]

# Diccionario profundidad → velocidad
prof_vel = {prof: vel for prof, vel in zip(profundidades, velocidades)}
# {3: 0.08, 5: 0.12, 7: 0.6, 9: 0.07, 11: 0.09}

# Filtrado: solo profundidades con velocidad alta
alertas = {prof: vel for prof, vel in prof_vel.items() if vel > 0.5}
# {7: 0.6}
```

La forma `{clave: valor for ... in ...}` es análoga a la list comprehension `[valor for ... in ...]` pero produce un diccionario en vez de una lista.

6.7 any() y all()

Dos funciones que verifican condiciones sobre listas o arrays enteros, sin escribir un loop:

```
velocidades = [0.08, 0.12, 0.60, 0.07, 0.09]

any(v > 0.5 for v in velocidades)   # True — ¿alguno supera 0.5?
all(v > 0.0 for v in velocidades)   # True — ¿todos son positivos?
all(v < 1.0 for v in velocidades)   # True — ¿ninguno supera 1 m/s?
```

Con NumPy/Pandas el resultado es directo sin el generador:

```
import numpy as np
vel = np.array([0.08, 0.12, 0.60, 0.07])

np.any(vel > 0.5)   # True
np.all(vel > 0.0)   # True

# Con pandas Series
(df['velocidad'] > 0.5).any()   # True si alguna fila cumple
(df['velocidad'] > 0.0).all()   # True si todas las filas cumplen
```

Muy útil para validar datos antes de procesarlos:

```
assert (df['velocidad'] >= 0).all(), "Hay velocidades negativas — revisar datos"
assert df['tiempo'].notna().all(),   "Hay timestamps nulos"
```

6.8 Manejo de errores: try / except

Fundamental cuando se leen archivos o datos que pueden estar incompletos o corruptos:

```
try:
    df = pd.read_csv('corrientes.csv')
except FileNotFoundError:
    print("Archivo no encontrado")
except Exception as e:
    print(f"Error inesperado: {e}")
```

6.8.1 finally — código que siempre se ejecuta

```
try:
    archivo = open('datos.txt')
    datos = archivo.read()
except FileNotFoundError:
    print("No se encontró el archivo")
finally:
    print("Proceso terminado")  # se ejecuta siempre
```

6.8.2 Ejemplo real del pipeline

En el código de automatización de informes, el try/except se usa para detectar si un archivo de figuras existe antes de intentar insertarlo en el Word:

```
try:
    doc.paragraphs[idx].runs[0].add_picture(ruta_figura, width=ancho)
except Exception as e:
    print(f"  ! No se pudo insertar figura: {e}")
```

6.9 Combinando estructuras

Donde el control de flujo se vuelve útil de verdad es cuando se combinan varias capas. Las estructuras individuales son simples; la habilidad está en encadenarlas para resolver problemas reales.

6.9.1 for + if: filtrar mientras se itera

```
profundidades = [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]
velocidades   = [0.08, 0.09, 0.6, 0.07, 0.08, 0.09, 0.10, 0.09, 0.08, 0.07, 0.06]

# Recorrer dos listas a la vez con zip()
for prof, vel in zip(profundidades, velocidades):
    if vel > 0.5:
        print(f"Alerta en {prof} m: velocidad = {vel:.2f} m/s")
```

`zip()` empareja dos listas elemento a elemento: en cada vuelta del for, `prof` y `vel` corresponden al mismo índice.

6.9.2 for + if + acumulador: encontrar el máximo con contexto

```
maxima = 0
prof_maxima = None

for prof, vel in zip(profundidades, velocidades):
    if vel > maxima:
        maxima = vel
        prof_maxima = prof

print(f"Velocidad máxima: {maxima} m/s a {prof_maxima} m de profundidad")
# → Velocidad máxima: 0.6 m/s a 7 m de profundidad
```

El acumulador (`maxima`, `prof_maxima`) guarda el resultado parcial mientras el loop avanza. Al terminar, tiene la respuesta final.

6.9.3 for + try/except: procesar lotes sin que un error detenga todo

```
import pandas as pd

archivos = ['oct2024.csv', 'nov2024.csv', 'dic_corrupto.csv', 'ene2025.csv']
dataframes = []

for nombre in archivos:
    try:
        df = pd.read_csv(nombre)
        dataframes.append(df)
        print(f"OK: {nombre} ({len(df)} filas)")
    except FileNotFoundError:
        print(f"  ! No encontrado: {nombre}")
    except Exception as e:
        print(f"  ! Error en {nombre}: {e}")

df_total = pd.concat(dataframes)
```

Sin el `try/except`, el primer archivo problemático detendría el script y perderías los datos buenos que venían después. Con él, el loop sigue y al final tienes todo lo que se pudo leer.

6.9.4 for anidado: recorrer filas y columnas

```
profundidades = [5, 10, 20]
meses = ['oct', 'nov', 'dic']

for mes in meses:
    for prof in profundidades:
        # aquí iría la lógica para cada combinación mes × profundidad
        print(f"Procesando {mes} a {prof} m")
```

Útil cuando se necesita hacer algo para cada combinación de dos listas. Ojo: si cada lista tiene N elementos, el loop interno se ejecuta N^2 veces. Con listas grandes, conviene pensar si hay una forma vectorizada (NumPy/Pandas) que sea más eficiente.

6.9.5 Comprehension con condición vs for + if

Cuando el objetivo es construir una lista filtrada, la comprehension es más clara:

```
# for + if (más verboso)
superficiales = []
for p in profundidades:
    if p <= 7:
        superficiales.append(p)

# comprehension (equivalente, más directo)
superficiales = [p for p in profundidades if p <= 7]
```

La comprehension se lee de corrido: "lista de p, para cada p en profundidades, si p ≤ 7". Úsala cuando la lógica cabe en una línea. Si hay condiciones anidadas o el cuerpo del loop hace varias cosas, el for explícito es más fácil de leer y modificar.

6.9.6 tqdm — barra de progreso en loops largos

Cuando un loop procesa decenas o cientos de archivos, no hay feedback visual de cuánto falta. `tqdm` agrega una barra de progreso con una línea de cambio:

```
pip install tqdm    # o: conda install tqdm
```

```
from tqdm import tqdm

archivos = ['oct2024.csv', 'nov2024.csv', 'dic2024.csv', ...]   # 80 archivos

for archivo in tqdm(archivos, desc='Leyendo archivos'):
    df = pd.read_csv(archivo)
    # ...
```

Salida:

```
Leyendo archivos: 100%|██████████| 80/80 [00:12<00:00,  6.4it/s]
```

Funciona con cualquier iterable — no hace falta cambiar el cuerpo del loop. También se puede combinar con `try/except`:

```
for archivo in tqdm(archivos, desc='Procesando', unit='archivo'):
    try:
        procesar(archivo)
    except Exception as e:
        tqdm.write(f'  ! {archivo}: {e}')   # tqdm.write no rompe la barra
```

6.9.7 Patrón completo: lectura + filtrado + reporte

```
import os
import pandas as pd

carpeta = 'datos/'
resultados = []

for archivo in os.listdir(carpeta):
    if not archivo.endswith('.csv'):
        continue   # saltar archivos que no son CSV

    try:
        df = pd.read_csv(os.path.join(carpeta, archivo))
        vel_max = df['velocidad'].max()
        resultados.append({'archivo': archivo, 'vel_max': vel_max})

    except Exception as e:
        print(f"  ! {archivo}: {e}")

# Ordenar por velocidad máxima
resultados.sort(key=lambda x: x['vel_max'], reverse=True)

for r in resultados:
    print(f"{r['archivo']:30s}  vel_max = {r['vel_max']:.2f} m/s")
```

Este patrón — iterar sobre archivos, saltar los irrelevantes, capturar errores, acumular resultados, ordenar y reportar — aparece constantemente en scripts de análisis real.

7 Funciones y módulos

7.1 Funciones

7.1.1 ¿Por qué hacer una función?

Cuando un mismo bloque de código aparece más de una vez — aunque sea con pequeñas variaciones — conviene convertirlo en función. Así se escribe una sola vez, se prueba una sola vez, y si hay que corregirlo se cambia en un solo lugar.

```
# Sin función: repetir la misma lógica para cada mes
vel_oct = datos_oct['velocidad']
media_oct = vel_oct.mean()
std_oct   = vel_oct.std()
max_oct   = vel_oct.max()
print(f"Oct — media={media_oct:.2f}  std={std_oct:.2f}  max={max_oct:.2f}")

vel_nov = datos_nov['velocidad']
media_nov = vel_nov.mean()
std_nov   = vel_nov.std()
max_nov   = vel_nov.max()
print(f"Nov — media={media_nov:.2f}  std={std_nov:.2f}  max={max_nov:.2f}")

# Con función: escribir la lógica una vez
def reportar_estadisticas(df, nombre):
    vel = df['velocidad']
    print(f"{nombre} — media={vel.mean():.2f}  std={vel.std():.2f}
    ↪  max={vel.max():.2f}")

reportar_estadisticas(datos_oct, 'Oct')
reportar_estadisticas(datos_nov, 'Nov')
reportar_estadisticas(datos_dic, 'Dic')
```

La función no solo ahorra líneas — hace el código más fácil de entender porque el nombre `reportar_estadisticas` describe exactamente qué hace.

Una función agrupa código reutilizable bajo un nombre. En un pipeline de procesamiento de datos casi toda la lógica está organizada en funciones: una para leer datos, otra para filtrar, otra para graficar, etc.

```
def calcular_media_vectorial(velocidades, direcciones):
    """Calcula la dirección y velocidad resultante del vector medio."""
    import numpy as np
    u = velocidades * np.sin(np.radians(direcciones))
    v = velocidades * np.cos(np.radians(direcciones))
    u_media = np.mean(u)
    v_media = np.mean(v)
    vel_media = np.sqrt(u_media**2 + v_media**2)
    dir_media = np.degrees(np.arctan2(u_media, v_media)) % 360
```

```
    return vel_media, dir_media
```

7.1.2 Sintaxis básica

```
def nombre_funcion(parametro1, parametro2):
    # cuerpo de la función
    resultado = parametro1 + parametro2
    return resultado

# Llamar la función
total = nombre_funcion(3, 5)   # total = 8
```

7.1.3 Parámetros por defecto

```
def leer_corrientes(ruta, sep=';', skiprows=0, encoding='utf-8'):
    import pandas as pd
    return pd.read_csv(ruta, sep=sep, skiprows=skiprows, encoding=encoding)

# Uso mínimo (usa los valores por defecto)
df = leer_corrientes('datos.csv')

# Sobreescribir un parámetro específico
df = leer_corrientes('datos.csv', sep=',', skiprows=2)
```

7.1.4 Múltiples valores de retorno

Python puede retornar múltiples valores como una tupla:

```
def estadisticas(datos):
    import numpy as np
    return np.mean(datos), np.max(datos), np.std(datos)

media, maximo, desviacion = estadisticas(velocidades)
```

7.1.5 Argumentos con nombre (kwargs)

```
def guardar_figura(fig, nombre, dpi=150, formato='png'):
    ruta = f"figuras/{nombre}.{formato}"
    fig.savefig(ruta, dpi=dpi, bbox_inches='tight')
    print(f"Figura guardada: {ruta}")

# Se pueden pasar en cualquier orden si se nombran
guardar_figura(fig, "rosa_corrientes", formato='pdf', dpi=300)
```

7.2 Módulos e imports

Un módulo es un archivo `.py` que contiene funciones, clases y variables. Las librerías como NumPy, Pandas y Matplotlib son colecciones de módulos.

7.2.1 Formas de importar

```
# Importar la librería completa
import numpy

# Con alias (convención estándar)
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Importar solo lo que se necesita
from datetime import datetime, timezone
from pathlib import Path
```

7.2.2 Importar funciones propias

Si tienes tus funciones en otro archivo, por ejemplo `utils.py`:

```
# utils.py
def convertir_uv(velocidad, direccion):
    import numpy as np
    u = velocidad * np.sin(np.radians(direccion))
    v = velocidad * np.cos(np.radians(direccion))
    return u, v
```

Se importa así desde otro script:

```
from utils import convertir_uv

u, v = convertir_uv(0.35, 45)
```

7.2.3 sys.path — importar desde otra carpeta

Cuando el archivo que necesitas está en otro directorio, hay que agregar esa ruta al path de Python:

```
import sys
sys.path.insert(0, '/ruta/a/mi/libreria')

from mi_modulo import mi_funcion
```

Esto se usa constantemente en el pipeline de procesamiento, por ejemplo en `run_pipeline.py` para cargar los módulos de cada instrumento.

7.2.4 importlib — carga dinámica

Cuando la ruta del módulo se conoce solo en tiempo de ejecución (por ejemplo, depende de qué proyecto se está procesando), se usa `importlib`:

```
import importlib.util

def cargar_modulo(ruta_archivo):
    spec = importlib.util.spec_from_file_location("modulo", ruta_archivo)
    modulo = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(modulo)
    return modulo

# Cargar el script de notas del analista
notas = cargar_modulo('/proyectos/LosVilos/notas_informe.py')
parrafo_viento = notas.parrafo_viento
```

7.3 Organizar código en módulos

A medida que los scripts crecen, conviene separar el código en archivos temáticos. Un ejemplo de estructura para un proyecto de análisis oceanográfico:

```
ocean_data_analysis/
├── __init__.py
├── read_data.py          ← funciones de lectura
├── preprocessing.py      ← filtros y correcciones
├── wave_processing.py    ← análisis de oleaje
├── wind_figures.py       ← figuras de viento
├── current_z_figures.py  ← figuras de corrientes
└── extreme_events.py     ← análisis de eventos extremos
```

El archivo `__init__.py` (puede estar vacío) le indica a Python que esa carpeta es un paquete importable.

7.4 Scope (alcance de variables)

Las variables definidas dentro de una función son locales — no existen fuera de ella:

```
def procesar():
    resultado = 42      # variable local
    return resultado

procesar()
print(resultado)        # NameError: resultado no existe aquí
```

Para compartir datos entre funciones, lo correcto es usar el valor de retorno, no variables globales.

[>] Buena práctica — Escribe funciones pequeñas que hagan una sola cosa. Es más fácil probarlas, reutilizarlas y entenderlas. Si una función supera las 50 líneas, probablemente conviene dividirla.

8 Errores y debugging

Cuando Python encuentra un problema, detiene la ejecución y muestra un **traceback** — el rastro del error. Saber leer ese mensaje es la habilidad más práctica para trabajar con código: ahorra tiempo y evita buscar soluciones al problema equivocado.

8.1 Anatomía de un traceback

```
import pandas as pd

def cargar_datos(ruta):
    df = pd.read_csv(ruta)
    return df['velocidad'].mean()

resultado = cargar_datos('datos.csv')
```

Si `datos.csv` no existe, Python muestra:

```
Traceback (most recent call last):
  File "script.py", line 7, in <module>
    resultado = cargar_datos('datos.csv')
  File "script.py", line 4, in cargar_datos
    df = pd.read_csv(ruta)
  File ".../pandas/io/parsers.py", line 912, in read_csv
    return _read(filepath_or_buffer, kwds)
FileNotFoundError: [Errno 2] No such file or directory: 'datos.csv'
```

Cómo leerlo:

1. **Ignorar el medio** — las líneas del traceback van de la llamada más externa a la más interna. Lo útil está al principio (tu código) y al final (el error).
2. **Última línea** — el tipo de error y el mensaje. Aquí: `FileNotFoundError: No such file or directory: 'datos.csv'`. Es lo primero que hay que leer.
3. **Tu código en el traceback** — buscar las líneas que referencian tu archivo (`script.py`), no las de librerías. Ahí está el origen del problema.

8.2 Errores más comunes

8.2.1 `FileNotFoundError` — archivo no encontrado

```
df = pd.read_csv('datos.csv')
# FileNotFoundError: No such file or directory: 'datos.csv'
```

Causa: la ruta no existe o el script corre desde un directorio distinto al que contiene el archivo.

Diagnóstico:

```
import os
print(os.getcwd())          # directorio actual desde donde corre el script
print(os.path.exists('datos.csv'))  # True/False
```

8.2.2 `KeyError` — clave o columna inexistente

```
df['velocidat']
# KeyError: 'velocidat'
```

Causa: el nombre de columna está mal escrito, tiene espacios ocultos, o la columna no existe en ese DataFrame.

Diagnóstico:

```
print(df.columns.tolist())   # ver exactamente qué columnas hay
```

8.2.3 `IndexError` — índice fuera de rango

```
lista = [1, 2, 3]
lista[5]
# IndexError: list index out of range
```

Causa: se intenta acceder a una posición que no existe. Recuerda que el índice máximo es `len(lista) - 1`.

8.2.4 `TypeError` — tipo incorrecto para la operación

```
"Profundidad: " + 7
# TypeError: can only concatenate str (not "int") to str
```

Causa: se mezclan tipos incompatibles. Solución: convertir explícitamente.

```
"Profundidad: " + str(7)      # OK
f"Profundidad: {7}"           # OK con f-string
```

8.2.5 `ValueError` — valor incorrecto aunque el tipo es correcto

```
int("3.5")
# ValueError: invalid literal for int() with base 10: '3.5'

pd.to_datetime("no es una fecha")
# ValueError: ...
```

Causa: el tipo es correcto (es un string) pero el contenido no es válido para la operación pedida.

8.2.6 `AttributeError` — atributo o método inexistente

```
lista = [1, 2, 3]
lista.mean()
# AttributeError: 'list' object has no attribute 'mean'
```

Causa: se llama un método que no existe en ese tipo. `.mean()` existe en arrays NumPy y Series de pandas, no en listas de Python.

8.2.7 `NameError` — variable no definida

```
print(resultado)
# NameError: name 'resultado' is not defined
```

Causa: la variable nunca fue asignada, o la celda que la define no se ejecutó todavía.

8.2.8 `IndentationError` — indentación incorrecta

```
if velocidad > 0:
print("positivo")
# IndentationError: expected an indented block
```

Causa: falta la indentación después de `if`, `for`, `def`, etc.

8.2.9 `ModuleNotFoundError` — librería no instalada

```
import xarray
# ModuleNotFoundError: No module named 'xarray'
```

Solución:

```
pip install xarray
# o
conda install xarray
```

8.3 Debugging en Spyder

Para errores que no se entienden solo con el traceback, el debugger permite pausar el código en cualquier punto e inspeccionar el estado de las variables.

8.3.1 Breakpoints

Un **breakpoint** es una marca que le dice a Python ”pausa aquí”. En Spyder:

- Hacer clic en el número de línea donde se quiere pausar (aparece un punto rojo)
- O posicionar el cursor en la línea y presionar `F12`

Luego ejecutar el script con **F5** en modo debug (botón con el insecto , o menú **Debug → Debug file**).

8.3.2 Controles del debugger

Acción	Atajo	Descripción
Continuar hasta el próximo breakpoint	F9	Sigue corriendo
Ejecutar línea actual	F10	Avanza una línea
Entrar a la función	F11	Entra al interior de la función llamada
Salir de la función	Shift+F11	Sale de la función actual
Detener debugging	Shift+F12	Termina el modo debug

8.3.3 Inspeccionar variables en el debugger

Mientras el código está pausado, el **Variable Explorer** muestra el estado actual de todas las variables. También se puede escribir en la consola IPython para evaluar expresiones:

```
# Con el código pausado, en la consola:
df.shape
df['velocidad'].isna().sum()
type(resultado)
```

8.4 Estrategias de debugging

8.4.1 Leer el error antes de buscar en Google

El mensaje de error dice exactamente qué pasó y dónde. `KeyError: 'velocidad'` es más útil que buscar "pandas error". Leer el traceback completo toma 10 segundos y frecuentemente resuelve el problema.

8.4.2 Reducir el problema

Si el error ocurre en un loop que procesa 100 archivos, ejecutar primero con uno solo:

```
# En vez de:
for archivo in archivos:
    procesar(archivo)

# Primero probar con el primero:
procesar(archivos[0])
```

8.4.3 Imprimir el estado intermedio

El método más simple y más efectivo:

```
def procesar(df):
    print(f"shape: {df.shape}")        # ¿cuántas filas/columnas?
    print(f"columnas: {df.columns.tolist()}")
    print(f"NaN: {df.isna().sum()}")
    resultado = df['velocidad'].mean()
    print(f"resultado: {resultado}")
    return resultado
```

8.4.4 Verificar suposiciones

Los errores frecuentemente ocurren porque el dato no tiene el formato que se espera:

```
# Antes de operar, verificar
print(type(df['velocidad'].iloc[0]))   # ¿es float o string?
print(df['velocidad'].dtype)           # ¿numpy float o object?
print(df.index[:3])                    # ¿cómo se ve el índice?
```

8.4.5 `assert` para detectar condiciones inesperadas

```
df = pd.read_csv('datos.csv')
assert len(df) > 0, "El DataFrame está vacío"
assert 'velocidad' in df.columns, f"Columna 'velocidad' no encontrada. Columnas:
 ↪  {df.columns.tolist()}"
```

Si la condición es falsa, Python lanza un `AssertionError` con el mensaje — más claro que esperar a que falle más adelante.

8.5 logging — alternativa a print para pipelines

`print` es suficiente para exploración interactiva. En un pipeline automático que corre sin supervisión, `logging` tiene ventajas concretas: cada mensaje tiene timestamp, nivel de severidad, y se puede escribir a un archivo sin tocar el código.

```
import logging

# Configuración básica — una sola vez al inicio del script
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s  %(levelname)-8s  %(message)s',
    datefmt='%H:%M:%S',
    handlers=[
        logging.StreamHandler(),                        # consola
        logging.FileHandler('pipeline.log', 'w')       # archivo
    ]
```

```
)

logger = logging.getLogger(__name__)
```

```
# Usar en el código en vez de print
logger.debug('Detalle interno útil para debugging')     # solo visible en nivel DEBUG
logger.info('Leyendo archivo corrientes_oct2024.csv')    # progreso normal
logger.warning('Columna "dir" tiene 3% de NaN')          # algo inesperado pero no
↪  fatal
logger.error('No se encontró el archivo de viento')     # error que impide continuar
```

```
# Patrón para el pipeline de archivos
for archivo in archivos:
    try:
        df = pd.read_csv(archivo)
        logger.info(f'OK  {archivo}  ({len(df)} filas)')
    except FileNotFoundError:
        logger.warning(f'No encontrado: {archivo}')
    except Exception as e:
        logger.error(f'Error en {archivo}: {e}')
```

Salida en consola y en `pipeline.log`:

```
14:32:01  INFO      OK  oct2024.csv  (8640 filas)
14:32:02  INFO      OK  nov2024.csv  (8352 filas)
14:32:02  WARNING   No encontrado: dic2024.csv
14:32:03  INFO      OK  ene2025.csv  (8928 filas)
```

Cuándo usar logging en vez de print: cuando el script corre automáticamente (por ejemplo, desde un cron o un bat), cuando se necesita guardar el historial de ejecuciones, o cuando se quiere poder aumentar el nivel de detalle (`DEBUG`) sin modificar el código.

8.6 Reiniciar el kernel y el estado de la sesión

8.6.1 Limpiar consola vs reiniciar kernel

Son dos operaciones distintas que confunden al principio:

Limpiar la consola (`Ctrl+L`, o escribir `%clear`) borra el texto visible — outputs, errores impresos, historial. El namespace de variables **no cambia**: `df`, `velocidades`, `resultado` siguen existiendo en memoria.

Reiniciar el kernel (`Ctrl+.` en Spyder, o menú **Consolas → Reiniciar kernel**) termina el proceso Python y arranca uno nuevo. El namespace queda completamente vacío. Es equivalente a cerrar Python y abrirlo de nuevo.

8.6.2 El problema del estado acumulado

En Spyder se ejecutan celdas en cualquier orden, se prueban cosas en la consola y se vuelve a correr solo una parte del script. Con el tiempo el namespace acumula variables de corridas anteriores que el script actual nunca definió — pero como ya están en memoria, no aparece error.

El síntoma clásico:

```
# El script "funciona" en Spyder...
resultado = calcular(df)   # df existe porque la cargaste antes en la consola

# ...pero falla al correrlo con F5 desde cero, o desde la terminal:
# NameError: name 'df' is not defined
```

Regla práctica: si el script funciona celda por celda pero falla al correr con `F5` desde cero, hay estado acumulado. Reiniciar el kernel y volver a correr `F5` revela qué falta realmente.

8.6.3 Cuándo reiniciar el kernel

Situación	Acción
Script falla con `NameError` al correr `F5` completo	Reiniciar kernel → `F5`
Modificaste una función y el cambio no tiene efecto	Reiniciar kernel o `%reset`
Cambiaste `%matplotlib` backend y no responde	Reiniciar kernel
Variables grandes consumen demasiada RAM	Reiniciar kernel
Quieres verificar que el script es reproducible	Reiniciar kernel → `F5`

8.6.4 %reset — limpiar el namespace sin reiniciar

`%reset` limpia las variables pero mantiene el intérprete activo. Más rápido que reiniciar el kernel cuando solo se quiere empezar con un namespace limpio:

```
%reset        # pide confirmación
%reset -f     # fuerza sin confirmación (más cómodo)
```

8.6.5 Cambios en el código y cuándo re-ejecutar

Si modificás una función en el mismo script: re-ejecutar la celda que la define reemplaza la definición en el namespace. Con `F5` se re-ejecuta todo el script y todas las definiciones quedan actualizadas.

Si modificás un módulo importado (`from utils import calcular`): volver a ejecutar el `import` no recarga el módulo — Python lo tiene en caché. Hay que forzar la recarga:

```
import importlib
import utils                  # importación inicial

# Después de editar utils.py en el editor:
importlib.reload(utils)
from utils import calcular    # ahora usa la versión actualizada
```

O desde la consola de Spyder, más directo:

```
%run utils.py    # ejecuta el archivo y actualiza el namespace con lo que define
```

Si modificás una clase: las instancias ya creadas no se actualizan al recargar el módulo. Hay que reiniciar el kernel para que las nuevas instancias usen la clase modificada.

Regla general: si cambiaste el código y el comportamiento no cambió, es casi siempre porque Python sigue usando la versión anterior en caché. La solución más segura y más rápida de verificar es siempre: reiniciar kernel → `F5`.

8.7 Errores silenciosos

Los más difíciles de detectar son los que no producen error pero dan un resultado incorrecto. Ejemplos frecuentes:

```
# División entera en vez de float (Python 2 vs 3)
7 / 2    # 3.5 en Python 3 — correcto
7 // 2   # 3   — si esto era lo que querías, bien; si no, error silencioso

# NaN que se propagan sin aviso
velocidad = np.array([1.0, np.nan, 3.0])
print(velocidad.mean())   # nan — no hay error, pero el resultado es inútil
print(np.nanmean(velocidad))  # 2.0 — correcto

# Filtrado que elimina más datos de los esperados
df_filtrado = df[df['velocidad'] > 0.5]
# Si df_filtrado está vacío, las operaciones siguientes dan NaN o error lejano
print(f"Filas después de filtrar: {len(df_filtrado)}")  # verificar siempre
```

9 NumPy

NumPy es la librería fundamental para cálculo numérico en Python. Su estructura central es el **array**: un arreglo multidimensional de valores del mismo tipo, mucho más eficiente que una lista de Python para operaciones matemáticas.

```
import numpy as np
```

9.1 Arrays vs listas de Python

La pregunta más común al empezar con NumPy es: ¿cuándo usar un array y cuándo una lista?

	Lista de Python	Array de NumPy
Tipos de elementos	Mixtos (`[1, "texto", True]`)	Todos del mismo tipo
Operaciones matemáticas	Requieren un loop	Directas sobre todo el array
Velocidad	Lenta para cálculos	10-100× más rápida
Uso típico	Colecciones generales, texto	Series numéricas, matrices

```
# Lista — la operación no funciona como se espera
velocidades_lista = [0.08, 0.09, 0.6, 0.07]
velocidades_lista * 1.944    # repite la lista, no multiplica cada elemento

# Array — la operación se aplica a cada elemento
velocidades = np.array([0.08, 0.09, 0.6, 0.07])
velocidades * 1.944          # [0.156, 0.175, 1.166, 0.136] — correcto
```

Regla práctica: si vas a hacer cálculos (sumas, medias, trigonometría), usa arrays. Si solo necesitas guardar una colección de cosas para iterar sobre ellas, una lista está bien.

9.2 Arrays

```
# Desde una lista
velocidades = np.array([0.08, 0.09, 0.6, 0.07, 0.08, 0.09, 0.10])

# Array de ceros o unos
np.zeros(12)           # [0. 0. 0. ... 0.]
np.ones((3, 4))        # matriz 3×4 de unos

# Secuencias
np.arange(0, 24, 2)    # [0, 2, 4, ..., 22]
np.linspace(0, 1, 50)  # 50 puntos equiespaciados entre 0 y 1
```

9.2.1 dtype — el tipo de los elementos

Cada array tiene un `dtype` que determina qué tipo de número almacena y cuántos bits usa:

```
vel = np.array([0.08, 0.09, 0.6, 0.07])
vel.dtype    # float64 — el default para números decimales
```

dtype	Bits	Precisión	Uso típico
`float64`	64	~15 dígitos	Default de NumPy, máxima precisión
`float32`	32	~7 dígitos	Archivos NetCDF, ahorra memoria
`int32`	32	enteros $\pm 2.1 \times 10^9$	Índices, contadores
`int64`	64	enteros grandes	Timestamps en nanosegundos
`uint8`	8	enteros 0–255	Imágenes (píxeles)

Por qué importa: los archivos NetCDF y muchos instrumentos oceanográficos almacenan datos en `float32` para ahorrar espacio. Cuando los lees con xarray o NumPy, los datos ya vienen como `float32`. Si mezclas `float32` con `float64` en una operación, NumPy promueve todo a `float64` — lo cual está bien, pero puede sorprender.

```
# Verificar
vel.dtype                 # float64

# Especificar al crear
vel32 = np.array([0.08, 0.09, 0.6], dtype=np.float32)
vel32.dtype               # float32

# Convertir
vel32 = vel.astype(np.float32)   # float64 → float32
vel64 = vel32.astype(np.float64) # float32 → float64
```

```
# Leer desde NetCDF (xarray) — frecuentemente viene como float32
import xarray as xr
ds = xr.open_dataset('corrientes.nc')
ds['velocidad'].dtype    # float32

# Convertir si necesitas precisión para análisis espectral
vel64 = ds['velocidad'].values.astype(np.float64)
```

[!] float32 en análisis espectral — La pérdida de precisión de float32 (~7 dígitos) rara vez importa en estadísticas descriptivas. Sí puede importar en análisis espectral o cuando se hacen muchas operaciones encadenadas sobre los mismos datos. En esos casos conviene convertir a float64 antes de calcular.

9.2.2 Arrays 2D — matrices

En corrientes se trabaja frecuentemente con matrices de `(tiempo × profundidad)`:

```
# Matriz de velocidades: 5 ensembles × 11 profundidades
datos = np.array([
    [0.08, 0.09, 0.6,  0.07, 0.08, 0.09, 0.10, 0.09, 0.08, 0.07, 0.06],
    [0.07, 0.08, 0.55, 0.06, 0.07, 0.08, 0.09, 0.08, 0.07, 0.06, 0.05],
    # ...
])

datos.shape    # (5, 11) — filas × columnas
datos.ndim     # 2
datos.size     # 55 — total de elementos
```

9.3 Operaciones vectorizadas y broadcasting

La ventaja de NumPy es que las operaciones se aplican a todo el array sin necesidad de un loop. A esto se le llama **vectorización**:

```
vel = np.array([0.08, 0.09, 0.6, 0.07])

# Operaciones elemento a elemento
vel * 1.944        # convertir m/s a nudos
vel ** 2           # cuadrado de cada elemento
np.sqrt(vel)       # raíz cuadrada
np.log(vel)        # logaritmo natural

# Comparación — devuelve array de booleanos
vel > 0.5          # [False, False, True, False]
```

Broadcasting es la regla que permite operar un array con un escalar (o con arrays de distinta forma compatible). NumPy ”expande” el escalar para que coincida con el tamaño del array:

```
vel = np.array([0.08, 0.09, 0.6, 0.07])

# En vez de hacer un loop para convertir cada elemento:
# for i in range(len(vel)):
#     vel[i] = vel[i] * 1.944

# NumPy lo hace solo — aplica el *1.944 a cada elemento:
vel * 1.944    # [0.156, 0.175, 1.166, 0.136]

# Sumar dos arrays del mismo tamaño — suma elemento a elemento:
u = np.array([0.1, 0.2, 0.3])
v = np.array([0.4, 0.5, 0.6])
magnitud = np.sqrt(u**2 + v**2)   # calcula raíz de (u²+v²) para cada par
```

9.3.1 Conversión velocidad/dirección ↔ componentes U, V

Esta operación es fundamental en oceanografía y aparece en múltiples scripts del pipeline:

```
def uv_desde_vel_dir(velocidad, direccion_grad):
    """Convierte (velocidad, dirección) a componentes (U, V)."""
    dir_rad = np.radians(direccion_grad)
    u = velocidad * np.sin(dir_rad)   # componente Este
    v = velocidad * np.cos(dir_rad)   # componente Norte
    return u, v

def vel_dir_desde_uv(u, v):
    """Convierte (U, V) a (velocidad, dirección)."""
    velocidad  = np.sqrt(u**2 + v**2)
    direccion  = np.degrees(np.arctan2(u, v)) % 360
    return velocidad, direccion
```

9.4 Indexación y slicing

```
prof = np.array([3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23])

prof[0]        # 3  — primer elemento
prof[-1]       # 23 — último
prof[2:5]      # [7, 9, 11]
prof[::2]      # [3, 7, 11, 15, 19, 23] — cada dos

# En 2D: [fila, columna]
datos[0, :]    # primera fila completa (ensemble 0, todas las profundidades)
datos[:, 2]    # columna 2 completa (todos los ensembles, profundidad 7 m)
datos[1:3, 3:6]  # submatriz
```

9.4.1 Indexación booleana — filtrado

```
vel = np.array([0.08, 0.09, 0.6, 0.07, 0.55, 0.09])

# Seleccionar solo los valores > 0.5
vel[vel > 0.5]         # [0.6, 0.55]

# Reemplazar valores fuera de rango con NaN
vel[vel > 0.5] = np.nan
```

9.5 Estadísticas

```
vel = np.array([0.08, 0.09, 0.60, 0.07, 0.08])

np.mean(vel)        # media
np.median(vel)      # mediana
np.std(vel)         # desviación estándar
np.max(vel)         # máximo
np.min(vel)         # mínimo
np.percentile(vel, 95)  # percentil 95

np.argmax(vel)      # índice del máximo → 2
```

9.5.1 Funciones que ignoran NaN

Cuando los datos tienen valores faltantes (`NaN`), usar las versiones `nan*`:

```
np.nanmean(vel)
np.nanmax(vel)
np.nanstd(vel)
np.nanpercentile(vel, 95)
```

9.5.2 Estadísticas por eje en matrices

```
datos.mean(axis=0)   # media por profundidad (a lo largo del tiempo)
datos.mean(axis=1)   # media por ensemble (a lo largo de profundidades)
datos.max(axis=0)    # máximo por profundidad
```

9.6 Funciones trigonométricas

NumPy trabaja en radianes. Para datos de dirección oceánica (en grados):

```
np.radians(180)      # π
np.degrees(np.pi)    # 180.0

np.sin(np.radians(90))   # 1.0
np.cos(np.radians(0))    # 1.0

# arctan2 — dirección del vector (U, V)
u, v = 0.5, 0.5
direccion = np.degrees(np.arctan2(u, v)) % 360   # 45.0°
```

9.7 NaN — valores faltantes

```
np.nan                      # valor especial "Not a Number"
np.isnan(vel)               # array booleano: True donde hay NaN
np.isnan(vel).sum()         # cantidad de NaN
~np.isnan(vel)              # máscara de datos válidos

# Contar datos válidos
datos_validos = vel[~np.isnan(vel)]
```

9.8 Operaciones útiles

```
# Diferencia entre elementos consecutivos
np.diff(np.array([1, 3, 6, 10]))   # [2, 3, 4]

# Concatenar arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
np.concatenate([a, b])             # [1, 2, 3, 4, 5, 6]

# Apilar matrices verticalmente
np.vstack([datos[:2, :], datos[3:, :]])

# Ordenar
np.sort(vel)                       # copia ordenada
vel.argsort()                      # índices que ordenarían el array

# Redondear
np.round(3.14159, 2)               # 3.14
```

[>] NumPy vs listas de Python — Para operaciones matemáticas sobre grandes conjuntos de datos, NumPy es entre 10 y 100 veces más rápido que un loop sobre una lista Python. En series temporales de 6 meses con datos cada 10 minutos (~26.000 registros), esa diferencia es significativa.

10 Pandas

Pandas es la librería principal para manejo de datos tabulares. Su estructura central es el **DataFrame**: una tabla con filas y columnas etiquetadas, similar a una hoja de Excel pero operable desde código.

En el procesamiento de datos oceanográficos, los datos de corrientes, viento y oleaje se manejan como DataFrames de Pandas.

```
import pandas as pd
```

10.1 Series y DataFrames

Una **Series** es una columna con índice. Un **DataFrame** es una tabla de Series con el mismo índice.

```
# Series
velocidad = pd.Series([0.08, 0.09, 0.6, 0.07], name='velocidad')

# DataFrame desde diccionario
df = pd.DataFrame({
    'velocidad': [0.08, 0.09, 0.60, 0.07],
    'direccion': [  45,   90,  352,  180],
    'profundidad': [  3,    3,    7,    3],
})
```

10.2 Explorar un DataFrame

```
df.head()           # primeras 5 filas
df.tail(10)         # últimas 10 filas
df.shape            # (filas, columnas)
df.columns          # nombres de columnas
df.dtypes           # tipo de cada columna
df.describe()       # estadísticas básicas
df.info()           # resumen general: tipos, NaN, memoria
```

10.3 Acceder a datos

```
# Columna por nombre
df['velocidad']
df[['velocidad', 'direccion']]   # varias columnas
```

10.3.1 loc vs iloc — etiqueta vs posición

Esta es la distinción que más confunde al principio. La diferencia es simple:

- **`iloc`** — accede por **posición numérica**, como los índices de una lista (0, 1, 2…)

- **`loc`** — accede por **etiqueta del índice**, que puede ser un número, una fecha, un string

```
# Si el índice del DataFrame es 0, 1, 2... ambos parecen iguales
df.iloc[0]      # primera fila (por posición)
df.loc[0]       # fila con etiqueta 0 (por etiqueta)

# La diferencia importa cuando el índice es una fecha
df = df.set_index('tiempo')   # índice es ahora DatetimeIndex

df.iloc[0]                          # primera fila del DataFrame
df.loc['2025-10-01']                # fila con esa fecha exacta
df.loc['2025-10-01':'2025-10-31']   # rango de fechas — solo funciona con loc
```

Regla práctica: si trabajas con series temporales (índice de fechas), usa `loc`. Si solo necesitas "las primeras N filas" o "la fila en la posición X", usa `iloc`.

```
# Combinando filas y columnas
df.iloc[0:5, 1:3]                       # filas 0-4, columnas 1-2 (posición)
df.loc['2025-10', ['velocidad', 'dir']]    # octubre, columnas por nombre
```

10.4 Filtrado

```
# Condición simple
df[df['velocidad'] > 0.5]

# Múltiples condiciones
df[(df['velocidad'] > 0.1) & (df['profundidad'] == 3)]
df[(df['direccion'] < 45) | (df['direccion'] > 315)]

# isin — valores en una lista
df[df['profundidad'].isin([3, 7, 15])]

# notna / isna
df[df['velocidad'].notna()]      # excluir NaN
df[df['velocidad'].isna()]       # solo NaN
```

10.5 Series temporales

En oceanografía, el índice del DataFrame suele ser un `DatetimeIndex`. Esto permite filtrar y agrupar por tiempo de forma muy eficiente.

```python
# Crear índice temporal desde una columna
df['tiempo'] = pd.to_datetime(df['tiempo'])
df = df.set_index('tiempo')

# Filtrar por rango de fechas
df['2025-10':'2026-03']
df.loc['2025-10-01':'2026-03-30']

# Resamplear: promedios horarios, diarios, mensuales
df.resample('1h').mean()    # promedio cada hora
df.resample('1D').max()     # máximo diario
df.resample('1ME').mean()   # promedio mensual
```

10.5.1 Timezone

Los datos del ADCP y la boya están en UTC. Para convertir a hora local (UTC-3):

```python
from datetime import timezone, timedelta

df.index = df.index.tz_localize('UTC')
df.index = df.index.tz_convert('America/Santiago')
```

10.6 Operaciones por columna

```python
# Crear nuevas columnas
df['vel_nudos'] = df['velocidad'] * 1.944

# Operaciones sobre columnas existentes
df['u'] = df['velocidad'] * np.sin(np.radians(df['direccion']))
df['v'] = df['velocidad'] * np.cos(np.radians(df['direccion']))

# Reemplazar valores
df['velocidad'] = df['velocidad'].replace(-9999, np.nan)

# Aplicar función personalizada
def clasificar_beaufort(vel):
    if vel < 0.3:   return "calma"
    elif vel < 1.5: return "ventolina"
    elif vel < 3.3: return "brisa leve"
    else:           return "brisa moderada o más"

df['beaufort'] = df['velocidad'].apply(clasificar_beaufort)
```

10.7 Estadísticas

```
df['velocidad'].mean()
df['velocidad'].max()
df['velocidad'].std()
df['velocidad'].quantile(0.95)   # percentil 95

# Por columna
df.mean()
df.describe()

# Contar valores no nulos
df['velocidad'].count()
df['velocidad'].isna().sum()    # cantidad de NaN
```

10.8 groupby — estadísticas por categoría

`groupby` divide el DataFrame en grupos según el valor de una columna, calcula algo dentro de cada grupo, y devuelve los resultados combinados. Es el equivalente en código de "agrupar por X en una tabla pivot de Excel".

```
DataFrame original          Después de groupby('profundidad')
──────────────────          ──────────────────────────────────

profundidad  velocidad       profundidad  velocidad_media
3            0.08            3            0.085
3            0.09            7            0.335
7            0.60            ← promedio dentro de cada grupo
7            0.07
```

```
# Estadísticas por profundidad
df.groupby('profundidad')['velocidad'].mean()
df.groupby('profundidad')['velocidad'].agg(['mean', 'max', 'std'])

# Estadísticas por mes
df.groupby(df.index.month)['velocidad'].mean()

# Por hora del día (patrón diurno)
df.groupby(df.index.hour)['velocidad'].mean()
```

10.8.1 Ejemplo real: patrón diurno del viento

```
patron_diurno = df.groupby(df.index.hour)['velocidad'].mean()
patron_diurno.index.name = 'hora'
```

10.8.2 groupby con múltiples agregaciones

Cuando se necesita calcular varias estadísticas a la vez, `agg` acepta un diccionario con nombre de salida y función:

```
resumen = df.groupby('profundidad')['velocidad'].agg(
    media='mean',
    maxima='max',
    p95=lambda x: x.quantile(0.95),
    n_datos='count'
)
```

Esto devuelve un DataFrame con una columna por estadística, nombrada explícitamente — más claro que encadenar varias llamadas separadas.

```
# Agrupar por dos columnas a la vez
df.groupby(['profundidad', df.index.month])['velocidad'].mean()
# → media por cada combinación (profundidad, mes)
```

10.9 Rolling — ventana deslizante

`rolling` aplica una función sobre una ventana móvil de N filas. Es la forma estándar de suavizar series temporales y calcular estadísticas en un intervalo de tiempo deslizante.

```
# Suavizado: media móvil de 1 hora (datos cada 10 min → ventana de 6 puntos)
df['vel_suavizada'] = df['velocidad'].rolling(window=6).mean()

# La ventana es centrada por defecto hacia atrás:
# el valor en t es la media de [t-5, t-4, t-3, t-2, t-1, t]
# Las primeras N-1 filas serán NaN porque no hay suficientes datos previos
```

Con series temporales de frecuencia regular, es más claro usar el tamaño de ventana en tiempo:

```
df['vel_suavizada'] = df['velocidad'].rolling('1h').mean()    # media de 1 hora
df['vel_suavizada'] = df['velocidad'].rolling('6h').mean()    # media de 6 horas
df['std_movil']     = df['velocidad'].rolling('1D').std()     # std diaria móvil
```

```
# Percentil 95 móvil — más pesado pero válido
df['p95_movil'] = df['velocidad'].rolling('7D').quantile(0.95)
```

[>] Cuándo suavizar — Los datos de ADCP tienen ruido acústico y efectos de ondas superficiales. Una media móvil de 10–60 minutos elimina variabilidad de alta frecuencia sin afectar la señal de corriente. No suavizar antes de hacer estadísticas mensuales — eso puede sesgar los resultados.

10.10 pd.cut — crear intervalos

`pd.cut` divide una columna continua en intervalos (bins) y asigna una etiqueta a cada valor. Se usa para construir tablas de incidencia (velocidad × dirección) y estadísticas por rango de profundidad.

```
# Crear intervalos de velocidad: 0-0.1, 0.1-0.25, 0.25-0.5, 0.5-1.0 m/s
bins   = [0, 0.1, 0.25, 0.5, 1.0]
labels = ['calma', 'leve', 'moderada', 'fuerte']

df['intervalo_vel'] = pd.cut(df['velocidad'], bins=bins, labels=labels)
```

```
# Número de intervalos iguales — pandas elige los límites automáticamente
df['quintil_vel'] = pd.cut(df['velocidad'], bins=5)

# pd.qcut — misma cantidad de datos en cada intervalo (cuantiles)
df['cuartil_vel'] = pd.qcut(df['velocidad'], q=4, labels=['Q1','Q2','Q3','Q4'])
```

Una vez que cada dato tiene su etiqueta de intervalo, se puede agrupar:

```
df.groupby('intervalo_vel')['velocidad'].count()   # frecuencia por intervalo
df.groupby('intervalo_vel')['velocidad'].mean()    # media dentro de cada rango
```

10.11 pivot_table — tabla de frecuencias

Las tablas de incidencia (velocidad × dirección) del informe se generan con pivot_table:

```
# Tabla de frecuencia: velocidad × dirección
tabla = pd.pivot_table(
    df,
    values='velocidad',
    index='intervalo_vel',
    columns='octante',
    aggfunc='count',
    fill_value=0
)

# Normalizar a porcentaje
tabla_pct = tabla / tabla.values.sum() * 100
```

10.12 merge y concat — combinar DataFrames

10.12.1 concat — apilar filas

```
# Misma estructura, distintos períodos — apilar verticalmente
df_total = pd.concat([df_septiembre, df_octubre, df_noviembre])

# Si los índices originales se repiten, resetear
df_total = pd.concat([df_sep, df_oct, df_nov], ignore_index=True)

# Para saber de qué DataFrame vino cada fila
df_total = pd.concat(
```

```
    [df_sep, df_oct, df_nov],
    keys=['sep', 'oct', 'nov']
)
```

10.12.2 merge — combinar por columna común

`merge` une dos DataFrames que comparten una columna clave. El parámetro `how` controla qué filas se conservan cuando no hay match:

```
# inner (default): solo las filas que existen en ambos
df = pd.merge(df_corrientes, df_oleaje, on='tiempo', how='inner')

# left: todas las filas de df_corrientes, NaN donde no hay oleaje
df = pd.merge(df_corrientes, df_oleaje, on='tiempo', how='left')

# outer: todas las filas de ambos, NaN donde falta alguno
df = pd.merge(df_corrientes, df_oleaje, on='tiempo', how='outer')
```

```
df_corrientes    df_oleaje        inner merge          left merge
──────────────   ──────────       ──────────────       ──────────
tiempo  vel      tiempo  Hm0      tiempo  vel  Hm0     tiempo  vel  Hm0
10:00   0.3      10:00   1.2      10:00   0.3  1.2     10:00   0.3  1.2
10:10   0.4      10:20   0.8      10:20   0.5  0.8     10:10   0.4  NaN
10:20   0.5                                           10:20   0.5  0.8
```

```
# Si las columnas clave tienen distinto nombre
df = pd.merge(df_corrientes, df_viento,
              left_on='tiempo_adcp', right_on='tiempo_boya',
              how='left')
```

Regla práctica: usar `left` cuando el DataFrame izquierdo es el principal y el derecho agrega información complementaria que puede no estar para todos los instantes. Usar `inner` cuando solo interesan los instantes donde hay datos de ambos instrumentos.

10.13 Manejar NaN

```
df.dropna()                          # eliminar filas con cualquier NaN
df.dropna(subset=['velocidad'])      # solo si velocidad es NaN
df.fillna(0)                         # rellenar con 0
df['velocidad'].interpolate()        # interpolación lineal
df.ffill()                           # propagar último valor válido hacia adelante
```

10.14 Leer y guardar datos

```
# Leer
df = pd.read_csv('corrientes.csv', sep=';', parse_dates=['tiempo'])
df = pd.read_excel('corrientes.xlsx', sheet_name='VELOCIDAD')

# Guardar
df.to_csv('resultado.csv', index=False)
df.to_excel('resultado.xlsx', sheet_name='Datos', index=False)
```

[>] Variable Explorer en Spyder — Hacer doble clic en un DataFrame en el Variable Explorer de Spyder abre una vista de tabla interactiva donde se pueden ordenar columnas y buscar valores, sin escribir ningún código adicional.

11 Lectura de archivos

Un pipeline de datos científicos lee de múltiples fuentes y formatos: CSV de dataloggers, Excel de procesamiento, archivos `.mat` de MATLAB, y documentos Word para extracción de metadata.

11.1 Archivos de texto y CSV

```
import pandas as pd

# CSV estándar
df = pd.read_csv('viento.csv')

# Opciones comunes
df = pd.read_csv(
    'viento.csv',
    sep=';',                        # separador de columna
    decimal=',',                    # decimal con coma (datos europeos)
    skiprows=2,                     # saltar filas de encabezado
    encoding='latin-1',             # encoding para tildes
    parse_dates=['fecha'],          # convertir columna a datetime
    index_col='fecha',              # usar fecha como índice
    na_values=['-9999', 'N/A', '']  # valores a tratar como NaN
)
```

11.1.1 Detectar encoding

Si el archivo tiene tildes y aparecen caracteres extraños, probar:

```
for enc in ['utf-8', 'latin-1', 'cp1252']:
    try:
        df = pd.read_csv('archivo.csv', encoding=enc)
        print(f"Funciona con: {enc}")
        break
    except UnicodeDecodeError:
        continue
```

11.2 Archivos Excel

```
# Hoja por nombre
df = pd.read_excel('corrientes.xlsx', sheet_name='VELOCIDAD')

# Primera hoja
df = pd.read_excel('corrientes.xlsx', sheet_name=0)

# Leer todas las hojas
```

```
hojas = pd.read_excel('corrientes.xlsx', sheet_name=None)
# hojas es un dict: {'VELOCIDAD': df1, 'DIRECCION': df2, ...}

for nombre, df in hojas.items():
    print(f"Hoja '{nombre}': {df.shape}")
```

11.2.1 Leer varias hojas de un Excel con múltiples pestañas

Cuando el Excel tiene hojas con nombres conocidos, conviene leerlas en un diccionario y manejar el caso de que alguna falte:

```
hojas_requeridas = ['Datos', 'Estadisticas', 'Resumen']

datos = {}
for hoja in hojas_requeridas:
    try:
        datos[hoja] = pd.read_excel('procesamiento.xlsx', sheet_name=hoja)
    except Exception as e:
        print(f"  ! No se encontró la hoja '{hoja}': {e}")
```

11.3 Archivos MATLAB (.mat)

Algunos instrumentos oceanográficos (sensores de oleaje, perfiladores) exportan datos en formato `.mat`. Se leen con `scipy.io`:

```
from scipy.io import loadmat

mat = loadmat('datos_storm2.mat')

# mat es un diccionario con las variables del workspace de MATLAB
print(mat.keys())

# Extraer variables
Hm0 = mat['Hm0'].flatten()      # flatten convierte (N,1) → (N,)
Tp  = mat['Tp'].flatten()
tiempo = mat['time'].flatten()
```

11.3.1 Convertir tiempo MATLAB a datetime de Python

MATLAB guarda el tiempo como número de días desde el 0-ene-0000. Para convertir a pandas:

```
import pandas as pd
import numpy as np

def matlab_datenum_a_datetime(datenum):
    return pd.to_datetime(datenum - 719529, unit='D', origin='unix')

timestamps = matlab_datenum_a_datetime(tiempo)
```

11.4 Archivos YAML

YAML es un formato de configuración más legible que JSON: no usa comillas en los strings simples, admite comentarios con `#`, y su indentación lo hace más natural para estructuras anidadas. Es el estándar para archivos de configuración de proyectos y pipelines.

```
conda install pyyaml    # o: pip install pyyaml
```

```
import yaml

# Leer
with open('config.yaml', 'r', encoding='utf-8') as f:
    config = yaml.safe_load(f)    # safe_load evita ejecución de código arbitrario

empresa = config['empresa']
ruta    = config['rutas']['datos']

# Guardar
with open('config.yaml', 'w', encoding='utf-8') as f:
    yaml.dump(config, f, allow_unicode=True, default_flow_style=False)
```

11.4.1 config.yaml para un proyecto oceanográfico

```
# Configuración del proyecto — editar aquí, no en el código
proyecto:
  empresa:  "Compas Marine"
  centro:   "Los Vilos"
  campana:  "oct2025"

rutas:
  datos:    "/mnt/c/Users/Usuario/proyectos/los_vilos/datos"
  figuras:  "/mnt/c/Users/Usuario/proyectos/los_vilos/figuras"
  informe:  "/mnt/c/Users/Usuario/proyectos/los_vilos/informe"

instrumentos:
  adcp:
    archivo:       "corrientes_adcp.csv"
```

```
    profundidades: [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]
  viento:
    archivo:       "viento_boya.csv"

parametros:
  velocidad_max_corte: 1.5    # m/s — valores sobre este umbral son outliers
  suavizado_horas:     1      # ventana de rolling en horas
```

```
# Usar en el pipeline
import yaml
from pathlib import Path

with open('config.yaml', 'r', encoding='utf-8') as f:
    cfg = yaml.safe_load(f)

ruta_datos  = Path(cfg['rutas']['datos'])
prof_adcp   = cfg['instrumentos']['adcp']['profundidades']
vel_max     = cfg['parametros']['velocidad_max_corte']

df = pd.read_csv(ruta_datos / cfg['instrumentos']['adcp']['archivo'])
df = df[df['velocidad'] < vel_max]
```

La ventaja sobre rutas hardcodeadas: el mismo script corre para cualquier proyecto cambiando solo el `config.yaml`. No hay que tocar el código.

11.5 Archivos JSON

```
import json

# Leer
with open('config.json', 'r', encoding='utf-8') as f:
    config = json.load(f)

empresa = config['empresa']

# Guardar
with open('estado.json', 'w', encoding='utf-8') as f:
    json.dump(config, f, ensure_ascii=False, indent=2)
```

11.6 Archivos de texto plano

```python
# Leer completo
with open('notas.txt', 'r', encoding='utf-8') as f:
    contenido = f.read()

# Leer línea por línea
with open('log.txt', 'r') as f:
    for linea in f:
        print(linea.strip())

# Escribir
with open('resultado.txt', 'w', encoding='utf-8') as f:
    f.write("Velocidad máxima: 0.6 m/s\n")
    f.write("Profundidad: 7 m\n")
```

11.7 Buscar archivos con glob y os

En el pipeline es frecuente buscar archivos por patrón sin saber el nombre exacto:

```python
import glob
import os

# Todos los Excel en una carpeta
archivos = glob.glob('/ruta/carpeta/*.xlsx')

# Recursivo (busca en subcarpetas también)
archivos = glob.glob('/ruta/**/*.xlsx', recursive=True)

# Filtrar con condición
excels_corrientes = [
    f for f in glob.glob('/ruta/*.xlsx')
    if 'corriente' in f.lower()
]

# Listar archivos y carpetas
os.listdir('/ruta/carpeta')

# Verificar si existe
os.path.exists('/ruta/archivo.csv')

# Construir rutas de forma segura (funciona en Windows y Linux)
ruta = os.path.join('/ruta/base', 'subcarpeta', 'archivo.csv')
```

11.8 Documentos Word (.docx)

Python-docx permite leer texto desde documentos Word, útil para extraer metadata de informes anteriores:

```
from docx import Document

doc = Document('informe.docx')

# Leer todos los párrafos
for parrafo in doc.paragraphs:
    if parrafo.text.strip():
        print(parrafo.text)

# Leer tablas
for tabla in doc.tables:
    for fila in tabla.rows:
        celdas = [celda.text for celda in fila.cells]
        print(celdas)
```

11.8.1 Leer .docx sin python-docx (XML directo)

Para extraer texto con más control, el `.docx` es en realidad un archivo ZIP con XML adentro:

```
import zipfile
import re

def extraer_texto_docx(ruta):
    with zipfile.ZipFile(ruta) as z:
        xml = z.read('word/document.xml').decode('utf-8')
    # Eliminar etiquetas XML
    texto = re.sub(r'<[^>]+>', ' ', xml)
    texto = re.sub(r'\s+', ' ', texto).strip()
    return texto
```

11.9 Manejo de rutas con pathlib

La librería `pathlib` ofrece una forma más moderna de trabajar con rutas:

```
from pathlib import Path

carpeta = Path('/ruta/a/mis/datos')
archivo = carpeta / 'campana_oct2025' / 'corrientes.csv'

archivo.exists()        # True/False
archivo.suffix          # '.csv'
archivo.stem            # 'corrientes'
archivo.parent          # carpeta Los Vilos
archivo.name            # 'corrientes.csv'

# Listar archivos
```

```
list(carpeta.glob('*.xlsx'))
list(carpeta.rglob('*.csv'))   # recursivo
```

[!] Rutas en Windows desde WSL — Al usar Python desde WSL (Windows Subsystem for Linux), las rutas de Windows se acceden como /mnt/c/…. Las rutas con espacios deben ir entre comillas o manejarse con Path: python ruta = Path(‘/mnt/c/Users/Usuario/Mi Carpeta/archivo.csv’)

12 Matplotlib

Matplotlib es la librería de visualización estándar en Python. Permite crear desde gráficos simples hasta figuras multipanel complejas listas para publicación.

```
import matplotlib.pyplot as plt
import numpy as np
```

12.1 Gráfico básico

```
tiempo = np.arange(0, 186)
velocidad = np.random.normal(3.93, 2.8, 186)

plt.figure(figsize=(12, 4))
plt.plot(tiempo, velocidad, color='steelblue', linewidth=0.8)
plt.xlabel('Día')
plt.ylabel('Velocidad (m/s)')
plt.title('Serie temporal de velocidad del viento')
plt.tight_layout()
plt.savefig('serie_viento.png', dpi=150)
plt.show()
```

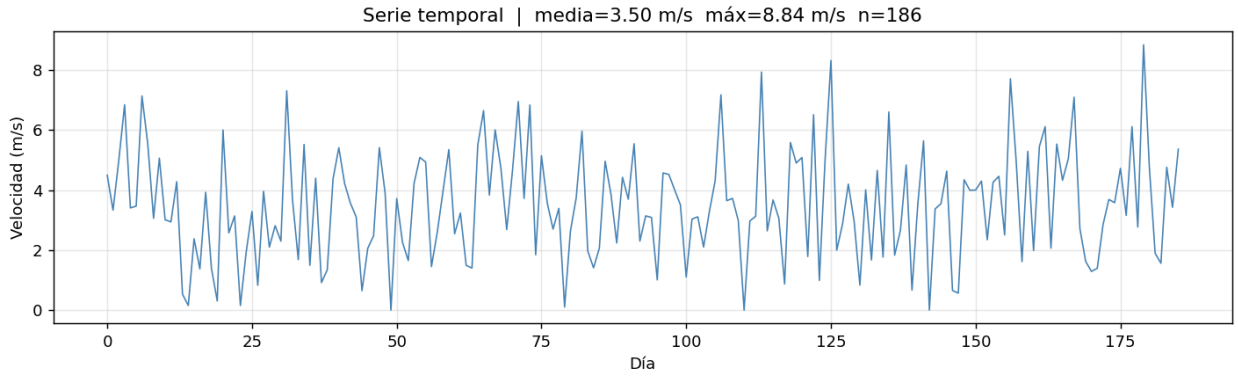


Figura 1: Serie temporal con título calculado

12.2 Figure y Axes — la estructura correcta

Para figuras con múltiples paneles o más control, se trabaja directamente con objetos `Figure` y `Axes`:

```
fig, ax = plt.subplots(figsize=(12, 4))

ax.plot(tiempo, velocidad, color='steelblue', linewidth=0.8)
ax.set_xlabel('Día')
ax.set_ylabel('Velocidad (m/s)')
ax.set_title('Serie temporal')
ax.grid(True, alpha=0.3)

fig.tight_layout()
fig.savefig('serie_viento.png', dpi=150)
```

12.3 Múltiples paneles

12.3.1 subplot simple

```
fig, axes = plt.subplots(3, 1, figsize=(12, 10), sharex=True)

axes[0].plot(tiempo, Hm0, color='navy')
axes[0].set_ylabel('Hm0 (m)')

axes[1].plot(tiempo, Tm, color='teal')
axes[1].set_ylabel('Tm (s)')

axes[2].plot(tiempo, Dm, color='darkorange', marker='.', markersize=1,
↪  linestyle='none')
axes[2].set_ylabel('Dm (°)')
axes[2].set_xlabel('Tiempo')

fig.tight_layout()
```

12.3.2 subplot2grid — paneles de distinto tamaño

```
fig = plt.figure(figsize=(14, 8))

ax1 = plt.subplot2grid((2, 3), (0, 0), colspan=2)   # fila 0, cols 0-1
ax2 = plt.subplot2grid((2, 3), (0, 2))               # fila 0, col 2
ax3 = plt.subplot2grid((2, 3), (1, 0), colspan=3)   # fila 1, ancho completo
```

12.4 Tipos de gráfico

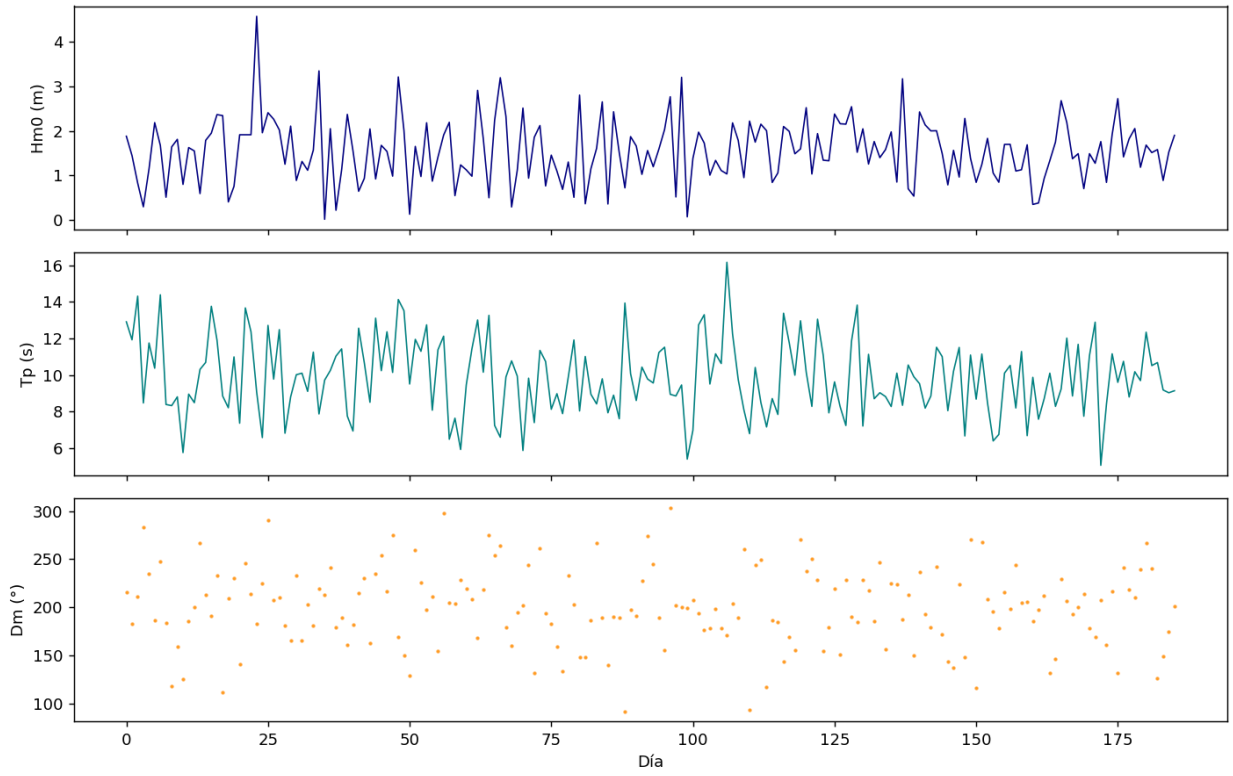


Figura 2: Tres paneles con sharex

```
# Línea
ax.plot(x, y, color='steelblue', linewidth=1, linestyle='--', label='velocidad')

# Puntos
ax.scatter(x, y, c=colores, s=20, alpha=0.5, cmap='viridis')

# Barras
ax.bar(meses, promedios, color='teal', edgecolor='white')

# Histograma
ax.hist(velocidad, bins=20, color='steelblue', edgecolor='white', density=True)

# Área rellena
ax.fill_between(tiempo, y_min, y_max, alpha=0.3, color='steelblue')

# Línea horizontal / vertical de referencia
ax.axhline(y=6, color='red', linestyle='--', linewidth=0.8, label='umbral')
ax.axvline(x=45, color='gray', linestyle=':')
```

12.5 Colores y estilos

```
# Colores nombrados
'steelblue', 'navy', 'teal', 'darkorange', 'firebrick', 'gray'

# Hex
'#2196F3'

# Escala de grises
'0.5'   # gris medio

# Transparencia
ax.plot(x, y, color='steelblue', alpha=0.7)

# Colormaps — para datos continuos
cmap = plt.cm.viridis
cmap = plt.cm.RdBu_r   # divergente, útil para anomalías
```

12.6 Ejes y etiquetas

```
ax.set_xlim(0, 186)
ax.set_ylim(0, 20)

# Ticks personalizados
ax.set_xticks([0, 30, 60, 90, 120, 150, 180])
ax.set_xticklabels(['sep', 'oct', 'nov', 'dic', 'ene', 'feb', 'mar'])
ax.tick_params(axis='x', rotation=45)

# Formato de números en eje
from matplotlib.ticker import PercentFormatter
ax.yaxis.set_major_formatter(PercentFormatter(decimals=1))

# Escala logarítmica
ax.set_yscale('log')
```

12.7 Leyenda y anotaciones

```
ax.plot(x, y1, label='Velocidad media')
ax.plot(x, y2, label='Percentil 95')
ax.legend(loc='upper right', fontsize=9)

# Anotación de texto
ax.text(0.02, 0.95, f'Máximo: {vmax:.2f} m/s',
        transform=ax.transAxes,   # coordenadas relativas al axes (0-1)
        fontsize=9, verticalalignment='top')

# Flecha con texto
ax.annotate('Evento energético', xy=(45, 3.0), xytext=(60, 3.5),
            arrowprops=dict(arrowstyle='->', color='red'))
```

12.8 Mapas de calor (heatmap)

Muy usado en corrientes para visualizar velocidad por tiempo y profundidad:

```
# datos: matriz (n_tiempos × n_profundidades)
im = ax.pcolormesh(tiempos, profundidades, datos.T,
                   cmap='RdBu_r', vmin=-0.3, vmax=0.3)
fig.colorbar(im, ax=ax, label='Velocidad (m/s)')
ax.set_ylabel('Profundidad (m)')
ax.invert_yaxis()   # profundidad creciente hacia abajo
```

12.9 Guardar figuras

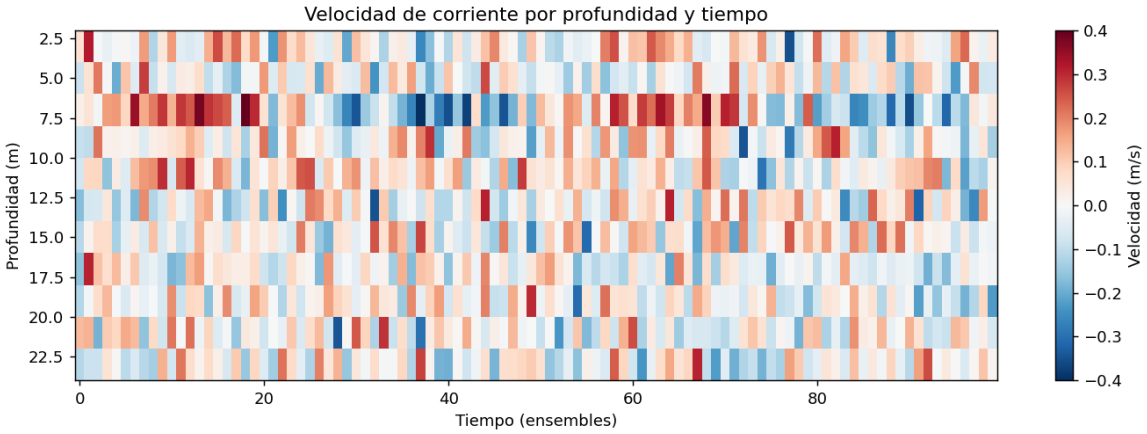


Figura 3: Heatmap de corrientes

```
fig.savefig('figura.png', dpi=150, bbox_inches='tight')
fig.savefig('figura.pdf', bbox_inches='tight')    # vectorial
fig.savefig('figura.svg', bbox_inches='tight')    # vectorial editable

plt.close(fig)   # liberar memoria — importante en scripts que generan muchas figuras
```

12.9.1 PDF multipágina

```
from matplotlib.backends.backend_pdf import PdfPages

with PdfPages('informe_figuras.pdf') as pdf:
    for mes in meses:
        fig, ax = plt.subplots()
        ax.plot(datos_mes[mes])
        ax.set_title(mes)
        pdf.savefig(fig)
        plt.close(fig)
```

12.10 Figuras con valores calculados

En análisis real, los títulos, etiquetas y leyendas deben mostrar valores del propio conjunto de datos — no texto fijo. Así la figura se documenta a sí misma.

12.10.1 Título y subtítulo con estadísticas

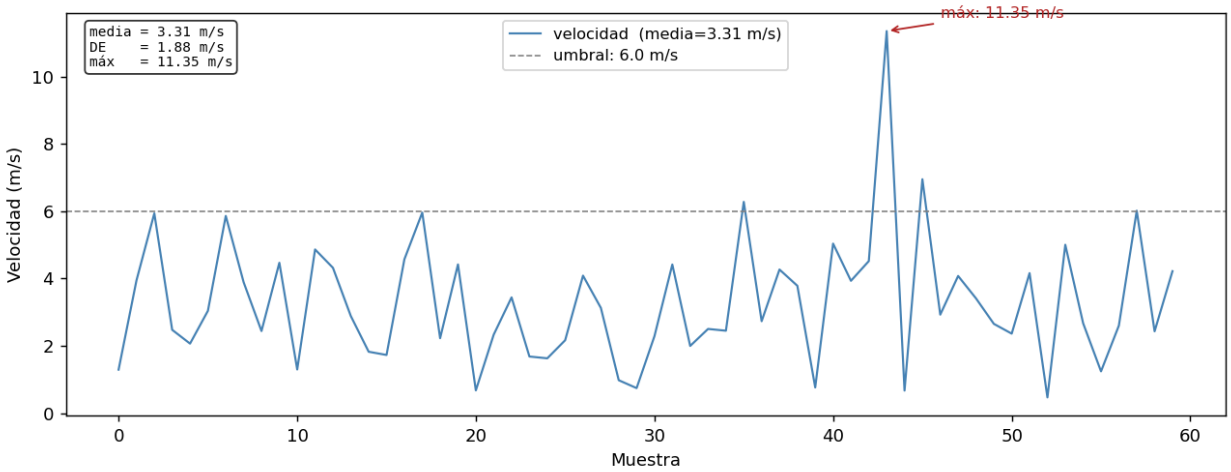


Figura 4: Anotaciones y leyenda con valores calculados

```
import matplotlib.pyplot as plt
import numpy as np

velocidad = np.array([1.2, 3.4, 2.1, 4.5, 0.8, 3.9, 2.7])

fig, ax = plt.subplots(figsize=(10, 4))
ax.plot(velocidad, color='steelblue', linewidth=1.2)

# Calcular estadísticas para usar en el título
v_media = velocidad.mean()
v_max   = velocidad.max()
n       = len(velocidad)

ax.set_title(f'Serie temporal de velocidad  |  media={v_media:.2f} m/s
↪  máx={v_max:.2f} m/s  n={n}')
ax.set_xlabel('Muestra')
ax.set_ylabel('Velocidad (m/s)')
fig.tight_layout()
```

12.10.2 Leyendas con valores calculados

```
profundidades = [5, 10, 20]
datos = {
    5:  np.random.normal(2.5, 0.8, 186),
    10: np.random.normal(1.8, 0.6, 186),
    20: np.random.normal(0.9, 0.4, 186),
}
```

```
fig, ax = plt.subplots(figsize=(12, 4))

for prof, serie in datos.items():
    media = serie.mean()
    # La etiqueta de la leyenda incluye el valor calculado
    ax.plot(serie, label=f'{prof} m  (media={media:.2f} m/s)', linewidth=0.8)

ax.set_xlabel('Tiempo (días)')
ax.set_ylabel('Velocidad (m/s)')
ax.legend(loc='upper right', fontsize=9)
fig.tight_layout()
```

12.10.3 Anotaciones sobre el gráfico

```
fig, ax = plt.subplots(figsize=(10, 4))
ax.plot(velocidad, color='steelblue', linewidth=1.2)

# Marcar el máximo con una anotación
idx_max = velocidad.argmax()
v_max   = velocidad[idx_max]

ax.annotate(
    f'máx: {v_max:.2f} m/s',
    xy=(idx_max, v_max),           # punto al que apunta la flecha
    xytext=(idx_max + 0.5, v_max + 0.3),  # posición del texto
    arrowprops=dict(arrowstyle='->', color='firebrick'),
    fontsize=9, color='firebrick'
)

# Línea de referencia con su propio label
umbral = 3.0
ax.axhline(umbral, color='gray', linestyle='--', linewidth=0.8,
           label=f'umbral operacional: {umbral} m/s')
ax.legend(fontsize=9)
fig.tight_layout()
```

12.10.4 Texto estadístico dentro del panel

```
fig, ax = plt.subplots(figsize=(10, 4))
ax.plot(velocidad, color='steelblue')

# Bloque de estadísticas en esquina del axes
stats_texto = (
    f"media = {velocidad.mean():.2f} m/s\n"
    f"DE    = {velocidad.std():.2f} m/s\n"
```

```
    f"máx   = {velocidad.max():.2f} m/s\n"
    f"n     = {len(velocidad)}"
)
ax.text(
    0.02, 0.97, stats_texto,
    transform=ax.transAxes,       # coordenadas 0-1 relativas al panel
    fontsize=8, verticalalignment='top',
    fontfamily='monospace',
    bbox=dict(boxstyle='round', facecolor='white', alpha=0.8)
)
fig.tight_layout()
```

12.10.5 Título desde metadatos del archivo

```
import pandas as pd

df = pd.read_csv('corrientes_oct2025.csv', parse_dates=['fecha'])

fecha_ini = df['fecha'].min().strftime('%d %b %Y')
fecha_fin = df['fecha'].max().strftime('%d %b %Y')
n_dias    = (df['fecha'].max() - df['fecha'].min()).days

fig, ax = plt.subplots(figsize=(12, 4))
ax.plot(df['fecha'], df['velocidad'], linewidth=0.8)
ax.set_title(f'Velocidad de corriente — {fecha_ini} al {fecha_fin} ({n_dias} días)')
ax.set_xlabel('Fecha')
ax.set_ylabel('Velocidad (m/s)')
fig.autofmt_xdate()   # rota etiquetas de fecha automáticamente
fig.tight_layout()
```

12.11 Backend gráfico

El backend controla cómo se muestran las figuras:

```
import matplotlib
matplotlib.use('Qt5Agg')   # ventana interactiva (ideal en Spyder)
# o
matplotlib.use('Agg')      # sin ventana, solo guardar a archivo (ideal en scripts
↪  automáticos)
```

En Spyder se configura desde **Preferences → IPython console → Graphics → Backend**.

[>] plt.close() en scripts automáticos — Cuando un script genera decenas de figuras (como el autoinforme), es importante cerrar cada figura después de guardarla con plt.close(fig). De lo contrario Matplotlib acumula todas en memoria y Spyder puede volverse lento o colapsar.

13 Rosas de viento y diagramas polares

Las rosas de dirección y los diagramas polares son las figuras más características del análisis oceanográfico. Se usan para visualizar la distribución de velocidades y direcciones del viento, corrientes y oleaje.

13.1 Rosa de viento con windrose

La librería `windrose` genera rosas directamente desde arrays de velocidad y dirección:

```
from windrose import WindroseAxes
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure(figsize=(7, 7))
ax = WindroseAxes.from_ax(fig=fig)

ax.bar(
    direccion,           # array de direcciones (°, convención meteorológica: FROM)
    velocidad,           # array de velocidades (m/s)
    normed=True,         # mostrar como porcentaje
    opening=0.9,
    edgecolor='white',
    bins=[0, 2, 4, 6, 8, 10],   # intervalos de velocidad
    cmap=plt.cm.YlOrRd
)

ax.set_legend(title='Velocidad (m/s)', loc='lower right')
ax.set_title('Rosa de viento — Los Vilos')

fig.savefig('rosa_viento.png', dpi=150, bbox_inches='tight')
plt.close(fig)
```

13.2 Rosa de corrientes (sin windrose)

Para corrientes se suele hacer una rosa manual usando proyección polar de Matplotlib, con más control sobre el diseño:

```
import matplotlib.pyplot as plt
import numpy as np

def rosa_corrientes(direcciones, velocidades, titulo='', ax=None):
    """Rosa de corrientes en 8 octantes."""
    octantes = np.arange(0, 360, 45)
    etiquetas = ['N', 'NE', 'E', 'SE', 'S', 'SO', 'O', 'NO']
    colores_vel = plt.cm.Blues(np.linspace(0.3, 1.0, 4))
    bins_vel = [0, 0.05, 0.1, 0.2, np.inf]
    labels_vel = ['0–0.05', '0.05–0.1', '0.1–0.2', '>0.2']
```

```
    if ax is None:
        fig, ax = plt.subplots(subplot_kw={'projection': 'polar'}, figsize=(6, 6))

    # Orientar norte arriba, sentido horario
    ax.set_theta_zero_location('N')
    ax.set_theta_direction(-1)

    ancho = np.radians(45) * 0.85
    bottom = np.zeros(8)

    for i, (vmin, vmax) in enumerate(zip(bins_vel[:-1], bins_vel[1:])):
        mask = (velocidades >= vmin) & (velocidades < vmax)
        conteos = np.zeros(8)
        for j, oct in enumerate(octantes):
            mask_dir = (direcciones >= oct - 22.5) & (direcciones < oct + 22.5)
            conteos[j] = np.sum(mask & mask_dir)

        pct = conteos / len(velocidades) * 100
        theta = np.radians(octantes)
        ax.bar(theta, pct, width=ancho, bottom=bottom,
               color=colores_vel[i], label=labels_vel[i], edgecolor='white',
↪   linewidth=0.5)
        bottom += pct

    ax.set_xticks(np.radians(octantes))
    ax.set_xticklabels(etiquetas)
    ax.set_title(titulo, pad=15)
    ax.legend(title='Vel (m/s)', loc='lower right', bbox_to_anchor=(1.3, -0.1))

    return ax
```

13.3 Diagrama polar de dispersión

Muestra la distribución conjunta de velocidad y dirección, con cada punto representando un registro:

```
fig, ax = plt.subplots(subplot_kw={'projection': 'polar'}, figsize=(7, 7))
ax.set_theta_zero_location('N')
ax.set_theta_direction(-1)

sc = ax.scatter(
    np.radians(direccion),
    velocidad,
    c=velocidad,           # color según velocidad
    cmap='YlOrRd',
    s=3,
```

```
    alpha=0.4
)

# Marcar el máximo
idx_max = np.argmax(velocidad)
ax.scatter(np.radians(direccion[idx_max]), velocidad[idx_max],
           c='red', s=80, zorder=5, label=f'Máx: {velocidad[idx_max]:.2f} m/s')

fig.colorbar(sc, ax=ax, label='Velocidad (m/s)', shrink=0.7)
ax.set_title('Diagrama polar de dispersión')
ax.legend(loc='lower right')
```

13.4 Vector progresivo

El vector progresivo acumula el desplazamiento (U, V) a lo largo del tiempo, mostrando la trayectoria resultante:

```
def vector_progresivo(velocidad, direccion, dt_horas=0.167):
    """
    Calcula el vector progresivo.
    dt_horas: intervalo entre registros en horas (10 min = 0.167 h)
    """
    u = velocidad * np.sin(np.radians(direccion))
    v = velocidad * np.cos(np.radians(direccion))

    # Desplazamiento acumulado en km
    x = np.cumsum(u * dt_horas * 3.6)   # m/s × h × 3.6 → km
    y = np.cumsum(v * dt_horas * 3.6)

    return x, y

# Graficar
x, y = vector_progresivo(vel_prof, dir_prof)

fig, ax = plt.subplots(figsize=(6, 6))
ax.plot(x, y, color='steelblue', linewidth=0.8)
ax.plot(0, 0, 'go', markersize=8, label='Inicio')
ax.plot(x[-1], y[-1], 'r*', markersize=12, label=f'Final:
↪  {np.sqrt(x[-1]**2+y[-1]**2):.1f} km')
ax.axhline(0, color='gray', linewidth=0.5)
ax.axvline(0, color='gray', linewidth=0.5)
ax.set_xlabel('Desplazamiento E–O (km)')
ax.set_ylabel('Desplazamiento N–S (km)')
ax.set_aspect('equal')
ax.legend()
ax.set_title('Vector progresivo')
```

13.5 Patrón diurno

Gráfico de líneas por mes mostrando el ciclo horario de la velocidad:

```
import pandas as pd

# df tiene índice DatetimeIndex
patron = df.groupby([df.index.month,
 ↪  df.index.hour])['velocidad'].mean().unstack(level=0)
# patron: filas=hora (0–23), columnas=mes (1–12)

nombres_meses = {9:'Sep', 10:'Oct', 11:'Nov', 12:'Dic', 1:'Ene', 2:'Feb', 3:'Mar'}
colores_meses = plt.cm.tab10(np.linspace(0, 1, 12))

fig, ax = plt.subplots(figsize=(10, 5))

for mes, col in zip(patron.columns, colores_meses):
    ax.plot(patron.index, patron[mes], label=nombres_meses.get(mes, mes),
            color=col, linewidth=1.2)

# Promedio global
ax.plot(patron.index, patron.mean(axis=1),
        color='black', linewidth=2.5, linestyle='--', label='Promedio global')

ax.set_xlabel('Hora del día (UTC-3)')
ax.set_ylabel('Velocidad media (m/s)')
ax.set_title('Patrón diurno de velocidad del viento')
ax.set_xticks(range(0, 24, 2))
ax.set_xticklabels([f'{h:02d}:00' for h in range(0, 24, 2)], rotation=45)
ax.legend(ncol=4, fontsize=8)
ax.grid(True, alpha=0.3)
fig.tight_layout()
```

[>] Convención de dirección — En meteorología la dirección indica desde dónde viene el viento (FROM). En oceanografía de corrientes, la convención es hacia dónde va (TO). Al graficar rosas, hay que tener claro cuál convención se está usando para que la rosa tenga sentido físico.

14 Figuras para informes

En un pipeline de informes automatizados las figuras no solo se visualizan en Spyder, sino que se guardan como archivos PNG para luego insertarlas en el informe Word. Esto implica cuidar resolución, tamaño, fuente y nomenclatura de archivos.

14.1 Estilo consistente

Definir un estilo base al comienzo del script garantiza que todas las figuras tengan el mismo aspecto:

```
import matplotlib.pyplot as plt
import matplotlib as mpl

# Estilo global
mpl.rcParams.update({
    'font.family':      'sans-serif',
    'font.size':        10,
    'axes.titlesize':   11,
    'axes.labelsize':   10,
    'xtick.labelsize':  9,
    'ytick.labelsize':  9,
    'legend.fontsize':  9,
    'figure.dpi':       100,
    'axes.grid':        True,
    'grid.alpha':       0.3,
    'axes.spines.top':  False,
    'axes.spines.right':False,
})
```

O usar un estilo predefinido:

```
plt.style.use('seaborn-v0_8-whitegrid')
```

14.2 Guardar con calidad para informe

```
fig.savefig(
    ruta_figura,
    dpi=150,               # 150 dpi es suficiente para Word
    bbox_inches='tight',   # elimina márgenes en blanco
    facecolor='white'      # fondo blanco (no transparente)
)
plt.close(fig)
```

14.3 Nomenclatura de archivos de figura

El autoinforme busca las figuras por nombre. Es importante seguir una convención estricta:

```
import os

carpeta_figuras = os.path.join(ruta_proyecto, 'figuras_magnitud')
os.makedirs(carpeta_figuras, exist_ok=True)

# Nomenclatura usada en el pipeline de corrientes
fig_serie.savefig(    os.path.join(carpeta_figuras, 'serie_velocidad.png'),
 ↪  dpi=150, bbox_inches='tight')
fig_rosa.savefig(     os.path.join(carpeta_figuras, 'rosa_corrientes_3m.png'),
 ↪  dpi=150, bbox_inches='tight')
fig_heatmap.savefig(  os.path.join(carpeta_figuras, 'heatmap_velocidad.png'),
 ↪  dpi=150, bbox_inches='tight')
```

14.4 Figuras por profundidad

Cuando se genera una figura por cada profundidad (rosas, vectores progresivos), se ittera sobre las capas y se guarda con sufijo:

```
profundidades = [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]

for prof in profundidades:
    fig, ax = plt.subplots(subplot_kw={'projection': 'polar'}, figsize=(6, 6))

    vel_capa = df[df['profundidad'] == prof]['velocidad'].values
    dir_capa = df[df['profundidad'] == prof]['direccion'].values

    # ... graficar ...

    nombre = f'rosa_corrientes_{prof}m.png'
    fig.savefig(os.path.join(carpeta_figuras, nombre), dpi=150, bbox_inches='tight')
    plt.close(fig)
    print(f'  Figura guardada: {nombre}')
```

14.5 Ciclo anual mensual

Para el informe de viento se genera una figura por mes con barras de percentiles:

```
fig, ax = plt.subplots(figsize=(10, 5))

meses_nombres = ['Sep', 'Oct', 'Nov', 'Dic', 'Ene', 'Feb', 'Mar']
x = np.arange(len(meses_nombres))

ax.bar(x, promedios, color='steelblue', label='Promedio', zorder=3)
ax.vlines(x, p05, p95, color='navy', linewidth=2, label='P5–P95', zorder=4)
ax.scatter(x, maximos, color='firebrick', s=40, zorder=5, label='Máximo')

ax.set_xticks(x)
ax.set_xticklabels(meses_nombres)
ax.set_ylabel('Velocidad (m/s)')
ax.set_title('Ciclo anual de velocidad del viento')
ax.legend()
fig.tight_layout()
fig.savefig(os.path.join(carpeta_figuras, 'ciclo_anual.png'), dpi=150,
 ↪  bbox_inches='tight')
plt.close(fig)
```

14.6 Insertar figuras en Word

Una vez guardadas, las figuras se insertan en la plantilla Word desde el autoinforme. El proceso se describe en detalle en el capítulo de python-docx, pero el patrón básico es:

```
from docx import Document
from docx.shared import Cm

doc = Document('plantilla.docx')

for parrafo in doc.paragraphs:
    if '[FIGURA_ROSA]' in parrafo.text:
        parrafo.clear()
        run = parrafo.add_run()
        run.add_picture('figuras_magnitud/rosa_corrientes_3m.png', width=Cm(12))
        break

doc.save('informe_final.docx')
```

14.7 Figura multipanel para series de oleaje

El informe incluye una figura con tres paneles sincronizados (Hm0, Tm, Dm):

```
fig, axes = plt.subplots(3, 1, figsize=(14, 8), sharex=True,
                         gridspec_kw={'hspace': 0.08})

# Panel 1: Altura Hm0
axes[0].plot(df.index, df['Hm0'], color='navy', linewidth=0.7)
axes[0].set_ylabel('Hm0 (m)')
axes[0].set_ylim(0, df['Hm0'].max() * 1.1)

# Panel 2: Periodo Tm
axes[1].plot(df.index, df['Tm'], color='teal', linewidth=0.7)
axes[1].set_ylabel('Tm (s)')

# Panel 3: Dirección Dm (puntos, no línea)
axes[2].scatter(df.index, df['Dm'], s=1, color='darkorange', alpha=0.6)
axes[2].set_ylabel('Dm (°)')
axes[2].set_ylim(0, 360)
axes[2].set_yticks([0, 90, 180, 270, 360])
axes[2].set_yticklabels(['N', 'E', 'S', 'O', 'N'])
axes[2].set_xlabel('Fecha')

# Formatear eje X con fechas
import matplotlib.dates as mdates
axes[2].xaxis.set_major_formatter(mdates.DateFormatter('%b %Y'))
axes[2].xaxis.set_major_locator(mdates.MonthLocator())
plt.setp(axes[2].xaxis.get_majorticklabels(), rotation=30, ha='right')

fig.savefig('serie_oleaje.png', dpi=150, bbox_inches='tight')
plt.close(fig)
```

14.8 GridSpec — layouts complejos

`subplots` crea paneles de igual tamaño. Cuando se necesita que algunos paneles sean más anchos o altos que otros, se usa `GridSpec`:

```
import matplotlib.gridspec as gridspec

fig = plt.figure(figsize=(14, 8))
gs  = gridspec.GridSpec(2, 3, figure=fig, hspace=0.35, wspace=0.4)

# Panel grande a la izquierda (ocupa toda la columna 0)
ax_serie  = fig.add_subplot(gs[:, 0])    # filas 0:2, columna 0

# Dos paneles pequeños a la derecha
ax_hist   = fig.add_subplot(gs[0, 1:])  # fila 0, columnas 1-2
ax_rosa   = fig.add_subplot(gs[1, 1:], projection='polar')  # fila 1, columnas 1-2
```

```
# Layout con ratios de tamaño explícitos
gs = gridspec.GridSpec(
    3, 1,
    height_ratios=[3, 1, 1],   # panel superior 3× más alto que los otros dos
    hspace=0.05
)
ax_vel = fig.add_subplot(gs[0])
ax_dir = fig.add_subplot(gs[1], sharex=ax_vel)
ax_qc  = fig.add_subplot(gs[2], sharex=ax_vel)
```

14.9 Twin axes — dos escalas en el mismo panel

Cuando se grafican dos variables con distinta escala en el mismo panel (por ejemplo, velocidad y temperatura), se usa un eje secundario:

```
fig, ax1 = plt.subplots(figsize=(12, 4))

# Eje izquierdo: velocidad
ax1.plot(df.index, df['velocidad'], color='steelblue', linewidth=0.8,
 ↪  label='Velocidad')
ax1.set_ylabel('Velocidad (m/s)', color='steelblue')
ax1.tick_params(axis='y', labelcolor='steelblue')

# Eje derecho: temperatura — comparte el eje X
ax2 = ax1.twinx()
ax2.plot(df.index, df['temperatura'], color='firebrick', linewidth=0.8,
         linestyle='--', label='Temperatura')
ax2.set_ylabel('Temperatura (°C)', color='firebrick')
ax2.tick_params(axis='y', labelcolor='firebrick')

# Leyenda combinada de ambos ejes
lineas1, labels1 = ax1.get_legend_handles_labels()
lineas2, labels2 = ax2.get_legend_handles_labels()
ax1.legend(lineas1 + lineas2, labels1 + labels2, loc='upper left')

fig.tight_layout()
```

[!] Cuándo no usar twin axes — Dos ejes con distinta escala en el mismo panel pueden inducir correlaciones visuales falsas — el lector ve las curvas cruzarse o alinearse dependiendo de cómo se elijan las escalas. Preferir paneles separados con sharex=True cuando las variables son independientes. Reservar twinx para cuando la relación entre las dos variables es el punto central de la figura.

14.10 Tamaño de figura para Word

El ancho de la zona de texto de un documento Word A4 con márgenes estándar es **~15.5 cm**. Para que las figuras queden alineadas y a escala correcta al insertarlas:

```python
# Figura de ancho completo (serie temporal, heatmap)
fig, ax = plt.subplots(figsize=(15.5/2.54, 5/2.54))   # cm → pulgadas (/2.54)

# Figura de media página (rosa, histograma)
fig, ax = plt.subplots(figsize=(7.5/2.54, 7.5/2.54))

# Guardar siempre con bbox_inches='tight' para no perder espacio en blanco
fig.savefig('figura.png', dpi=150, bbox_inches='tight', facecolor='white')
```

```python
# Función utilitaria para convertir cm a pulgadas
def cm(x): return x / 2.54

fig, ax = plt.subplots(figsize=(cm(15.5), cm(6)))
```

[>] Cuándo usar sharex=True — En series temporales con múltiples variables (Hm0, Tm, Dm), sharex=True sincroniza el zoom y el paneo entre paneles. Si el usuario hace zoom en un panel en Spyder, todos los demás se actualizan juntos.

15 Gráficos interactivos y el event loop

Matplotlib puede mostrar figuras estáticas, abrir ventanas interactivas con zoom y pan, responder a clics del usuario, y generar animaciones. Cada uno de estos modos tiene restricciones técnicas que conviene entender antes de mezclarlos en un script.

15.1 El event loop — por qué existe la limitación

Una ventana de escritorio (Qt, Tk, cualquier GUI) necesita estar constantemente ”escuchando” eventos: clics, redimensionado, movimiento del mouse. Este proceso de escucha continua es el **event loop** — un bucle que corre indefinidamente hasta que se cierra la ventana.

Python es de un solo hilo por defecto. Cuando el event loop de una ventana está corriendo, Python no puede ejecutar código al mismo tiempo. Por eso `plt.show()` en un script normal bloquea: hasta que el usuario cierra la figura, el script no avanza.

En Spyder e IPython esto se resuelve de otra forma: el kernel de IPython corre un event loop propio que puede procesar eventos de ventana en paralelo con la ejecución del código. Es lo que permite que escribas en la consola mientras una figura de Tkinter está abierta.

15.2 Los backends de matplotlib

El **backend** es el motor que matplotlib usa para renderizar y mostrar figuras. Se divide en dos categorías:

Backend	Tipo	Descripción
`Agg`	No interactivo	Renderiza a memoria. Sin ventana. Solo para guardar archivos.
`TkAgg`	Interactivo	Ventana Tkinter. Estable en Spyder.
`Qt5Agg`	Interactivo	Ventana Qt5. Sin conflicto en VSCode; puede interferir en Spyder.
`inline`	IPython	Renderiza a imagen estática embebida en la consola. No es un backend real — es un hook de IPython sobre Agg.

```
# Ver el backend activo
import matplotlib
matplotlib.get_backend()   # 'module://matplotlib_inline.backend_inline' en Spyder
 ↪  inline

# Cambiar desde la consola IPython (antes de crear cualquier figura)
%matplotlib tk      # Tkinter interactivo
%matplotlib qt5     # Qt5 interactivo
%matplotlib inline  # volver a inline

# En un script .py fuera de IPython, antes de importar pyplot:
import matplotlib
matplotlib.use('Agg')    # solo para guardar, sin ventana
import matplotlib.pyplot as plt
```

15.3 El problema de mezclar backends en un script

El backend se inicializa la primera vez que se importa `matplotlib.pyplot`. Una vez activo, **no se puede cambiar limpiamente en la misma sesión**: matplotlib lanza una advertencia y el comportamiento es impredecible.

```
# Esto no funciona bien:
%matplotlib inline
import matplotlib.pyplot as plt
plt.plot([1, 2, 3])     # figura inline — OK

%matplotlib tk          # advertencia: backend ya inicializado
plt.figure()            # puede no abrir ventana, o abrirla sin event loop
```

Solución práctica: decidir un backend al inicio y no cambiarlo. Si se necesitan figuras interactivas Y figuras guardadas en el mismo script, usar el backend interactivo y guardar con `fig.savefig()` — eso funciona desde cualquier backend, incluido Tk y Qt5.

```
# Patrón correcto: backend interactivo, guardar explícitamente
%matplotlib tk

fig, ax = plt.subplots()
ax.plot(df['velocidad'])
plt.show()                              # ventana interactiva para explorar

fig.savefig('velocidad.png', dpi=150)   # guardar también — funciona igual
plt.close(fig)
```

15.4 plt.ion() — modo interactivo sin bloquear

`plt.ion()` activa el modo interactivo: las figuras se actualizan inmediatamente cuando se modifica el gráfico, sin necesidad de llamar a `plt.show()`. El script sigue corriendo sin que la ventana bloquee la ejecución.

```
%matplotlib tk
import matplotlib.pyplot as plt
import numpy as np

plt.ion()   # activar modo interactivo

fig, ax = plt.subplots()
linea, = ax.plot([], [])
ax.set_xlim(0, 100)
ax.set_ylim(-1, 1)

# Actualizar la figura en un loop — simula datos llegando en tiempo real
for i in range(100):
    x = np.arange(i)
    y = np.sin(x * 0.3)
    linea.set_data(x, y)
    ax.relim()
    fig.canvas.draw()
    plt.pause(0.05)   # procesar eventos de ventana y esperar 50 ms

plt.ioff()   # volver al modo normal
```

`plt.pause(t)` es fundamental en este patrón: le da tiempo al event loop de la ventana para procesar eventos (zoom, clic, redimensionado) y luego devuelve el control al script. Sin `pause`, la ventana se congela.

15.5 matplotlib.widgets — interactividad básica dentro del canvas

Los widgets de matplotlib viven dentro del canvas de la figura — no requieren construir una aplicación Qt o Tk completa. Son suficientes para ajustar parámetros de forma interactiva en un script exploratorio.

```
%matplotlib tk
import matplotlib.pyplot as plt
import matplotlib.widgets as widgets
import numpy as np

# Serie temporal de ejemplo
t   = np.linspace(0, 10, 500)
vel = np.sin(2 * np.pi * t)

fig, ax = plt.subplots(figsize=(10, 5))
plt.subplots_adjust(bottom=0.25)   # espacio para el slider
```

```
linea, = ax.plot(t, vel)
ax.set_xlabel('Tiempo (h)')
ax.set_ylabel('Velocidad (m/s)')

# Slider para cambiar la frecuencia
ax_slider = plt.axes([0.2, 0.08, 0.6, 0.04])
slider = widgets.Slider(ax_slider, 'Frecuencia (Hz)', 0.1, 5.0, valinit=1.0)

def actualizar(valor):
    freq = slider.val
    linea.set_ydata(np.sin(2 * np.pi * freq * t))
    fig.canvas.draw_idle()

slider.on_changed(actualizar)
plt.show()
```

Otros widgets disponibles: `Button`, `CheckButtons` (checkboxes), `RadioButtons`, `TextBox`, `RectangleSelector` (seleccionar región con el mouse).

Limitación: los widgets de matplotlib son básicos. No hay menús, ni cuadros de diálogo para abrir archivos, ni tablas. Para eso se necesita PyQt5.

15.6 FuncAnimation — animaciones

`FuncAnimation` crea una animación llamando una función de actualización para cada frame. Puede mostrarse en vivo o guardarse como GIF o MP4.

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import numpy as np

fig, ax = plt.subplots(figsize=(10, 4))
ax.set_xlim(0, 4 * np.pi)
ax.set_ylim(-1.5, 1.5)
ax.set_xlabel('Tiempo')
ax.set_ylabel('Velocidad (m/s)')

linea, = ax.plot([], [], linewidth=1.5)
x = np.linspace(0, 4 * np.pi, 300)

def init():
    linea.set_data([], [])
    return (linea,)

def update(frame):
    y = np.sin(x - 0.05 * frame)   # ola que avanza
    linea.set_data(x, y)
```

```
    return (linea,)

ani = animation.FuncAnimation(
    fig,
    update,
    frames=200,
    init_func=init,
    interval=50,    # ms entre frames
    blit=True       # solo redibuja lo que cambió — más eficiente
)

plt.show()
```

15.6.1 Guardar la animación

```
# GIF — requiere pillow: pip install pillow
ani.save('corriente.gif', writer='pillow', fps=20, dpi=100)

# MP4 — requiere ffmpeg instalado en el sistema
ani.save('corriente.mp4', writer='ffmpeg', fps=30, dpi=150)
```

15.6.2 Ejemplo oceanográfico: evolución de perfil vertical

```
fig, ax = plt.subplots(figsize=(4, 8))
ax.set_xlim(0, 1)
ax.set_ylim(-25, 0)
ax.set_xlabel('Velocidad (m/s)')
ax.set_ylabel('Profundidad (m)')
ax.invert_yaxis()

profundidades = np.array([-3, -5, -7, -9, -11, -13, -15, -17, -19, -21, -23])
linea, = ax.plot([], profundidades, 'o-', color='steelblue')
titulo = ax.set_title('')

def update(frame):
    # Perfil que oscila con la marea (ejemplo sintético)
    vel = 0.3 + 0.2 * np.sin(2 * np.pi * frame / 60) * np.exp(profundidades / 10)
    linea.set_xdata(vel)
    titulo.set_text(f'Hora: {frame:03d}')
    return linea, titulo

ani = animation.FuncAnimation(fig, update, frames=120, interval=100, blit=True)
plt.tight_layout()
plt.show()
```

15.7 PyQt5 — herramientas con interfaz propia

Cuando `matplotlib.widgets` no es suficiente — se necesita navegar entre archivos, tener checkboxes y botones con lógica compleja, o mostrar una tabla junto con el gráfico — se puede embeber matplotlib dentro de una ventana Qt5 construida a medida.

Este nivel de desarrollo es más complejo y está fuera del scope del pipeline de corrientes, pero aparece cuando se construyen herramientas reutilizables: un editor de control de calidad de ADCP, una interfaz para digitalizar batimetría, un visor de salidas de CROCO.

La estructura básica:

```
import sys
import numpy as np
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget,
                             QVBoxLayout, QPushButton, QLabel)
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg
from matplotlib.figure import Figure

class VentanaPrincipal(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Visor de corrientes')
        self.resize(900, 600)

        # Widget central con layout vertical
        central = QWidget()
        layout  = QVBoxLayout(central)
        self.setCentralWidget(central)

        # Canvas de matplotlib embebido
        fig = Figure(figsize=(8, 4))
        self.canvas = FigureCanvasQTAgg(fig)
        self.ax = fig.add_subplot(111)
        layout.addWidget(self.canvas)

        # Botón de Qt
        boton = QPushButton('Actualizar')
        boton.clicked.connect(self.actualizar_figura)
        layout.addWidget(boton)

        self.label = QLabel('Listo.')
        layout.addWidget(self.label)

        self.actualizar_figura()

    def actualizar_figura(self):
        t   = np.linspace(0, 24, 144)
```

```
        vel = 0.3 + 0.15 * np.sin(2 * np.pi * t / 12.4)   # ciclo mareal
        self.ax.clear()
        self.ax.plot(t, vel, color='steelblue')
        self.ax.set_xlabel('Hora')
        self.ax.set_ylabel('Velocidad (m/s)')
        self.canvas.draw()
        self.label.setText(f'Velocidad máxima: {vel.max():.3f} m/s')

if __name__ == '__main__':
    app = QApplication(sys.argv)
    ventana = VentanaPrincipal()
    ventana.show()
    sys.exit(app.exec_())
```

Nota sobre Spyder: este script debe correrse desde la terminal o desde VSCode, no desde la consola de Spyder. `QApplication` crea su propia instancia de Qt y entra en su propio event loop (`app.exec_()`), lo que entra en conflicto con el event loop Qt que ya usa Spyder. Desde la terminal funciona perfectamente.

```
python visor_corrientes.py
```

15.8 Cuándo usar cada enfoque

Necesidad	Solución
Explorar datos, hacer zoom y pan	Backend Tk o Qt5, `%matplotlib tk`
Guardar figuras para informe	`Agg` o cualquier backend + `savefig()`
Ajustar parámetros en vivo con un slider	`matplotlib.widgets`
Mostrar evolución temporal / animación	`FuncAnimation` → guardar GIF/MP4
Herramienta reutilizable con botones y menús	PyQt5 + FigureCanvasQTAgg
Script automático sin display (servidor)	`matplotlib.use('Agg')` antes del import

15.9 Reglas prácticas

1. **Elegir un backend al inicio del script y no cambiarlo.** Si se necesita guardar y también ver la figura, usar backend interactivo y llamar a `savefig()` — funciona igual.
2. **Separar scripts de exploración de scripts de producción.** El script que genera las figuras del informe siempre usa `Agg` (o simplemente `savefig` con cualquier backend) y corre sin intervención. El script de exploración usa Tk para inspeccionar.
3. **`plt.show()` bloquea en scripts fuera de IPython.** Si el script tiene código después de `plt.show()`, ese código no corre hasta que el usuario cierra la ventana. Usar `plt.ion()` + `plt.pause()` para evitar el bloqueo.

4. **PyQt5 no va en Spyder — va en un script independiente.** La aplicación Qt crea su propia `QApplication`; si Spyder ya creó una, el conflicto produce cierres inesperados o que la ventana no aparezca.

16 Estadísticas circulares

Los datos de dirección (viento, corrientes, oleaje) son **circulares**: 0° y 360° son el mismo ángulo, por lo que promediar linealmente da resultados absurdos. Si una serie tiene mediciones de 10° y 350°, el promedio lineal es 180° (Sur), pero el promedio circular correcto es 0° (Norte). Python incluye herramientas para manejar esta circularidad correctamente.

16.1 Media y desviación estándar circular

`scipy.stats` provee `circmean` y `circstd` que trabajan directamente en grados o radianes:

```
from scipy.stats import circmean, circstd
import numpy as np

# Las funciones esperan radianes por defecto; high/low definen el rango
direcciones_deg = np.array([350, 5, 10, 355, 2, 358])

media_circ = circmean(direcciones_deg, high=360, low=0)
std_circ   = circstd( direcciones_deg, high=360, low=0)

print(f"Media circular:  {media_circ:.1f}°")
print(f"Desv. estándar:  {std_circ:.1f}°")
```

16.1.1 Cálculo manual con vectores unitarios

El mismo resultado se obtiene convirtiendo cada ángulo a un vector unitario y promediando sus componentes:

```
def media_circular(angulos_deg):
    rad = np.radians(angulos_deg)
    sin_m = np.nanmean(np.sin(rad))
    cos_m = np.nanmean(np.cos(rad))
    media = np.degrees(np.arctan2(sin_m, cos_m)) % 360
    return media

def concentracion_circular(angulos_deg):
    """Longitud del vector resultante medio (R̄). Rango [0, 1]: 1 = perfectamente
    ↪  concentrado."""
    rad = np.radians(angulos_deg)
    sin_m = np.nanmean(np.sin(rad))
    cos_m = np.nanmean(np.cos(rad))
    return np.sqrt(sin_m**2 + cos_m**2)

print(f"Media:        {media_circular(direcciones_deg):.1f}°")
print(f"Concentración R̄: {concentracion_circular(direcciones_deg):.3f}")
```

16.2 Dirección predominante

La dirección predominante no es necesariamente la media circular; en vientos bimodales (p. ej. brisa diurna que alterna Norte/Sur) la media puede apuntar a un sector vacío. En ese caso se usa la **moda** por octante:

```
def direccion_predominante(direcciones_deg, n_sectores=8):
    """Retorna el centro del sector con mayor frecuencia."""
    ancho = 360 / n_sectores
    sectores = (np.floor((direcciones_deg % 360) / ancho) * ancho + ancho /
↪   2).astype(int)
    valores, conteos = np.unique(sectores, return_counts=True)
    return valores[np.argmax(conteos)]

print(f"Dirección predominante: {direccion_predominante(direcciones_deg)}°")
```

16.3 Diferencia angular

Para calcular la diferencia entre dos ángulos de forma correcta (resultado siempre en [-180°, 180°]):

```
def diferencia_angular(a, b):
    """Diferencia a - b en el rango [-180, 180]."""
    diff = (a - b + 180) % 360 - 180
    return diff

# Ejemplo: diferencia entre dirección modelada y observada
dir_modelo = 320.0
dir_obs    = 15.0
print(f"Diferencia: {diferencia_angular(dir_modelo, dir_obs):.1f}°")
# → -55.0° (el modelo es 55° más hacia el oeste)
```

16.4 Error cuadrático medio circular

Métrica estándar para evaluar el desempeño de un modelo de dirección:

```
def rmse_circular(dir_modelo, dir_obs):
    diffs = diferencia_angular(np.asarray(dir_modelo), np.asarray(dir_obs))
    return np.sqrt(np.nanmean(diffs**2))

# Comparar dirección de corriente modelada vs. medida
rmse_dir = rmse_circular(df['dir_modelo'], df['dir_obs'])
print(f"RMSE direccional: {rmse_dir:.1f}°")
```

16.5 Estadísticas por sector en el pipeline

En el autoinforme de corrientes se calculan estadísticas separadas por octante para la tabla de incidencia:

```
import pandas as pd

def tabla_incidencia(df, col_vel, col_dir, n_sectores=8):
    """
    Retorna DataFrame con frecuencia, velocidad media y máxima por sector.
    """
    ancho = 360 / n_sectores
    nombres_sectores = ['N', 'NE', 'E', 'SE', 'S', 'SO', 'O', 'NO']

    # Asignar sector a cada registro
    df = df.copy()
    df['sector_idx'] = (np.floor((df[col_dir] % 360) / ancho)).astype(int) %
↪   n_sectores
    df['sector'] = df['sector_idx'].map(dict(enumerate(nombres_sectores)))

    resumen = df.groupby('sector').agg(
        n=(col_vel, 'count'),
        vel_media=(col_vel, 'mean'),
        vel_max=(col_vel, 'max')
    )
    resumen['frecuencia_%'] = (resumen['n'] / len(df) * 100).round(1)

    return resumen[['frecuencia_%', 'vel_media', 'vel_max']]

tabla = tabla_incidencia(df_corrientes, 'velocidad', 'direccion')
print(tabla.to_string())
```

16.6 Histograma direccional

Visualizar la distribución de direcciones en una barra circular es más informativo que un histograma cartesiano:

```
import matplotlib.pyplot as plt

def histograma_direccional(direcciones_deg, n_sectores=16, titulo=''):
    ancho = 360 / n_sectores
    bordes = np.arange(0, 361, ancho)
    conteos, _ = np.histogram(direcciones_deg % 360, bins=bordes)
    centros = bordes[:-1] + ancho / 2

    fig, ax = plt.subplots(subplot_kw={'projection': 'polar'}, figsize=(6, 6))
    ax.set_theta_zero_location('N')
    ax.set_theta_direction(-1)

    theta = np.radians(centros)
    ax.bar(theta, conteos / len(direcciones_deg) * 100,
           width=np.radians(ancho * 0.85),
```

```
            color='steelblue', edgecolor='white', alpha=0.8)

    ax.set_title(titulo or 'Distribución direccional (%)', pad=15)
    return fig, ax
```

[>] Calma o datos faltantes — Cuando la velocidad es cero (calma), la dirección no tiene significado físico. Antes de calcular estadísticas circulares, filtrar los registros con velocidad inferior al umbral de arranque del instrumento (típicamente 0.02 m/s en correntómetros, 0.5 m/s en anemómetros).

17 Análisis espectral

El análisis espectral descompone una serie temporal en sus componentes de frecuencia, mostrando cuánta energía hay en cada período. En oceanografía se aplica principalmente al oleaje (para obtener el espectro de energía y extraer parámetros como Hm0 y Tp) y a las corrientes (para identificar mareas, inerciales y otras señales periódicas).

17.1 Preparar la serie antes del análisis

La FFT y Welch requieren una serie **sin NaN** y con **resolución temporal regular**. Los datos oceanográficos frecuentemente tienen huecos. El paso previo es siempre verificar y limpiar:

```
import pandas as pd
import numpy as np

# Verificar que el índice es regular (sin saltos de tiempo)
diffs = df.index.to_series().diff().dropna()
dt_esperado = pd.Timedelta(minutes=10)
irregulares = diffs[diffs != dt_esperado]
if not irregulares.empty:
    print(f"Saltos de tiempo en: {irregulares.index.tolist()}")

# Completar el índice temporal si hay filas faltantes
df = df.resample('10min').asfreq()   # inserta NaN donde faltaban filas

# Interpolación lineal para huecos cortos (≤ 1 hora = 6 muestras)
df['velocidad'] = df['velocidad'].interpolate(method='linear', limit=6)

# Verificar que no quedan NaN antes de la FFT
n_nan = df['velocidad'].isna().sum()
if n_nan > 0:
    print(f"  ! {n_nan} NaN no interpolados — se eliminarán los extremos")
    df = df.dropna(subset=['velocidad'])

serie = df['velocidad'].values
```

[!] NaN en la FFT — np.fft.rfft y scipy.signal.welch no toleran NaN — producen un espectro de NaN sin error ni aviso. Siempre verificar con np.isnan(serie).any() antes de transformar.

17.2 Transformada de Fourier con NumPy

La FFT es el punto de partida para cualquier análisis espectral:

```
import numpy as np

# Serie temporal de velocidad de corriente (N datos, dt segundos entre muestras)
N  = len(serie)
dt = 600   # 10 minutos en segundos
fs = 1 / dt   # frecuencia de muestreo (Hz)

# FFT
espectro = np.fft.rfft(serie - serie.mean())   # rfft: solo frecuencias positivas
freqs    = np.fft.rfftfreq(N, d=dt)            # Hz

# Potencia espectral
potencia = (np.abs(espectro) ** 2) / (N * fs)

# Convertir frecuencias a períodos (en horas)
with np.errstate(divide='ignore'):
    periodos_h = 1 / freqs / 3600
```

17.3 Densidad espectral de potencia con Welch

El método de Welch es más robusto que la FFT directa porque promedia segmentos solapados, reduciendo el ruido estadístico:

```
from scipy.signal import welch

# nperseg: número de muestras por segmento (elegir ~10% de N, potencia de 2)
freqs_w, psd = welch(
    serie - serie.mean(),
    fs=fs,
    nperseg=512,
    noverlap=256,
    window='hann'
)

periodos_w_h = 1 / freqs_w / 3600

import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(10, 5))
ax.semilogy(periodos_w_h, psd, color='navy', linewidth=0.8)
ax.set_xlim(0, 50)   # períodos hasta 50 h
ax.set_xlabel('Período (h)')
ax.set_ylabel('PSD ((m/s)²/Hz)')
ax.set_title('Espectro de densidad de potencia — Corriente 7 m')
ax.grid(True, which='both', alpha=0.3)

# Marcar componentes de marea
```

```
for periodo, nombre in [(24, 'K1'), (12.42, 'M2'), (6.21, 'M4')]:
    ax.axvline(periodo, color='firebrick', linewidth=0.8, linestyle='--', alpha=0.7)
    ax.text(periodo, ax.get_ylim()[1] * 0.5, nombre, color='firebrick', fontsize=8)
```

17.4 Espectro de oleaje

Los parámetros integrados (Hm0, Tm02, Tp) se derivan del espectro de densidad de energía:

```
def espectro_oleaje(eta, dt_s, nperseg=512):
    """
    Calcula el espectro de energía de una serie de superficie libre.
    Retorna freqs (Hz) y S (m²/Hz).
    """
    freqs, psd = welch(eta - eta.mean(), fs=1/dt_s,
                       nperseg=nperseg, window='hann')
    return freqs[1:], psd[1:]   # descartar componente DC

def momento_espectral(freqs, S, n):
    """Momento espectral de orden n: mₙ = ∫ fⁿ S(f) df."""
    return np.trapz(freqs**n * S, freqs)

freqs, S = espectro_oleaje(eta, dt_s=1800)   # muestras cada 30 min

m0  = momento_espectral(freqs, S, 0)
m2  = momento_espectral(freqs, S, 2)

Hm0  = 4 * np.sqrt(m0)         # altura significativa espectral (m)
Tm02 = np.sqrt(m0 / m2)        # período medio (s)
Tp   = 1 / freqs[np.argmax(S)] # período de pico (s)

print(f"Hm0  = {Hm0:.2f} m")
print(f"Tm02 = {Tm02:.1f} s")
print(f"Tp   = {Tp:.1f} s")
```

17.5 Visualizar el espectro de oleaje

```
fig, ax = plt.subplots(figsize=(9, 5))

ax.fill_between(freqs, S, alpha=0.3, color='steelblue')
ax.plot(freqs, S, color='navy', linewidth=1)

# Marcar frecuencia de pico
fp = freqs[np.argmax(S)]
ax.axvline(fp, color='firebrick', linestyle='--', linewidth=1,
           label=f'fp = {fp:.4f} Hz  (Tp = {1/fp:.1f} s)')
```

```
# Separar swell y viento localmente (límite convencional 0.1 Hz)
ax.axvspan(freqs.min(), 0.1, alpha=0.08, color='green', label='Swell (<0.1 Hz)')
ax.axvspan(0.1, freqs.max(),  alpha=0.08, color='orange', label='Sea (>0.1 Hz)')

ax.set_xlabel('Frecuencia (Hz)')
ax.set_ylabel('S(f)  (m²/Hz)')
ax.set_title(f'Espectro de oleaje — Hm0 = {Hm0:.2f} m, Tp = {Tp:.1f} s')
ax.legend(fontsize=9)
ax.grid(True, alpha=0.3)
fig.tight_layout()
```

17.6 Identificar componentes de marea

Las mareas son señales periódicas con frecuencias bien conocidas:

```
MAREAS = {
    'K1':  23.93,   # diurna
    'O1':  25.82,
    'M2':  12.42,   # semidiurna principal
    'S2':  12.00,
    'N2':  12.66,
    'M4':   6.21,   # cuarto-diurna
}

import pandas as pd

df_mareas = []
for nombre, periodo_h in MAREAS.items():
    f_central = 1 / (periodo_h * 3600)
    mascara = np.abs(freqs_w - f_central) < f_central * 0.05
    if mascara.any():
        energia = np.trapz(psd[mascara], freqs_w[mascara])
        df_mareas.append({'componente': nombre, 'período_h': periodo_h, 'energía':
↪   energia})

df_m = pd.DataFrame(df_mareas).sort_values('energía', ascending=False)
print(df_m.to_string(index=False))
```

17.7 Espectrograma (espectro en tiempo)

Para ver cómo evoluciona el espectro a lo largo del tiempo:

```
from scipy.signal import spectrogram

f_sg, t_sg, Sxx = spectrogram(
    serie - serie.mean(),
    fs=fs,
    nperseg=256,
    noverlap=192,
    window='hann'
)

fig, ax = plt.subplots(figsize=(12, 5))
im = ax.pcolormesh(t_sg / 3600, 1 / f_sg[1:] / 3600, np.log10(Sxx[1:]),
                   cmap='viridis', shading='gouraud')
ax.set_ylim(0, 30)   # períodos hasta 30 h
ax.set_xlabel('Tiempo (h)')
ax.set_ylabel('Período (h)')
ax.set_title('Espectrograma de velocidad de corriente')
fig.colorbar(im, ax=ax, label='log₁₀ PSD')
```

[>] Efecto de ventana — Aplicar una ventana (Hann, Hamming) antes de la FFT reduce las fugas espectrales causadas por el truncamiento de la serie. welch aplica la ventana internamente; si usas np.fft.rfft directamente, multiplica la serie por np.hanning(N) antes de transformar.

[!] Resolución espectral — La resolución frecuencial es $\Delta f = 1 / (N \times dt)$. Series cortas tienen baja resolución y no pueden separar componentes de período similar (p. ej. M2 y S2, que difieren solo 0.42 h). Para estudios de marea se necesitan al menos 30 días de datos.

18 Filtros e interpolación

Los datos oceanográficos rara vez llegan limpios: tienen ruido de alta frecuencia, datos faltantes (NaN) y resolución temporal variable. El filtrado elimina frecuencias no deseadas; la interpolación rellena huecos; el remuestreo homogeneiza la resolución temporal. Estas tres operaciones son parte estándar de cualquier pipeline de preprocesamiento.

18.1 Media móvil (filtro de paso bajo simple)

Es el filtro más sencillo y el más rápido de aplicar con pandas:

```
import pandas as pd

# Ventana de 1 hora en datos de 10 min → 6 muestras
df['vel_suavizada'] = df['velocidad'].rolling(window=6, center=True).mean()

# Con mínimo de datos para no producir NaN en los extremos
df['vel_suavizada'] = df['velocidad'].rolling(window=6, center=True,
 ↪  min_periods=3).mean()
```

La media móvil atenúa bien el ruido pero tiene una respuesta de frecuencia imperfecta: no rechaza completamente las frecuencias altas y tiene fugas. Para filtrado más preciso se usa Butterworth.

18.2 Filtro Butterworth (scipy)

El filtro Butterworth tiene una respuesta plana en la banda de paso y cae abruptamente en la banda de rechazo:

```
from scipy.signal import butter, filtfilt
import numpy as np

def filtro_paso_bajo(serie, fs_hz, fc_hz, orden=4):
    """
    Aplica filtro Butterworth de paso bajo.

    fs_hz : frecuencia de muestreo (Hz)
    fc_hz : frecuencia de corte (Hz)
    """
    nyq = fs_hz / 2
    wn  = fc_hz / nyq   # frecuencia de corte normalizada [0, 1]
    b, a = butter(orden, wn, btype='low')
    return filtfilt(b, a, serie)   # filtfilt: sin desfase de fase

# Datos cada 10 min → fs = 1/600 Hz
fs = 1 / 600
```

```
# Cortar periodos menores a 2 horas → fc = 1/(2*3600) Hz
fc = 1 / (2 * 3600)

vel_filtrada = filtro_paso_bajo(df['velocidad'].dropna().values, fs, fc)
```

18.2.1 Filtro de paso alto (eliminar marea)

Para separar la corriente residual (sin marea) de la señal total:

```
def filtro_paso_alto(serie, fs_hz, fc_hz, orden=4):
    nyq = fs_hz / 2
    wn  = fc_hz / nyq
    b, a = butter(orden, wn, btype='high')
    return filtfilt(b, a, serie)

# Eliminar componentes con período mayor a 30 horas
fc_alta = 1 / (30 * 3600)
vel_residual = filtro_paso_alto(df['velocidad'].dropna().values, fs, fc_alta)
```

18.2.2 Filtro de banda (aislar marea semidiurna)

```
def filtro_banda(serie, fs_hz, fc_low_hz, fc_high_hz, orden=4):
    nyq = fs_hz / 2
    wn  = [fc_low_hz / nyq, fc_high_hz / nyq]
    b, a = butter(orden, wn, btype='band')
    return filtfilt(b, a, serie)

# Aislar M2: banda 11–14 horas de período
fc_low  = 1 / (14 * 3600)
fc_high = 1 / (11 * 3600)
marea_M2 = filtro_banda(df['velocidad'].dropna().values, fs, fc_low, fc_high)
```

18.3 Interpolación de NaN

18.3.1 Interpolación lineal con pandas

```
# Interpolar NaN por interpolación lineal (por defecto)
df['vel_interp'] = df['velocidad'].interpolate(method='linear')

# Limitar la cantidad de NaN consecutivos a rellenar
df['vel_interp'] = df['velocidad'].interpolate(method='linear', limit=3)

# Interpolar también en los extremos (por defecto pandas no interpola bordes)
df['vel_interp'] = df['velocidad'].interpolate(method='linear',
                                               limit_direction='both')
```

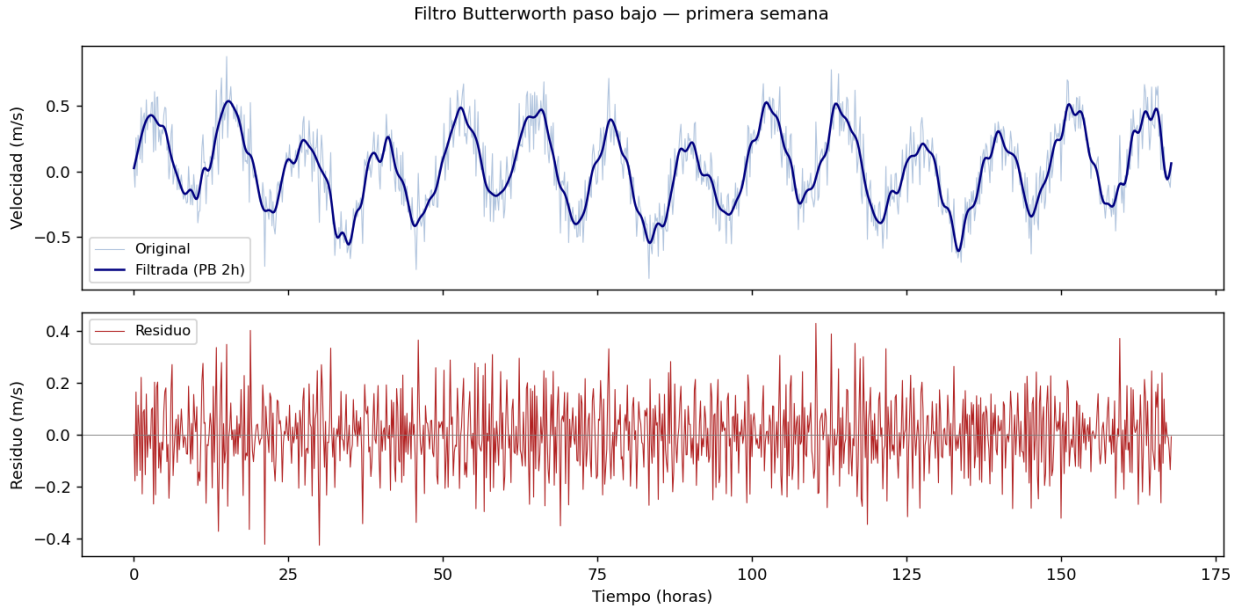


Figura 5: Filtro Butterworth paso bajo — señal original vs filtrada

18.3.2 Interpolación cúbica con scipy

Para series con huecos pequeños donde se quiere una curva más suave:

```
from scipy.interpolate import interp1d

# Separar datos válidos de NaN
mascara_valida = df['velocidad'].notna()
t_valido = df.index[mascara_valida].astype(np.int64)   # timestamps como enteros
v_valido = df['velocidad'][mascara_valida].values

# Construir interpolador
f_interp = interp1d(t_valido, v_valido, kind='cubic',
                    bounds_error=False, fill_value='extrapolate')

# Aplicar a todos los tiempos
t_todos = df.index.astype(np.int64)
df['vel_cubica'] = f_interp(t_todos)
```

18.4 Remuestreo temporal

18.4.1 Reducir resolución (downsampling)

```python
# Datos cada 10 min → promedios horarios
df_horario = df.resample('1h').mean()

# Estadísticas adicionales al remuestrear
df_hora = df.resample('1h').agg({
    'velocidad': ['mean', 'max', 'std'],
    'direccion': 'mean'   # ojo: dirección requiere media circular
})
df_hora.columns = ['vel_media', 'vel_max', 'vel_std', 'dir_media']
```

18.4.2 Aumentar resolución (upsampling)

```python
# Datos horarios → cada 10 minutos por interpolación lineal
df_10min = df_horario.resample('10min').interpolate(method='linear')
```

18.4.3 Alinear dos series con distinta resolución

```python
# Serie A: cada 10 min    Serie B: cada 1 hora
# Remuestrear B a 10 min para comparar punto a punto
df_B_10min = df_B.resample('10min').interpolate(method='linear')

# Alinear por índice de tiempo (inner join)
df_conjunto = df_A.join(df_B_10min, how='inner', rsuffix='_B')
```

18.5 Detección y eliminación de spikes

Un filtro de desviación estándar elimina valores atípicos aislados:

```python
def eliminar_spikes(serie, ventana=20, umbral_std=3.0):
    """
    Reemplaza por NaN los valores que se alejan más de umbral_std
    desviaciones estándar de la media móvil local.
    """
    media_local = serie.rolling(ventana, center=True, min_periods=5).mean()
    std_local   = serie.rolling(ventana, center=True, min_periods=5).std()
    mascara_spike = np.abs(serie - media_local) > umbral_std * std_local
    serie_limpia = serie.copy()
    serie_limpia[mascara_spike] = np.nan
    n_spikes = mascara_spike.sum()
    print(f"  Spikes eliminados: {n_spikes} ({n_spikes/len(serie)*100:.1f}%)")
    return serie_limpia

df['vel_limpia'] = eliminar_spikes(df['velocidad'])
```

18.6 Comparar señal original y filtrada

```
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

fig, axes = plt.subplots(2, 1, figsize=(14, 7), sharex=True)

axes[0].plot(df.index, df['velocidad'], color='lightsteelblue',
             linewidth=0.5, label='Original')
axes[0].plot(df.index[mascara_valida], vel_filtrada, color='navy',
             linewidth=1.0, label='Filtrada (PB 2h)')
axes[0].set_ylabel('Velocidad (m/s)')
axes[0].legend()

axes[1].plot(df.index, df['velocidad'] - df['vel_suavizada'],
            color='firebrick', linewidth=0.5, label='Residuo (original − suavizada)')
axes[1].axhline(0, color='gray', linewidth=0.5)
axes[1].set_ylabel('Residuo (m/s)')
axes[1].legend()

axes[1].xaxis.set_major_formatter(mdates.DateFormatter('%d %b'))
fig.tight_layout()
```

[>] filtfilt vs. lfilter — filtfilt aplica el filtro dos veces (ida y vuelta) para cancelar el desfase de fase. Esto es importante en oceanografía: un desfase introduciría un error temporal en los picos de marea o corriente. Usar siempre filtfilt salvo que se necesite causalidad estricta.

[!] NaN antes de filtrar — butter + filtfilt no toleran NaN en la serie. Siempre interpolar o eliminar los NaN antes de aplicar el filtro, y luego reponer los NaN en las posiciones originales si se quiere conservar la información de huecos.

19 python-docx

`python-docx` permite crear y modificar documentos Word (`.docx`) desde Python. Se usa para generar informes automáticamente: rellenar texto, insertar figuras, construir tablas y aplicar formato, todo a partir de una plantilla base.

19.1 Instalación

```
pip install python-docx
```

19.2 Estructura de un documento .docx

Un documento Word se organiza en **párrafos** (objetos `Paragraph`) que contienen **runs** (objetos `Run`). Cada run es un fragmento de texto con formato uniforme (fuente, negrita, tamaño, color). Entender esta jerarquía es clave para hacer reemplazos de texto sin perder el formato.

```
Document
  └── paragraphs[]
        └── runs[]    ← texto + formato
  └── tables[]
        └── rows[]
              └── cells[]
                    └── paragraphs[]
                          └── runs[]
```

19.3 Abrir y guardar

```
from docx import Document

# Abrir plantilla existente
doc = Document('plantilla_informe.docx')

# Guardar con nuevo nombre (no sobreescribir la plantilla)
doc.save('informe_final.docx')
```

19.4 Leer contenido

```
# Todos los párrafos
for p in doc.paragraphs:
    if p.text.strip():
        print(repr(p.text[:80]))

# Texto completo de una tabla
for tabla in doc.tables:
    for fila in tabla.rows:
```

```
        celdas = [c.text.strip() for c in fila.cells]
        print(celdas)
```

19.5 Reemplazar texto en párrafos

El método más directo, pero que puede romper el formato si un placeholder está dividido entre runs:

```
def reemplazar_en_parrafo(parrafo, viejo, nuevo):
    """Reemplaza texto en el texto completo del párrafo preservando los runs."""
    if viejo in parrafo.text:
        for run in parrafo.runs:
            if viejo in run.text:
                run.text = run.text.replace(viejo, nuevo)
```

Para placeholders que pueden quedar divididos entre runs (problema común con autoformato de Word), usar reemplazo a nivel de XML:

```
import re

def reemplazar_en_parrafo_robusto(parrafo, viejo, nuevo):
    """Reconstruye el texto del párrafo si el placeholder está partido."""
    texto_completo = parrafo.text
    if viejo not in texto_completo:
        return

    # Vaciar todos los runs excepto el primero
    for i, run in enumerate(parrafo.runs):
        if i == 0:
            run.text = texto_completo.replace(viejo, nuevo)
        else:
            run.text = ''
```

19.6 Insertar figuras

```
from docx.shared import Cm

def insertar_figura_en_placeholder(doc, placeholder, ruta_imagen, ancho_cm=14):
    """Reemplaza un placeholder de texto por una imagen."""
    for parrafo in doc.paragraphs:
        if placeholder in parrafo.text:
            parrafo.clear()
            run = parrafo.add_run()
            run.add_picture(str(ruta_imagen), width=Cm(ancho_cm))
            return True
    return False
```

```
insertar_figura_en_placeholder(doc, '[FIGURA_ROSA]',
↪  'figuras/rosa_corrientes_7m.png')
```

19.7 Construir tablas

```
from docx.shared import Pt, RGBColor
from docx.enum.text import WD_ALIGN_PARAGRAPH

def agregar_tabla_estadisticas(doc, df_stats, placeholder='[TABLA_STATS]'):
    """
    Inserta un DataFrame como tabla Word en la posición del placeholder.
    """
    for i, parrafo in enumerate(doc.paragraphs):
        if placeholder not in parrafo.text:
            continue

        parrafo.clear()

        # Crear tabla: 1 fila de encabezado + filas de datos
        ncols = len(df_stats.columns) + 1   # +1 para índice
        tabla = doc.add_table(rows=1, cols=ncols)
        tabla.style = 'Table Grid'

        # Encabezado
        fila_enc = tabla.rows[0]
        fila_enc.cells[0].text = df_stats.index.name or ''
        for j, col in enumerate(df_stats.columns):
            fila_enc.cells[j + 1].text = col

        # Datos
        for idx, fila_datos in df_stats.iterrows():
            fila = tabla.add_row()
            fila.cells[0].text = str(idx)
            for j, val in enumerate(fila_datos):
                fila.cells[j + 1].text = f'{val:.2f}' if isinstance(val, float) else
↪  str(val)

        # Mover tabla al lugar del párrafo
        parrafo._element.addnext(tabla._tbl)
        break
```

19.8 Aplicar formato a texto

```
from docx.shared import Pt, RGBColor
from docx.enum.text import WD_ALIGN_PARAGRAPH

# Agregar párrafo con formato
p = doc.add_paragraph()
run = p.add_run('Velocidad máxima registrada: ')
run.bold = True
run.font.size = Pt(11)

run2 = p.add_run('0.62 m/s')
run2.font.color.rgb = RGBColor(0x00, 0x5B, 0x96)
run2.bold = True
```

19.9 Iterar sobre cuerpo completo (párrafos + tablas)

Para aplicar reemplazos en todo el documento, incluyendo las celdas de las tablas:

```
def todos_los_parrafos(doc):
    """Generador que yields todos los párrafos del documento (cuerpo y tablas)."""
    yield from doc.paragraphs
    for tabla in doc.tables:
        for fila in tabla.rows:
            for celda in fila.cells:
                yield from celda.paragraphs

def reemplazar_en_doc(doc, reemplazos: dict):
    """
    Aplica un diccionario {placeholder: valor} en todo el documento.
    """
    for parrafo in todos_los_parrafos(doc):
        for viejo, nuevo in reemplazos.items():
            if viejo in parrafo.text:
                reemplazar_en_parrafo(parrafo, viejo, str(nuevo))
```

19.10 Flujo completo del autoinforme

```
from pathlib import Path

def generar_informe(ruta_plantilla, ruta_salida, reemplazos, figuras):
    """
    ruta_plantilla : Path a la plantilla .docx
    ruta_salida    : Path donde se guardará el informe
    reemplazos     : dict {placeholder_texto: valor}
    figuras        : dict {placeholder_figura: ruta_imagen}
    """
    doc = Document(ruta_plantilla)
```

```
    # 1. Reemplazar texto
    reemplazar_en_doc(doc, reemplazos)

    # 2. Insertar figuras
    for placeholder, ruta_img in figuras.items():
        ok = insertar_figura_en_placeholder(doc, placeholder, ruta_img)
        if not ok:
            print(f"  ! Placeholder no encontrado: {placeholder}")

    doc.save(ruta_salida)
    print(f"Informe guardado: {ruta_salida}")

# Uso
generar_informe(
    ruta_plantilla = Path('plantillas/corrientes_v3.docx'),
    ruta_salida    = Path('informes/Los_Vilos_Corrientes_2025.docx'),
    reemplazos = {
        '[PROYECTO]':   'Los Vilos — Campaña octubre 2025',
        '[VEL_MAX]':    '0.62 m/s',
        '[DIR_PRED]':   'NNO (337°)',
        '[FECHA_INI]':  '01/10/2025',
        '[FECHA_FIN]':  '31/10/2025',
    },
    figuras = {
        '[FIGURA_ROSA]':    'figuras/rosa_corrientes_7m.png',
        '[FIGURA_SERIE]':   'figuras/serie_velocidad.png',
        '[FIGURA_HEATMAP]': 'figuras/heatmap_velocidad.png',
    }
)
```

[!] Estilos y plantilla — Los estilos de Word (Título 1, Normal, Tabla Grid) están definidos en la plantilla. Si se crea un documento desde cero con Document(), los estilos por defecto de python-docx son distintos a los del template corporativo. Siempre trabajar sobre la plantilla para conservar los estilos visuales del informe.

20 Plantillas y placeholders

El autoinforme se basa en una plantilla Word preformateada (`.docx`) con marcadores de posición —**placeholders**— que Python reemplaza en tiempo de ejecución. Este enfoque separa el diseño visual (responsabilidad del Word) del contenido numérico (responsabilidad del script).

20.1 Convención de placeholders

Los placeholders se escriben en la plantilla entre corchetes, en mayúsculas y con guiones bajos:

```
[PROYECTO]         → nombre del proyecto
[VEL_MAX_7M]       → velocidad máxima en la capa de 7 m
[DIR_PRED_7M]      → dirección predominante en 7 m
[FIGURA_ROSA_7M]   → imagen de rosa de corrientes en 7 m
[TABLA_INCIDENCIA] → tabla de incidencia
```

[>] Nomenclatura consistente — Usar siempre el mismo formato en la plantilla y en el diccionario de reemplazos del script. Un error tipográfico en el placeholder deja el marcador sin reemplazar en el informe final, lo cual es fácil de detectar visualmente.

20.2 Validar que todos los placeholders se reemplazaron

Antes de guardar el informe, verificar que no quedó ningún placeholder sin llenar:

```
import re

def placeholders_pendientes(doc):
    """Retorna una lista de placeholders que no fueron reemplazados."""
    patron = re.compile(r'\[[A-Z_0-9]+\]')
    encontrados = set()
    for parrafo in todos_los_parrafos(doc):   # función del capítulo 14
        for match in patron.finditer(parrafo.text):
            encontrados.add(match.group())
    return sorted(encontrados)

pendientes = placeholders_pendientes(doc)
if pendientes:
    print(f"  ! Placeholders sin reemplazar: {pendientes}")
else:
    print("  ✓ Todos los placeholders reemplazados")
```

20.3 Placeholders en tablas

Los placeholders pueden estar dentro de celdas de tablas. La función `todos_los_parrafos` del capítulo anterior ya los cubre, pero a veces se necesita reemplazar una celda completa:

```python
def reemplazar_en_tablas(doc, reemplazos: dict):
    """Reemplaza placeholders dentro de todas las celdas de todas las tablas."""
    for tabla in doc.tables:
        for fila in tabla.rows:
            for celda in fila.cells:
                for parrafo in celda.paragraphs:
                    for viejo, nuevo in reemplazos.items():
                        if viejo in parrafo.text:
                            reemplazar_en_parrafo(parrafo, viejo, str(nuevo))
```

20.4 Placeholders que se repiten

Un mismo placeholder puede aparecer múltiples veces en el documento (p. ej. `[PROYECTO]` en el encabezado, en el cuerpo y en el pie de página). La función `reemplazar_en_doc` ya los reemplaza todos porque itera sobre todos los párrafos.

Para placeholders en **encabezados y pies de página**, hay que acceder explícitamente a las secciones:

```python
def reemplazar_en_encabezados_pies(doc, reemplazos: dict):
    for seccion in doc.sections:
        for parrafo in seccion.header.paragraphs:
            for viejo, nuevo in reemplazos.items():
                if viejo in parrafo.text:
                    reemplazar_en_parrafo(parrafo, viejo, str(nuevo))
        for parrafo in seccion.footer.paragraphs:
            for viejo, nuevo in reemplazos.items():
                if viejo in parrafo.text:
                    reemplazar_en_parrafo(parrafo, viejo, str(nuevo))
```

20.5 Placeholder partido entre runs

Word a veces divide un placeholder en runs separados cuando el usuario activa la corrección automática o cuando lo escribe carácter a carácter. Por ejemplo `[VEL_MAX]` puede quedar como:

```
Run 1: "[VEL_"
Run 2: "MAX]"
```

En ese caso `viejo in parrafo.text` detecta el placeholder, pero `viejo in run.text` no encuentra nada. La solución es fusionar los runs antes de reemplazar:

```python
def fusionar_runs_parrafo(parrafo):
    """Fusiona todos los runs del párrafo en el primero, preservando el formato del
    ↪ primero."""
    if not parrafo.runs:
        return
    texto_total = parrafo.text
    for i, run in enumerate(parrafo.runs):
        run.text = texto_total if i == 0 else ''

def reemplazar_robusto(doc, reemplazos: dict):
    """Fusiona runs y luego reemplaza. Usar solo cuando hay placeholders partidos."""
    patron = re.compile(r'\[[A-Z_0-9]+\]')
    for parrafo in todos_los_parrafos(doc):
        if patron.search(parrafo.text):
            fusionar_runs_parrafo(parrafo)
        for viejo, nuevo in reemplazos.items():
            if viejo in parrafo.text:
                reemplazar_en_parrafo(parrafo, viejo, str(nuevo))
```

[!] Fusionar runs borra el formato interno — Al fusionar todos los runs en uno solo, el texto queda con el formato del primer run (fuente, tamaño, negrita). Esto es aceptable si el placeholder ocupa su propio párrafo. Si el placeholder está en medio de un párrafo con texto mixto (parte en negrita, parte normal), la fusión puede romper el formato del párrafo completo.

20.6 Diccionario de reemplazos por campaña

En el pipeline se construye el diccionario de reemplazos a partir de los datos calculados:

```python
def construir_reemplazos(df_stats, meta):
    """
    df_stats : DataFrame con estadísticas por profundidad
    meta     : dict con metadatos del proyecto (fechas, nombre, etc.)
    """
    reemplazos = {
        '[PROYECTO]':   meta['nombre'],
        '[EMPRESA]':    meta['empresa'],
        '[FECHA_INI]':  meta['fecha_inicio'].strftime('%d/%m/%Y'),
        '[FECHA_FIN]':  meta['fecha_fin'].strftime('%d/%m/%Y'),
        '[N_DATOS]':    str(meta['n_datos']),
    }

    # Estadísticas por profundidad
    for prof in df_stats.index:
        fila = df_stats.loc[prof]
        clave = str(prof).replace('.', '_')   # 7.5 → 7_5
        reemplazos[f'[VEL_MEDIA_{clave}M]'] = f"{fila['vel_media']:.2f}"
```

```
        reemplazos[f'[VEL_MAX_{clave}M]']   = f"{fila['vel_max']:.2f}"
        reemplazos[f'[DIR_PRED_{clave}M]']  = f"{fila['dir_pred']:.0f}°"

    return reemplazos
```

20.7 Flujo recomendado

```
from pathlib import Path
from docx import Document

def generar_informe_completo(ruta_plantilla, ruta_salida, meta, df_stats, figuras):
    doc = Document(ruta_plantilla)

    # 1. Reemplazar texto en cuerpo
    reemplazos = construir_reemplazos(df_stats, meta)
    reemplazar_robusto(doc, reemplazos)

    # 2. Reemplazar en encabezados y pies
    reemplazar_en_encabezados_pies(doc, reemplazos)

    # 3. Insertar figuras
    for placeholder, ruta_img in figuras.items():
        insertar_figura_en_placeholder(doc, placeholder, ruta_img)

    # 4. Validar
    pendientes = placeholders_pendientes(doc)
    if pendientes:
        print(f"  ! Sin reemplazar: {pendientes}")

    doc.save(ruta_salida)
    print(f"Informe generado: {ruta_salida.name}")
```

21 Importación dinámica

El autoinforme es un script genérico que puede procesar cualquier campaña. La configuración específica de cada proyecto (rutas, parámetros, profundidades, nombre de empresa) vive en un módulo Python separado que se carga en tiempo de ejecución según el nombre del proyecto. Esto elimina la necesidad de editar el script central cada vez que se agrega una nueva campaña.

21.1 El problema: configuración por proyecto

Sin importación dinámica, el script tendría bloques `if/elif`:

```
# MAL: requiere modificar el script central para cada proyecto
if proyecto == 'los_vilos':
    from configs import los_vilos as cfg
elif proyecto == 'coquimbo':
    from configs import coquimbo as cfg
elif proyecto == 'iquique':
    from configs import iquique as cfg
# ... y así indefinidamente
```

Con importación dinámica, el script central no cambia nunca:

```
# BIEN: el script carga automáticamente el módulo correcto
import importlib
cfg = importlib.import_module(f'configs.{nombre_proyecto}')
```

21.2 importlib.import_module

```
import importlib

def cargar_config(nombre_proyecto):
    """
    Carga el módulo de configuración para el proyecto indicado.
    Lanza ImportError con mensaje claro si no existe.
    """
    nombre_modulo = f'configs.{nombre_proyecto}'
    try:
        modulo = importlib.import_module(nombre_modulo)
    except ModuleNotFoundError:
        raise FileNotFoundError(
            f"No existe configuración para '{nombre_proyecto}'. "
            f"Crear el archivo configs/{nombre_proyecto}.py"
        )
    return modulo

# Uso
```

```
cfg = cargar_config('los_vilos_oct2025')

ruta_datos = cfg.RUTA_DATOS
profundidades = cfg.PROFUNDIDADES
empresa = cfg.EMPRESA
```

21.3 Estructura de un módulo de configuración

Cada proyecto tiene un archivo en `configs/`:

```
autoinforme/
  configs/
    __init__.py          ← vacío, hace de configs un paquete
    los_vilos_oct2025.py
    coquimbo_mar2025.py
    iquique_ene2026.py
  autoinforme.py         ← script central (nunca se modifica)
```

Un módulo de configuración típico:

```
# configs/los_vilos_oct2025.py
from pathlib import Path

PROYECTO     = 'Los Vilos — Campaña octubre 2025'
EMPRESA      = 'Puerto Los Vilos S.A.'
FECHA_INICIO = '2025-10-01'
FECHA_FIN    = '2025-10-31'

RUTA_BASE    = Path('/mnt/c/Users/Usuario/proyectos/los_vilos_2025')
RUTA_DATOS   = RUTA_BASE / 'datos' / 'corrientes_procesadas.xlsx'
RUTA_FIGURAS = RUTA_BASE / 'figuras_magnitud'
RUTA_SALIDA  = RUTA_BASE / 'informe' / 'Los_Vilos_Corrientes_Oct2025.docx'
PLANTILLA    = Path('plantillas') / 'corrientes_v3.docx'

PROFUNDIDADES = [3, 5, 7, 9, 11, 13, 15]
BINS_VEL      = [0, 0.05, 0.1, 0.2, float('inf')]
```

21.4 Verificar que el módulo tiene los atributos necesarios

```
ATRIBUTOS_REQUERIDOS = [
    'PROYECTO', 'EMPRESA', 'FECHA_INICIO', 'FECHA_FIN',
    'RUTA_DATOS', 'RUTA_FIGURAS', 'RUTA_SALIDA', 'PLANTILLA',
    'PROFUNDIDADES',
]

def validar_config(cfg, nombre_proyecto):
    faltantes = [attr for attr in ATRIBUTOS_REQUERIDOS if not hasattr(cfg, attr)]
```

```
    if faltantes:
        raise AttributeError(
            f"Configuración '{nombre_proyecto}' incompleta. "
            f"Faltan: {faltantes}"
        )
```

21.5 Recargar un módulo modificado

Durante el desarrollo, si se edita el archivo de configuración con el script ya corriendo en Spyder, hay que recargar el módulo para ver los cambios:

```
import importlib

cfg = importlib.import_module('configs.los_vilos_oct2025')

# Después de editar el archivo:
importlib.reload(cfg)
```

21.6 Listar proyectos disponibles

```
from pathlib import Path

def listar_proyectos(carpeta_configs='configs'):
    carpeta = Path(carpeta_configs)
    proyectos = [
        p.stem for p in carpeta.glob('*.py')
        if p.stem != '__init__'
    ]
    return sorted(proyectos)

print("Proyectos disponibles:")
for p in listar_proyectos():
    print(f"  {p}")
```

21.7 Patrón del script central

El script `autoinforme.py` recibe el nombre del proyecto por argumento de línea de comandos o por input interactivo:

```
import sys
import importlib

def main():
    if len(sys.argv) > 1:
        nombre_proyecto = sys.argv[1]
    else:
```

```
        proyectos = listar_proyectos()
        print("Proyectos disponibles:")
        for i, p in enumerate(proyectos):
            print(f"  [{i}] {p}")
        idx = int(input("Seleccionar: "))
        nombre_proyecto = proyectos[idx]

    cfg = cargar_config(nombre_proyecto)
    validar_config(cfg, nombre_proyecto)

    print(f"\nProcesando: {cfg.PROYECTO}")

    # El resto del pipeline usa cfg.RUTA_DATOS, cfg.PROFUNDIDADES, etc.
    datos = cargar_datos(cfg.RUTA_DATOS)
    stats = calcular_estadisticas(datos, cfg.PROFUNDIDADES)
    generar_figuras(datos, stats, cfg.RUTA_FIGURAS)
    generar_informe(cfg.PLANTILLA, cfg.RUTA_SALIDA, cfg, stats)

if __name__ == '__main__':
    main()
```

[>] Alternativa con YAML — Si la configuración es puramente datos (sin expresiones Python ni herencia entre proyectos), se puede guardar como YAML y cargar con yaml.safe_load (ver capítulo 07). La ventaja del módulo .py es que permite expresiones, lógica condicional y paths calculados con Path(**file**).parent. La ventaja del YAML es que cualquiera puede editarlo sin saber Python.

22 Rasterio y GeoPandas

En el contexto oceanográfico, los datos geoespaciales aparecen en dos formas: **rasters** (grillas de valores, como batimetría, temperatura SST o campos de viento en grilla) y **vectores** (puntos, líneas y polígonos, como estaciones de medición, líneas de costa o polígonos de áreas de estudio). `rasterio` maneja rasters; `geopandas` maneja vectores.

22.1 Instalación

```
pip install rasterio geopandas
```

22.2 Leer un raster con rasterio

Un raster GeoTIFF contiene una o varias bandas de valores sobre una grilla georreferenciada:

```
import rasterio
import numpy as np

with rasterio.open('batimetria_zona.tif') as src:
    # Metadatos
    print(f"CRS:        {src.crs}")          # sistema de coordenadas
    print(f"Dimensiones: {src.width} × {src.height}")
    print(f"Bandas:     {src.count}")
    print(f"Bounding box: {src.bounds}")
    print(f"Resolución:  {src.res}")          # (dx, dy) en unidades del CRS

    # Leer banda 1
    datos = src.read(1).astype(float)         # array 2D (filas × columnas)
    datos[datos == src.nodata] = np.nan       # convertir nodata a NaN

    # Coordenadas del centroide de cada celda
    transform = src.transform
```

22.3 Obtener coordenadas de la grilla

```
import rasterio
from rasterio.transform import xy

with rasterio.open('batimetria_zona.tif') as src:
    datos = src.read(1).astype(float)
    filas, cols = np.indices(datos.shape)
    lons, lats = xy(src.transform, filas, cols)
    lons = np.array(lons)
    lats = np.array(lats)
```

22.4 Extraer el valor en un punto

Para extraer el valor del raster en una coordenada (lat, lon):

```
from rasterio.sample import sample_gen

def extraer_valor(raster_path, lon, lat):
    """Extrae el valor del raster en el punto (lon, lat)."""
    with rasterio.open(raster_path) as src:
        valores = list(src.sample([(lon, lat)]))
    return valores[0][0]

# Profundidad en la ubicación de una estación
prof = extraer_valor('batimetria.tif', lon=-71.52, lat=-29.90)
print(f"Profundidad: {prof:.1f} m")
```

22.5 Leer un shapefile con GeoPandas

```
import geopandas as gpd

# Línea de costa de Chile
costa = gpd.read_file('costa_chile.shp')
print(costa.crs)
print(costa.head())

# Filtrar por atributo
costa_4ta = costa[costa['region'] == 'Coquimbo']
```

22.6 Crear un GeoDataFrame desde coordenadas

```
import pandas as pd
import geopandas as gpd
from shapely.geometry import Point

# Estaciones de medición
estaciones = pd.DataFrame({
    'nombre': ['Los Vilos', 'Coquimbo', 'La Serena'],
    'lon':    [-71.52,      -71.35,     -71.25],
    'lat':    [-31.90,      -29.96,     -29.91],
    'vel_media': [0.12, 0.09, 0.14]
})

gdf_estaciones = gpd.GeoDataFrame(
    estaciones,
    geometry=gpd.points_from_xy(estaciones.lon, estaciones.lat),
    crs='EPSG:4326'   # WGS84 geográfico
```

```
)
```

22.7 Reproyectar

```
# De WGS84 (EPSG:4326) a UTM zona 19S (EPSG:32719)
gdf_utm = gdf_estaciones.to_crs('EPSG:32719')
print(gdf_utm.geometry[0])   # coordenadas en metros
```

22.8 Operaciones espaciales básicas

```
# Buffer de 5 km alrededor de cada estación
gdf_buffer = gdf_utm.copy()
gdf_buffer['geometry'] = gdf_utm.geometry.buffer(5000)   # metros

# Intersección: estaciones dentro de un polígono de estudio
area_estudio = gpd.read_file('area_estudio.shp').to_crs('EPSG:32719')
estaciones_en_area = gpd.sjoin(gdf_utm, area_estudio, how='inner',
 ↪  predicate='within')

# Distancia más cercana entre puntos y costa
gdf_utm['dist_costa_m'] = gdf_utm.geometry.apply(
    lambda p: costa.to_crs('EPSG:32719').distance(p).min()
)
```

22.9 Exportar resultados

```
# Guardar como shapefile
gdf_estaciones.to_file('estaciones_resultado.shp')

# Guardar como GeoJSON (más portable, sin limitación de nombre de columna)
gdf_estaciones.to_file('estaciones_resultado.geojson', driver='GeoJSON')

# Guardar raster modificado
with rasterio.open('batimetria.tif') as src:
    meta = src.meta.copy()
    datos = src.read(1).astype(float)

datos_suavizado = datos.copy()   # ... algún procesamiento

meta.update(dtype='float32')
with rasterio.open('batimetria_suavizada.tif', 'w', **meta) as dst:
    dst.write(datos_suavizado.astype('float32'), 1)
```

22.10 Estadísticas zonales

Calcular estadísticas del raster dentro de cada polígono:

```
from rasterstats import zonal_stats

# Profundidad media dentro de cada área de estudio
stats = zonal_stats(
    'areas_estudio.shp',
    'batimetria.tif',
    stats=['mean', 'min', 'max', 'std'],
    nodata=-9999
)

for area, s in zip(areas_estudio.itertuples(), stats):
    print(f"{area.nombre}: prof_media = {s['mean']:.1f} m")
```

[>] Sistema de coordenadas — Antes de cualquier operación espacial (buffer, distancias, áreas), reproyectar todos los datos a un CRS métrico como UTM. Los cálculos de distancia en grados (WGS84) son incorrectos y varían con la latitud. Para Chile central usar EPSG:32719 (UTM zona 19 Sur).

[!] Archivos shapefile — Un shapefile en realidad son varios archivos (.shp, .shx, .dbf, .prj). Al copiar o mover shapefiles, mover todos los archivos con el mismo nombre base. La alternativa moderna y más portable es GeoPackage (.gpkg) o GeoJSON.

23 Coordenadas y mapas

La visualización geoespacial en el pipeline incluye dos tareas: convertir coordenadas entre sistemas de referencia (UTM ↔ geográfico) y generar mapas de contexto para el informe. Los mapas sitúan la zona de estudio, marcan las estaciones y muestran resultados con color codificado por variable.

23.1 Conversión UTM ↔ lat/lon con pyproj

```
from pyproj import Transformer

# Definir transformador WGS84 ↔ UTM 19S
wgs84_a_utm  = Transformer.from_crs('EPSG:4326', 'EPSG:32719', always_xy=True)
utm_a_wgs84  = Transformer.from_crs('EPSG:32719', 'EPSG:4326', always_xy=True)

# Convertir un punto geográfico a UTM
lon, lat = -71.52, -31.90
este, norte = wgs84_a_utm.transform(lon, lat)
print(f"Este: {este:.1f} m, Norte: {norte:.1f} m")

# Convertir de vuelta
lon2, lat2 = utm_a_wgs84.transform(este, norte)
print(f"Lon: {lon2:.6f}°, Lat: {lat2:.6f}°")
```

23.1.1 Convertir un DataFrame de coordenadas

```
import pandas as pd
from pyproj import Transformer

def utm_a_geo(df, col_este='Este', col_norte='Norte', zona_utm='EPSG:32719'):
    """Agrega columnas Lon y Lat a un DataFrame con coordenadas UTM."""
    t = Transformer.from_crs(zona_utm, 'EPSG:4326', always_xy=True)
    lons, lats = t.transform(df[col_este].values, df[col_norte].values)
    df = df.copy()
    df['Lon'] = lons
    df['Lat'] = lats
    return df

df_estaciones = utm_a_geo(df_estaciones)
```

23.2 Mapa de contexto con GeoPandas y Matplotlib

```
import geopandas as gpd
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.colors as mcolors
import numpy as np

def mapa_estaciones(gdf_estaciones, col_valor, titulo='',
                    ruta_costa='costa_chile.shp', ruta_salida=None):
    """
    Genera un mapa con la línea de costa y las estaciones
    coloreadas por col_valor.
    """
    costa = gpd.read_file(ruta_costa)

    fig, ax = plt.subplots(figsize=(8, 10))

    # Línea de costa
    costa.plot(ax=ax, color='darkgreen', linewidth=0.5, alpha=0.7)

    # Estaciones coloreadas por el valor de interés
    vmin = gdf_estaciones[col_valor].min()
    vmax = gdf_estaciones[col_valor].max()
    norm = mcolors.Normalize(vmin=vmin, vmax=vmax)
    cmap = cm.YlOrRd

    sc = ax.scatter(
        gdf_estaciones.geometry.x,
        gdf_estaciones.geometry.y,
        c=gdf_estaciones[col_valor],
        cmap=cmap, norm=norm,
        s=80, zorder=5, edgecolors='black', linewidths=0.5
    )

    # Etiquetas de estación
    for _, row in gdf_estaciones.iterrows():
        ax.annotate(
            row['nombre'],
            xy=(row.geometry.x, row.geometry.y),
            xytext=(5, 5), textcoords='offset points',
            fontsize=8
        )

    plt.colorbar(sc, ax=ax, label=col_valor, shrink=0.6)

    ax.set_xlabel('Longitud (°)')
    ax.set_ylabel('Latitud (°)')
```

```
    ax.set_title(titulo)
    ax.grid(True, linestyle='--', alpha=0.4)
    fig.tight_layout()

    if ruta_salida:
        fig.savefig(ruta_salida, dpi=150, bbox_inches='tight')
        plt.close(fig)

    return fig, ax
```

23.3 Mapa de fondo con contextily (imágenes de satélite/OpenStreetMap)

```
import contextily as ctx

fig, ax = plt.subplots(figsize=(9, 9))

# Graficar estaciones (en WebMercator EPSG:3857)
gdf_web = gdf_estaciones.to_crs('EPSG:3857')
gdf_web.plot(ax=ax, color='red', markersize=60, zorder=4)

# Agregar mapa base desde internet
ctx.add_basemap(ax, source=ctx.providers.CartoDB.Positron, zoom=10)

ax.set_axis_off()
ax.set_title('Zona de estudio')
```

[!] contextily requiere internet — contextily descarga las teselas del mapa desde servidores externos. En equipos sin conexión usar el shapefile de costa en su lugar.

23.4 Grilla de campo vectorial (corrientes)

Para visualizar un campo de corrientes interpolado en una grilla:

```
from scipy.interpolate import griddata

# Puntos de medición dispersos
lons = gdf_estaciones.geometry.x.values
lats = gdf_estaciones.geometry.y.values
u_vals = gdf_estaciones['u_media'].values   # componente E-O
v_vals = gdf_estaciones['v_media'].values   # componente N-S

# Grilla regular de interpolación
lon_grid = np.linspace(lons.min() - 0.1, lons.max() + 0.1, 30)
lat_grid = np.linspace(lats.min() - 0.1, lats.max() + 0.1, 30)
LON, LAT = np.meshgrid(lon_grid, lat_grid)

U = griddata((lons, lats), u_vals, (LON, LAT), method='linear')
```

```
V = griddata((lons, lats), v_vals, (LON, LAT), method='linear')

fig, ax = plt.subplots(figsize=(9, 9))
ax.quiver(LON, LAT, U, V, np.sqrt(U**2 + V**2),
          cmap='YlOrRd', scale=5, width=0.003, alpha=0.8)
ax.scatter(lons, lats, c='black', s=20, zorder=5)
ax.set_xlabel('Longitud (°)')
ax.set_ylabel('Latitud (°)')
ax.set_title('Campo de corriente media superficial')
ax.set_aspect('equal')
```

23.5 Recortar raster a área de estudio

```
import rasterio
from rasterio.mask import mask
import geopandas as gpd
from shapely.geometry import mapping

area = gpd.read_file('area_estudio.shp').to_crs('EPSG:4326')

with rasterio.open('batimetria_nacional.tif') as src:
    out_image, out_transform = mask(
        src,
        [mapping(geom) for geom in area.geometry],
        crop=True,
        nodata=np.nan
    )
    out_meta = src.meta.copy()

out_meta.update({
    'driver': 'GTiff',
    'height': out_image.shape[1],
    'width':  out_image.shape[2],
    'transform': out_transform
})

with rasterio.open('batimetria_zona.tif', 'w', **out_meta) as dst:
    dst.write(out_image)
```

23.6 Calcular distancia a la costa

```
from pyproj import Geod

geod = Geod(ellps='WGS84')

def distancia_a_la_costa_km(lon_estacion, lat_estacion, gdf_costa):
    """
    Calcula la distancia mínima en km de un punto a la línea de costa.
    """
    gdf_utm = gdf_costa.to_crs('EPSG:32719')
    from shapely.geometry import Point
    punto_utm = gpd.GeoDataFrame(geometry=[Point(lon_estacion, lat_estacion)],
                                 crs='EPSG:4326').to_crs('EPSG:32719').geometry[0]
    dist_m = gdf_utm.geometry.distance(punto_utm).min()
    return dist_m / 1000

for _, row in gdf_estaciones.iterrows():
    d = distancia_a_la_costa_km(row.geometry.x, row.geometry.y, costa)
    print(f"{row['nombre']}: {d:.1f} km de la costa")
```

[>] Selección del sistema de coordenadas — Para Chile, UTM zona 19 Sur (EPSG:32719) cubre desde 66°O hasta 60°O, incluyendo toda la costa continental. La zona 18 Sur (EPSG:32718) cubre desde 72°O hasta 66°O y es necesaria para el extremo sur y la región de Los Lagos. Verificar siempre que los datos de entrada estén en el CRS correcto antes de operar.

24 OCR de cartas batimétricas

Las cartas batimétricas —tanto cartas náuticas escaneadas (SHOA) como mosaicos de cartografía digital (Navionics)— contienen cientos de sondeos de profundidad representados como números dispersos sobre la imagen. Digitalizarlos manualmente implica hacer clic en cada número y escribir el valor. Con OpenCV y Tesseract el proceso se automatiza: Python detecta los números, los lee y los georeferencia en minutos.

24.1 Dependencias

```
pip install opencv-python pytesseract pillow rasterio
# Tesseract (Windows): https://github.com/UB-Mannheim/tesseract/wiki
# Linux/WSL: sudo apt install tesseract-ocr
```

24.2 Flujo general del pipeline

```
imagen (JPG/TIFF/GeoTIFF)
    ↓ preprocesamiento (umbralización, máscara de tierra)
    ↓ detección de regiones candidatas (contornos)
    ↓ Tesseract OCR sobre cada región
    ↓ filtro de valores plausibles
    ↓ georeferencia (GCP lineal o GeoTIFF)
    ↓ exportar CSV / XYZ
```

24.3 Cargar y preprocesar la imagen

```
import cv2
import numpy as np

# cv2.imdecode con fromfile maneja rutas con tildes y espacios en Windows
img_bgr  = cv2.imdecode(np.fromfile('carta_shoa.tiff', dtype=np.uint8),
↪  cv2.IMREAD_COLOR)
img_gray = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2GRAY)
img_rgb  = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)

h_px, w_px = img_gray.shape
print(f'Imagen: {w_px} × {h_px} px')
```

24.3.1 Umbralización adaptativa (cartas B&N — SHOA)

Las cartas SHOA son blanco y negro con gradientes de iluminación. La umbralización adaptativa compensa variaciones de brillo locales:

```
# Binarizar: texto negro sobre fondo blanco
img_bin = cv2.adaptiveThreshold(
    img_gray, 255,
    cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
    cv2.THRESH_BINARY,
    blockSize=15,   # tamaño del vecindario local
    C=5             # constante restada a la media local
)

# Escalar × 3 para que Tesseract trabaje con números más grandes
ESCALA = 3
img_up = cv2.resize(img_bin, None, fx=ESCALA, fy=ESCALA,
                    interpolation=cv2.INTER_CUBIC)
```

24.3.2 Máscara de tierra por color HSV (cartas Navionics)

Las cartas Navionics usan color: tierra = amarillo/verde, agua = azul. Enmascarar la tierra evita que OCR confunda textos de topografía con sondeos marinos:

```
img_hsv = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2HSV)
H, S = img_hsv[:, :, 0], img_hsv[:, :, 1]

# Rangos en OpenCV: H está en [0, 179] (la mitad del ángulo estándar)
mask_tierra = (
    ((H >= 10) & (H <= 24) & (S > 45)) |   # amarillo tierra
    ((H >= 33) & (H <= 60) & (S > 35))      # verde intermareal
)

# Dilatar 2 iteraciones para cubrir bordes y píxeles de borde
kernel = np.ones((5, 5), np.uint8)
mask_tierra = cv2.dilate(mask_tierra.astype(np.uint8), kernel,
 ↪  iterations=2).astype(bool)

# Píxeles oscuros en zona de agua = candidatos a texto
UMBRAL_GRIS = 80
mask_texto = (img_gray < UMBRAL_GRIS) & (~mask_tierra)
mask_u8    = mask_texto.astype(np.uint8) * 255

# Eliminar ruido de compresión JPEG
mask_u8 = cv2.morphologyEx(mask_u8, cv2.MORPH_OPEN, np.ones((2, 2), np.uint8))
```

24.4 Detectar regiones candidatas

Cada número es un grupo de píxeles conectados. La dilatación horizontal conecta dígitos del mismo número antes de buscar contornos:

```
# Conectar dígitos del mismo número horizontalmente
ANCHO_MAX_PX = 90   # hasta 4 dígitos
kw = max(6, int(ANCHO_MAX_PX * 0.55))
kernel_h = cv2.getStructuringElement(cv2.MORPH_RECT, (kw, 1))
mask_groups = cv2.dilate(mask_u8, kernel_h)

contours, _ = cv2.findContours(mask_groups, cv2.RETR_EXTERNAL,
↪  cv2.CHAIN_APPROX_SIMPLE)

# Filtrar por tamaño esperado de un número (ajustar según zoom)
ALTO_MIN, ALTO_MAX = 6, 28
ANCHO_MIN, ANCHO_MAX = 4, 90

candidatos = []
for cnt in contours:
    x, y, w, h = cv2.boundingRect(cnt)
    if not (ALTO_MIN <= h <= ALTO_MAX and ANCHO_MIN <= w <= ANCHO_MAX):
        continue
    # Densidad mínima de píxeles oscuros (descartar cajas vacías)
    if mask_u8[y:y+h, x:x+w].mean() < 8:
        continue
    candidatos.append((x, y, w, h))

print(f'Candidatos a número: {len(candidatos)}')
```

24.5 OCR con Tesseract

```
import pytesseract
import re

# Windows: ajustar ruta al ejecutable de Tesseract
pytesseract.pytesseract.tesseract_cmd = r'C:\Program
↪  Files\Tesseract-OCR\tesseract.exe'

# Configuración: modo página 8 (un solo bloque de texto), solo dígitos y punto
config_tess = '--psm 8 --oem 3 -c tessedit_char_whitelist=0123456789.'
pat_num     = re.compile(r'^\d{1,4}(\.\d{1,2})?$')

PROF_MIN, PROF_MAX = 0.5, 9999.0
CONF_MINIMA = 45
ESCALA_OCR  = 4   # ampliar imagen para OCR
PAD = 5           # píxeles de margen alrededor del candidato

puntos_ocr = []

# Imagen binaria fondo blanco / texto negro para Tesseract
```

```
img_bin_ocr = np.where(mask_u8[:, :, np.newaxis] > 0,
                       np.zeros_like(img_rgb),
                       np.full_like(img_rgb, 255)).astype(np.uint8)

for x, y, w, h in candidatos:
    # Recortar con margen
    x1, y1 = max(0, x - PAD), max(0, y - PAD)
    x2, y2 = min(w_px, x+w+PAD), min(h_px, y+h+PAD)
    roi = img_bin_ocr[y1:y2, x1:x2]

    # Ampliar para OCR
    roi_up = cv2.resize(roi, None, fx=ESCALA_OCR, fy=ESCALA_OCR,
                        interpolation=cv2.INTER_NEAREST)

    datos = pytesseract.image_to_data(roi_up, config=config_tess,
                                      output_type=pytesseract.Output.DICT)

    for j, txt in enumerate(datos['text']):
        txt = str(txt).strip()
        if not pat_num.match(txt):
            continue
        try:
            val = float(txt)
        except ValueError:
            continue
        if not (PROF_MIN <= val <= PROF_MAX):
            continue
        if int(datos['conf'][j]) < CONF_MINIMA:
            continue

        # Centro del bbox en coordenadas de imagen original
        puntos_ocr.append({
            'col':     x + w / 2.0,
            'row':     y + h / 2.0,
            'depth_m': val,
            'conf':    int(datos['conf'][j])
        })
        break   # un número por región

print(f'Sondeos detectados: {len(puntos_ocr)}')
```

24.5.1 Modos PSM de Tesseract

PSM	Cuándo usar
8	Una sola palabra/número por recorte (recomendado para candidatos ya recortados)
11	Texto disperso sobre toda la imagen (útil para OCR sobre imagen completa)
6	Un bloque de texto uniforme

24.6 Deduplicar sondeos solapados

```
# Conservar el de mayor confianza entre puntos a menos de 15 px de distancia
pts = sorted(puntos_ocr, key=lambda p: -p['conf'])
pts_unicos = []
for p in pts:
    if all(np.hypot(p['row'] - u['row'], p['col'] - u['col']) > 15
           for u in pts_unicos):
        pts_unicos.append(p)

print(f'Tras deduplicar: {len(pts_unicos)} sondeos únicos')
```

24.7 Georeferencia

24.7.1 Opción A — GeoTIFF (automática con rasterio)

Si la imagen ya viene georeferenciada (p. ej. GeoTIFF con EPSG:4326):

```
import rasterio
from rasterio.transform import xy as rxy

with rasterio.open('navionics_area.tif') as src:
    tf = src.transform

def px_a_geo(col, row):
    lon, lat = rxy(tf, row, col)
    return float(lon), float(lat)
```

24.7.2 Opción B — GCPs manuales (cartas escaneadas SHOA)

Cuando la imagen no tiene georreferencia embebida, se usan puntos de control (GCPs): pares (fila_px, col_px) → (lon, lat) de referencias conocidas en la carta:

```
import numpy as np

# GCPs: (fila_px, col_px, lon, lat)
GCPS = [
    ( 100,  200, -72.600, -41.500),   # esquina superior izquierda
    ( 100, 3800, -71.900, -41.500),   # esquina superior derecha
    (2900,  200, -72.600, -42.200),   # esquina inferior izquierda
    (2900, 3800, -71.900, -42.200),   # esquina inferior derecha
]

gcps = np.array(GCPS, dtype=float)
A    = np.column_stack([gcps[:, 1], gcps[:, 0], np.ones(len(gcps))])

coef_lon, _, _, _ = np.linalg.lstsq(A, gcps[:, 2], rcond=None)
coef_lat, _, _, _ = np.linalg.lstsq(A, gcps[:, 3], rcond=None)

def px_a_geo(col, row):
    v = np.array([col, row, 1.0])
    return float(coef_lon @ v), float(coef_lat @ v)

# Verificar error en los propios GCPs
for fila, col, lon_r, lat_r in GCPS:
    lon_e, lat_e = px_a_geo(col, fila)
    err_m = np.hypot((lon_e - lon_r) * 111000, (lat_e - lat_r) * 111000)
    print(f'  Error en GCP: {err_m:.0f} m')
```

[>] Precisión con GCPs — Con 4 GCPs en las esquinas el error típico es 50–150 m. Agregar referencias internas de la retícula (intersecciones de meridianos y paralelos impresos en la carta) baja el error a 10–30 m. Con 8+ GCPs bien distribuidos la transformación lineal converge a la precisión de escaneo.

24.8 Exportar a CSV y XYZ

```
import pandas as pd

registros = []
for p in pts_unicos:
    lon, lat = px_a_geo(p['col'], p['row'])
    registros.append({'lon': lon, 'lat': lat, 'depth_m': p['depth_m']})

df = pd.DataFrame(registros).sort_values('depth_m').reset_index(drop=True)

# CSV para QGIS
df.to_csv('batimetria.csv', index=False, float_format='%.6f')

# XYZ (lon lat profundidad) para Blue Kenue / SWAN
df[['lon', 'lat', 'depth_m']].to_csv(
```

```
    'batimetria.xyz', sep=' ', index=False, header=False, float_format='%.6f')

print(f'Exportados: {len(df)} sondeos')
print(f'Rango: {df["depth_m"].min():.1f} – {df["depth_m"].max():.1f} m')
```

24.9 Visualizar resultado

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(9, 7))
sc = ax.scatter(df['lon'], df['lat'],
                c=df['depth_m'], cmap='Blues_r',
                s=15, edgecolors='k', linewidths=0.2, vmin=0)
plt.colorbar(sc, ax=ax, label='Profundidad (m)')
ax.set_xlabel('Longitud (°)')
ax.set_ylabel('Latitud (°)')
ax.set_title(f'Batimetría digitalizada – {len(df)} puntos')
ax.set_aspect('equal')
plt.tight_layout()
plt.savefig('batimetria_mapa.png', dpi=150, bbox_inches='tight')
plt.show()
```

[!] Unidades en cartas SHOA — Las cartas SHOA antiguas pueden estar en pies o brazas, no en metros. Verificar en el margen de la carta. Para convertir: 1 pie = 0.3048 m, 1 braza = 1.8288 m.

[>] Guardar sesión para no repetir OCR — El OCR sobre una imagen grande puede tardar varios minutos. Guardar los resultados en JSON permite retomarlos sin reprocesar: python import json with open(‘sesion.json’, ‘w’) as f: json.dump(pts_unicos, f) # Cargar al reiniciar: # with open(‘sesion.json’) as f: pts_unicos = json.load(f)

25 Outputs de CROCO-ROMS

CROCO (Coastal and Regional Ocean COmmunity model) es un modelo oceánico regional que simula temperatura, salinidad y corrientes en un dominio definido por el usuario. Sus archivos de salida están en formato **NetCDF4**: un estándar científico para almacenar datos multidimensionales con coordenadas etiquetadas (tiempo, profundidad, latitud, longitud).

La herramienta estándar para trabajar con estos archivos en Python es **xarray**, que carga los datos de forma diferida: lee la estructura del archivo sin traer todos los valores a memoria, y accede a ellos solo cuando se necesitan.

25.1 Instalación

```
pip install xarray netcdf4 matplotlib cartopy scipy
```

25.2 Abrir un archivo y explorar su estructura

```
import xarray as xr

ds = xr.open_dataset('croco_his.nc')
print(ds)
```

La salida muestra dimensiones, variables y atributos globales:

```
Dimensions:     (ocean_time: 365, s_rho: 20, eta_rho: 150, xi_rho: 200)
Data variables:
    temp     (ocean_time, s_rho, eta_rho, xi_rho) float32
    salt     (ocean_time, s_rho, eta_rho, xi_rho) float32
    u        (ocean_time, s_rho, eta_u, xi_u)     float32
    v        (ocean_time, s_rho, eta_v, xi_v)     float32
    zeta     (ocean_time, eta_rho, xi_rho)        float32
    lon_rho  (eta_rho, xi_rho)                    float64
    lat_rho  (eta_rho, xi_rho)                    float64
    h        (eta_rho, xi_rho)                    float64
```

Variable	Descripción
`temp`	Temperatura potencial (°C)
`salt`	Salinidad (PSU)
`u`, `v`	Corrientes E-O y N-S (m/s) en grillas desplazadas
`zeta`	Elevación de la superficie libre (m)
`h`	Batimetría — profundidad del fondo (m)
`lon_rho`, `lat_rho`	Coordenadas de cada celda de la grilla (2D)

25.3 Coordenadas verticales sigma

CROCO no usa profundidades fijas: usa **coordenadas sigma** (σ), que siguen la forma del fondo oceánico. En superficie σ = 0 y en el fondo σ = -1. Los N niveles se distribuyen entre estos extremos, con mayor resolución donde el modelo lo requiera (cerca de la superficie o del fondo).

La dimensión `s_rho` indexa estos niveles: `isel(s_rho=-1)` es el nivel de **superficie** y `isel(s_rho=0)` es el más cercano al **fondo**.

25.4 Mapa de temperatura superficial

```
import numpy as np
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
import cartopy.feature as cfeature

sst = ds['temp'].isel(ocean_time=0, s_rho=-1).values   # array 2D
lon = ds['lon_rho'].values
lat = ds['lat_rho'].values

fig, ax = plt.subplots(figsize=(9, 7),
                       subplot_kw={'projection': ccrs.PlateCarree()})
ax.add_feature(cfeature.COASTLINE, linewidth=0.8)
ax.add_feature(cfeature.LAND, facecolor='lightgray')

pc = ax.pcolormesh(lon, lat, sst, cmap='RdYlBu_r', vmin=8, vmax=20,
                   transform=ccrs.PlateCarree())
plt.colorbar(pc, ax=ax, label='Temperatura (°C)')
ax.set_title('SST — CROCO')
plt.tight_layout()
plt.savefig('sst.png', dpi=150)
```

25.5 Serie temporal en un punto

Para extraer la evolución temporal de una variable en la celda más cercana a una coordenada:

```
import numpy as np

def celda_mas_cercana(ds, lon_target, lat_target):
    """Retorna los índices (eta, xi) de la celda más cercana al punto."""
    lon = ds['lon_rho'].values
    lat = ds['lat_rho'].values
    dist = np.sqrt((lon - lon_target)**2 + (lat - lat_target)**2)
    eta_idx, xi_idx = np.unravel_index(dist.argmin(), dist.shape)
    return int(eta_idx), int(xi_idx)
```

```
eta_i, xi_i = celda_mas_cercana(ds, lon_target=-72.5, lat_target=-41.5)

sst_serie = ds['temp'].isel(s_rho=-1, eta_rho=eta_i, xi_rho=xi_i).values
tiempo    = ds['ocean_time'].values

plt.figure(figsize=(11, 3))
plt.plot(tiempo, sst_serie, lw=0.8, color='steelblue')
plt.ylabel('Temperatura (°C)')
plt.title(f'SST en ({ds["lon_rho"].values[eta_i, xi_i]:.2f}°, '
          f'{ds["lat_rho"].values[eta_i, xi_i]:.2f}°)')
plt.tight_layout()
```

25.6 Perfil vertical: convertir sigma a metros

Para graficar una variable en función de la profundidad real es necesario convertir los niveles sigma. La conversión usa la batimetría local (`h`), la superficie libre (`zeta`) y los parámetros del modelo:

```
def sigma_a_profundidad(h, zeta, theta_s, theta_b, hc, N, vtransform=2):
    """
    Calcula la profundidad en metros de cada nivel sigma en un punto.
    Retorna array de longitud N (valores negativos bajo la superficie).
    """
    sc = (1.0 / N) * (np.arange(1, N + 1) - N - 0.5)

    if vtransform == 2:
        csrf = (1 - np.cosh(theta_s * sc)) / (np.cosh(theta_s) - 1)
        csb  = (np.exp(theta_b * sc) - 1) / (1 - np.exp(-theta_b))
        Cs   = (1 - theta_b) * csrf + theta_b * csb
        z0   = (hc * sc + h * Cs) / (hc + h)
        z    = zeta * (1 + z0) + h * z0
    else:
        cff1 = 1.0 / np.sinh(theta_s)
        Cs   = (1 - theta_b) * cff1 * np.sinh(theta_s * sc) + \
               theta_b * (0.5 / np.tanh(0.5 * theta_s) *
               np.tanh(theta_s * (sc + 0.5)) - 0.5)
        z    = hc * sc + (h - hc) * Cs + zeta * (1 + hc * sc / h + Cs)
    return z

# Leer parámetros desde los atributos globales del archivo
theta_s    = ds.attrs['theta_s']
theta_b    = ds.attrs['theta_b']
hc         = ds.attrs['hc']
N          = ds.dims['s_rho']
vtransform = int(ds.attrs.get('Vtransform', 2))

h_pt    = float(ds['h'].values[eta_i, xi_i])
```

```
zeta_pt = float(ds['zeta'].isel(ocean_time=0).values[eta_i, xi_i])
z = sigma_a_profundidad(h_pt, zeta_pt, theta_s, theta_b, hc, N, vtransform)

temp_perfil = ds['temp'].isel(ocean_time=0, eta_rho=eta_i, xi_rho=xi_i).values
salt_perfil = ds['salt'].isel(ocean_time=0, eta_rho=eta_i, xi_rho=xi_i).values

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 6), sharey=True)
ax1.plot(temp_perfil, z, 'o-', color='tomato', ms=4)
ax1.set_xlabel('Temperatura (°C)'); ax1.set_ylabel('Profundidad (m)')
ax2.plot(salt_perfil, z, 'o-', color='steelblue', ms=4)
ax2.set_xlabel('Salinidad (PSU)')
for ax in (ax1, ax2):
    ax.invert_yaxis()
    ax.grid(True, lw=0.5)
plt.suptitle('Perfil T-S')
plt.tight_layout()
```

25.7 Sección transversal

Un corte longitudinal a índice `eta_rho` constante:

```
eta_corte = 75   # fila de la grilla (latitud fija)

temp_secc = ds['temp'].isel(ocean_time=0, eta_rho=eta_corte).values  # (N, xi_rho)
lon_secc  = ds['lon_rho'].values[eta_corte, :]
h_secc    = ds['h'].values[eta_corte, :]
zeta_secc = ds['zeta'].isel(ocean_time=0).values[eta_corte, :]

# Profundidad real en cada columna: shape (N, xi_rho)
z_secc = np.array([
    sigma_a_profundidad(h_secc[xi], zeta_secc[xi],
                        theta_s, theta_b, hc, N, vtransform)
    for xi in range(len(lon_secc))
]).T

fig, ax = plt.subplots(figsize=(12, 5))
pc = ax.pcolormesh(
    np.tile(lon_secc, (N, 1)),
    z_secc, temp_secc,
    cmap='RdYlBu_r', shading='auto'
)
plt.colorbar(pc, ax=ax, label='Temperatura (°C)')
ax.set_xlabel('Longitud'); ax.set_ylabel('Profundidad (m)')
ax.set_title('Sección de temperatura')
plt.tight_layout()
```

[>] Lazy loading y memoria — xr.open_dataset no carga los datos en memoria has-

ta que los usas. Si solo necesitas algunas variables, extráelas antes de hacer cálculos: python temp = ds[‘temp’].isel(ocean_time=0, s_rho=-1).values # solo aquí lee del disco

[>] Múltiples archivos — CROCO suele generar un archivo por período. Para abrirlos como una sola serie temporal: python ds = xr.open_mfdataset(’croco_his_Y*.nc’, combine=‘by_coords’)

26 Sentinel-2: API STAC de Copernicus y análisis de cobertura MGRS

Sentinel-2 es el satélite de observación terrestre de la Agencia Espacial Europea (ESA). Tiene un ciclo de revisita de 5 días, captura imágenes a 10 m de resolución en el espectro visible e infrarrojo, y los datos son de libre acceso.

26.1 El sistema de tiles MGRS

Las imágenes Sentinel-2 se organizan en el sistema **MGRS** (*Military Grid Reference System*): una grilla mundial de celdas de 100 km × 100 km identificadas por un código alfanumérico (por ejemplo `19HBB`, `18HYE`). Cada imagen descargada corresponde a un tile específico. Conocer qué tiles cubren tu zona de estudio es el primer paso antes de buscar o descargar imágenes.

26.2 La API STAC de Copernicus

La plataforma **Copernicus Data Space Ecosystem** (sucesor de SciHub desde 2023) expone su catálogo de imágenes mediante el protocolo **STAC** (*SpatioTemporal Asset Catalog*): un estándar JSON para búsqueda de datos satelitales por área geográfica, período y colección. Solo se necesita Python — sin cuenta, sin Google, sin credenciales para consultar metadatos.

```
pip install pystac-client geopandas matplotlib shapely
```

26.3 Buscar imágenes en un área

```
import pystac_client

client = pystac_client.Client.open(
    "https://catalogue.dataspace.copernicus.eu/stac"
)

# Bounding box: [lon_min, lat_min, lon_max, lat_max]
BBOX = [-73.2, -35.3, -68.8, -31.3]   # RM, Valparaíso y O'Higgins

resultados = client.search(
    collections=["sentinel-2-l2a"],
    bbox=BBOX,
    datetime="2024-01-01/2024-01-31",
)
items = list(resultados.items())
print(f"Imágenes encontradas: {len(items)}")
```

26.4 Extraer metadatos de cada imagen

Cada resultado (*item*) tiene propiedades de la escena. Las más útiles:

```
import pandas as pd

registros = []
for item in items:
    p = item.properties
    registros.append({
        "id":         item.id,
        "datetime":   p.get("datetime"),
        "mgrs_tile": p.get("grid:code", "").replace("MGRS-", ""),
        "cloud_pct": p.get("eo:cloud_cover"),
        "geometry":   item.geometry,       # dict GeoJSON con el polígono del tile
    })

df = pd.DataFrame(registros)
df["datetime"] = pd.to_datetime(df["datetime"])
print(df[["datetime", "mgrs_tile", "cloud_pct"]].head(10))
```

Campo	Descripción
`mgrs_tile`	Código del tile (ej. `19HBB`)
`cloud_pct`	Porcentaje de cobertura de nubes (0–100)
`geometry`	Polígono del tile en coordenadas geográficas

26.5 Descarga masiva de metadatos: mes a mes

Para un análisis de largo plazo lo eficiente es descargar solo metadatos y guardar un CSV local para no repetir la consulta:

```
import time, json
import pystac_client
import pandas as pd
from datetime import date

BBOX  = [-73.2, -35.3, -68.8, -31.3]
AÑOS  = [2021, 2022, 2023, 2024]
OUT   = "metadata_sentinel2.csv"

client = pystac_client.Client.open(
    "https://catalogue.dataspace.copernicus.eu/stac"
)

registros = []
for year in AÑOS:
    for month in range(1, 13):
```

```
        d0 = date(year, month, 1)
        d1 = date(year, month + 1, 1) if month < 12 else date(year + 1, 1, 1)
        print(f"{year}-{month:02d}...", end=" ", flush=True)

        for intento in range(3):
            try:
                items = list(client.search(
                    collections=["sentinel-2-l2a"],
                    bbox=BBOX,
                    datetime=f"{d0}/{d1}",
                ).items())
                break
            except Exception:
                time.sleep(10 * (intento + 1))
                client = pystac_client.Client.open(
                    "https://catalogue.dataspace.copernicus.eu/stac"
                )

        print(f"{len(items)} imágenes")
        for item in items:
            p = item.properties
            registros.append({
                "year":      year,
                "month":     month,
                "mgrs_tile": p.get("grid:code", "").replace("MGRS-", ""),
                "cloud_pct": p.get("eo:cloud_cover"),
                "datetime":  p.get("datetime"),
                "geometry":  json.dumps(item.geometry),
            })

df = pd.DataFrame(registros)
df.to_csv(OUT, index=False)
print(f"Guardado: {OUT}  ({len(df)} filas)")
```

26.6 Contar imágenes por tile y mes

```
import pandas as pd

df = pd.read_csv("metadata_sentinel2.csv")

conteo = (df.groupby(["year", "month", "mgrs_tile"])
            .size()
            .reset_index(name="n_images"))

# Tiles con mayor cobertura total
por_tile = df.groupby("mgrs_tile").size().sort_values(ascending=False)
```

```
print(por_tile.head(10).to_string())
```

26.7 Intersectar tiles con una región de estudio

Para quedarse solo con los tiles que caen dentro de tu zona de interés:

```
import json, geopandas as gpd
from shapely.geometry import shape
from shapely.ops import unary_union

df = pd.read_csv("metadata_sentinel2.csv")

# Reconstruir polígono real de cada tile (unión de todas sus escenas)
df["geom_obj"] = df["geometry"].apply(lambda g: shape(json.loads(g)))
tile_geoms = df.groupby("mgrs_tile")["geom_obj"].apply(unary_union)

tiles_gdf  = gpd.GeoDataFrame({"geometry": tile_geoms}, crs="EPSG:4326")
region_gdf = gpd.read_file("mi_region.geojson")   # polígono de la zona de estudio

tiles_en_region = set(
    gpd.sjoin(tiles_gdf, region_gdf[["geometry"]],
              how="inner", predicate="intersects").index.unique()
)
print(f"Tiles dentro de la región: {len(tiles_en_region)}")
```

26.8 Visualizar cobertura mensual en un mapa

```
import matplotlib.pyplot as plt
import geopandas as gpd

mes    = df[(df["year"] == 2024) & (df["month"] == 1)]
conteo = mes.groupby("mgrs_tile").size().reset_index(name="n")

tiles_mes = gpd.GeoDataFrame(
    conteo.assign(geometry=conteo["mgrs_tile"].map(tile_geoms)),
    crs="EPSG:4326"
).dropna(subset=["geometry"])

fig, ax = plt.subplots(figsize=(8, 9))
tiles_mes.plot(ax=ax, column="n", cmap="YlOrRd",
               edgecolor="gray", linewidth=0.4, alpha=0.85,
               legend=True, legend_kwds={"label": "Imágenes / tile"})
region_gdf.boundary.plot(ax=ax, edgecolor="steelblue", linewidth=0.8)
ax.set_title("Cobertura Sentinel-2 — Enero 2024")
ax.set_xlabel("Longitud"); ax.set_ylabel("Latitud")
plt.tight_layout()
```

```
plt.savefig("cobertura_enero_2024.png", dpi=150)
```

26.9 Descargar imágenes reales

Los metadatos STAC incluyen los enlaces a los archivos. Para descargar las bandas de una imagen concreta se necesita una cuenta gratuita en `dataspace.copernicus.eu`:

```
import requests, os

# Autenticación — obtener token
def obtener_token(usuario, clave):
    r = requests.post(
        "https://identity.dataspace.copernicus.eu/auth/realms/CDSE/protocol/openid-
        ↪  connect/token",
        data={
            "client_id": "cdse-public",
            "grant_type": "password",
            "username": usuario,
            "password": clave,
        }
    )
    r.raise_for_status()
    return r.json()["access_token"]

token = obtener_token("mi_email@ejemplo.com", "mi_clave")
headers = {"Authorization": f"Bearer {token}"}
```

```
# Encontrar la imagen con menos nubes en enero 2024
item_mejor = min(items, key=lambda i: i.properties.get("eo:cloud_cover", 100))

# Los assets incluyen cada banda por separado
for nombre, asset in item_mejor.assets.items():
    print(f"{nombre:20s}  {asset.href}")
# B02    https://.../.../B02.tif      ← azul (10 m)
# B03    https://.../.../B03.tif      ← verde (10 m)
# B04    https://.../.../B04.tif      ← rojo (10 m)
# B08    https://.../.../B08.tif      ← NIR (10 m)
# ...
```

```
def descargar_banda(href, ruta_local, headers):
    """Descarga una banda Sentinel-2 con autenticación."""
    os.makedirs(os.path.dirname(ruta_local), exist_ok=True)
    with requests.get(href, headers=headers, stream=True) as r:
        r.raise_for_status()
        with open(ruta_local, 'wb') as f:
            for chunk in r.iter_content(chunk_size=1 << 20):   # 1 MB
                f.write(chunk)
```

```
    print(f"Descargado: {os.path.basename(ruta_local)}")

# Descargar banda roja y NIR para calcular NDVI
tile_id = item_mejor.id
descargar_banda(item_mejor.assets['B04'].href,
                f"imagenes/{tile_id}_B04.tif", headers)
descargar_banda(item_mejor.assets['B08'].href,
                f"imagenes/{tile_id}_B08.tif", headers)
```

```
# Calcular NDVI con rasterio
import rasterio
import numpy as np

with rasterio.open(f"imagenes/{tile_id}_B04.tif") as src:
    rojo = src.read(1).astype(float)
with rasterio.open(f"imagenes/{tile_id}_B08.tif") as src:
    nir  = src.read(1).astype(float)
    meta = src.meta.copy()

ndvi = (nir - rojo) / (nir + rojo + 1e-10)

meta.update(dtype='float32', count=1)
with rasterio.open(f"imagenes/{tile_id}_NDVI.tif", 'w', **meta) as dst:
    dst.write(ndvi.astype('float32'), 1)
```

[>] Filtrar por nubosidad antes de descargar — La nubosidad está disponible en los metadatos, lo que permite filtrar antes de descargar cualquier imagen: python df_claras = df[df[“cloud_pct”] < 20] print(f”Imágenes con <20% nubes: {len(df_claras)} de {len(df)}“)

[>] Sin cuenta requerida — La API STAC de Copernicus es pública: no requiere registro para consultar metadatos. La cuenta gratuita en dataspace.copernicus.eu solo es necesaria si se quieren descargar los archivos de imagen completos (.SAFE, .tif).

27 Glosario

Referencia rápida de términos usados en el manual.

27.1 Python

tipo de dato (`type`) Clasificación que determina qué valores puede tener una variable y qué operaciones se le pueden aplicar. Los tipos fundamentales de Python son `float`, `int`, `str`, `bool` y `None`.

float Número de punto flotante (con decimales). En Python puro siempre es de 64 bits (~15 dígitos de precisión). En NumPy puede ser `float32` (32 bits, ~7 dígitos) o `float64`.

int Número entero sin parte decimal. En Python 3 no tiene límite de tamaño. Se usa para índices, contadores y cantidades discretas.

str Cadena de texto. Secuencia inmutable de caracteres Unicode.

bool Valor lógico: `True` o `False`. Es un subtipo de `int` (`True == 1`, `False == 0`).

None Ausencia de valor. Equivalente a `NULL` en SQL o `null` en otros lenguajes. Se verifica con `is None`, no con `== None`.

NaN (*Not a Number*) Valor especial de punto flotante que representa un dato faltante o inválido. `np.nan` en NumPy, generado automáticamente por pandas al leer celdas vacías. A diferencia de `None`, es de tipo `float` y permite operar con arrays numéricos. `NaN != NaN` es siempre `True` — para verificar usar `np.isnan()` o `pd.isna()`.

lista Colección ordenada y mutable de elementos de cualquier tipo. Se define con corchetes: `[1, "texto", True]`. Permite índices, slices y `.append()`.

tupla Como una lista pero inmutable — no se puede modificar después de creada. Se define con paréntesis: `(3.5, 270)`. Python la usa automáticamente para retornar múltiples valores de una función: `return media, std` devuelve una tupla.

diccionario Colección de pares clave-valor. Las claves son únicas. Se define con llaves: `{"lat": -31.9, "lon": -71.5}`. Acceso en O(1) por clave.

índice (`index`) Posición numérica de un elemento en una secuencia. En Python comienza en 0. Índices negativos cuentan desde el final: `lista[-1]` es el último elemento.

slice Selección de un rango de elementos: `lista[2:5]` extrae los elementos en posiciones 2, 3 y 4 (el límite superior no se incluye). También funciona en arrays y DataFrames.

comprehension Forma compacta de construir una lista, conjunto o diccionario. `[x*2 for x in lista if x > 0]` es equivalente a un `for` + `append` pero en una línea.

función Bloque de código con nombre que recibe parámetros, ejecuta una tarea y opcionalmente retorna un valor. Se define con `def`. Permite reutilizar lógica sin repetirla.

módulo Archivo `.py` que contiene funciones, clases y variables importables. `import numpy as np` carga el módulo numpy con el alias `np`.

paquete Carpeta que contiene módulos y un archivo `__init__.py`. Numpy, pandas y matplotlib son paquetes.

scope (alcance) Región del código donde una variable es visible. Las variables defi-

nidas dentro de una función son locales — no existen fuera de ella. Para compartir datos entre funciones, usar el valor de retorno.

excepción Error en tiempo de ejecución. Se captura con `try / except`. Los tipos más comunes: `FileNotFoundError`, `KeyError` (clave inexistente en dict), `IndexError` (índice fuera de rango), `TypeError` (tipo incorrecto), `ValueError` (valor inválido).

27.2 Herramientas y entorno

entorno virtual Instalación aislada de Python con sus propios paquetes, independiente del sistema global. Permite que distintos proyectos usen versiones distintas de las mismas librerías sin conflictos.

conda Gestor de paquetes y entornos incluido en Anaconda. A diferencia de pip, resuelve dependencias binarias (C, Fortran) además de Python. Comandos principales: `conda create`, `conda activate`, `conda install`, `conda env export`.

pip Gestor de paquetes oficial de Python. Instala desde PyPI. Se usa junto con conda para paquetes que no están en los repositorios de conda.

environment.yml Archivo YAML que describe un entorno conda: nombre, canales, y lista de paquetes con versiones exactas. Se genera con `conda env export` y permite recrear el entorno exacto en otro equipo con `conda env create -f environment.yml`.

YAML (*YAML Ain't Markup Language*) Formato de texto para configuración y datos estructurados. Más legible que JSON: no usa comillas en strings simples y admite comentarios con `#`. Estándar para archivos de configuración de proyectos y pipelines. Se lee en Python con `yaml.safe_load()` del paquete `pyyaml`.

tqdm Librería para barras de progreso. Envuelve cualquier iterable: `for x in tqdm(lista)` muestra una barra con porcentaje, velocidad y tiempo estimado. Especialmente útil en loops que procesan decenas o cientos de archivos.

logging Módulo de la librería estándar de Python para registrar mensajes de diagnóstico. A diferencia de `print`, cada mensaje tiene timestamp y nivel de severidad (`DEBUG`, `INFO`, `WARNING`, `ERROR`). Permite escribir simultáneamente a consola y a un archivo de log sin modificar el código.

27.3 NumPy

array Estructura de datos central de NumPy. Arreglo multidimensional de elementos del mismo tipo. A diferencia de una lista, permite operaciones matemáticas directas sobre todos sus elementos sin loop.

dtype Tipo de dato de los elementos de un array NumPy. Los más comunes: `float64` (64 bits, predeterminado para floats), `float32` (32 bits, usado en modelos numéricos y datos satelitales para ahorrar memoria), `int32`, `int64`, `bool`. Se verifica con `array.dtype` y se convierte con `array.astype(np.float64)`.

float32 vs float64 `float64` tiene ~15 dígitos significativos de precisión; `float32` tiene ~7. Los archivos NetCDF de modelos oceánicos (CROCO) y datos satelitales suelen usar `float32` para reducir tamaño en disco. Al operar entre `float32` y `float64`, NumPy convierte todo a `float64` automáticamente.

shape Tupla con las dimensiones de un array. `(186, 11)` significa 186 filas y 11 columnas. `array.shape`, `array.ndim` (número de dimensiones), `array.size` (total de elementos).

vectorización Aplicar una operación a todos los elementos de un array sin escribir un loop explícito. `vel * 1.944` convierte todos los valores a nudos en una sola instrucción, más rápido que un `for`.

broadcasting Regla de NumPy que permite operar arrays de distinto tamaño cuando sus formas son compatibles. Operaciones entre un array y un escalar, o entre arrays de formas como `(N,)` y `(N, M)`, se "expanden" automáticamente.

máscara booleana Array de `True`/`False` del mismo tamaño que otro array, usado para seleccionar o modificar elementos. `vel[vel > 0.5]` aplica una máscara generada por la comparación.

27.4 Pandas

DataFrame Tabla de datos con filas y columnas etiquetadas. Equivalente a una hoja de Excel pero operable desde código. Cada columna es una `Series`.

Series Columna de un DataFrame: array unidimensional con un índice. Tiene nombre, dtype y las mismas operaciones que un array NumPy.

índice (`Index`) Etiquetas de las filas. Por defecto es numérico (0, 1, 2…). En series temporales es un `DatetimeIndex`. El índice es lo que usa `loc` para seleccionar filas.

DatetimeIndex Índice de fechas y horas. Permite filtrar por rango (`df['2025-10':'2026-03']`), resamplear (`df.resample('1h').mean()`) y agrupar por hora, día o mes.

`loc` vs `iloc` `loc` selecciona por **etiqueta** del índice; `iloc` selecciona por **posición** numérica. Con un índice numérico parecen iguales, pero con `DatetimeIndex` la diferencia es crucial: `df.loc['2025-10-01']` filtra por fecha, `df.iloc[0]` siempre es la primera fila.

resampleo (`resample`) Cambiar la resolución temporal de una serie. `df.resample('1h').mean()` agrupa en ventanas de 1 hora y calcula la media de cada una.

groupby Divide el DataFrame en grupos según el valor de una columna, aplica una función a cada grupo y combina los resultados. Equivalente a una tabla dinámica por categoría.

rolling Ventana deslizante sobre una Serie temporal. `df['vel'].rolling('1h').mean()` calcula la media del último hora en cada punto. Se usa para suavizar series con ruido de alta frecuencia.

pd.cut Divide una columna continua en intervalos discretos con etiquetas. `pd.cut(df['vel'], bins=[0, 0.1, 0.5, 1.0], labels=['calma','leve','fuerte'])` asigna una categoría a cada valor. Se usa para construir tablas de incidencia velocidad × dirección.

merge Une dos DataFrames por una columna común, equivalente a un JOIN de SQL. El parámetro `how` controla qué filas se conservan: `inner` (solo matches), `left` (todas las del primero), `outer` (todas las de ambos).

concat Apila DataFrames con la misma estructura vertialmente (más filas) u horizontalmente (más columnas). Se usa para unir datos de distintos períodos antes de calcular estadísticas globales.

pathlib.Path Clase de la librería estándar para manejar rutas de archivos de forma orientada a objetos y multiplataforma. `Path('/ruta') / 'subcarpeta' / 'archivo.csv'` construye rutas sin concatenar strings con `/` o `os.path.join`.

27.5 Visualización

Figure El contenedor principal de matplotlib. Todo gráfico existe dentro de una `Figure`. Se crea con `plt.figure()` o `plt.subplots()`. Controla el tamaño total (`figsize`) y el guardado (`fig.savefig()`).

Axes El panel de dibujo dentro de una Figure. Contiene los ejes X e Y, los datos graficados, etiquetas y leyenda. Una Figure puede tener varios Axes (subplots).

backend Motor que matplotlib usa para renderizar y mostrar figuras. `Agg`: sin ventana, renderiza a memoria (para guardar archivos). `TkAgg`: ventana Tkinter interactiva, estable en Spyder. `Qt5Agg`: ventana Qt5, puede interferir con el event loop de Spyder. `inline`: hook de IPython sobre Agg que embebe la figura como imagen en la consola.

event loop Bucle de escucha continua de eventos (clics, teclado, redimensionado) que necesita cualquier ventana de escritorio para responder al usuario. Python solo puede correr un event loop a la vez en el hilo principal, por eso `plt.show()` bloquea la ejecución hasta que se cierra la ventana.

matplotlib.widgets Módulo de matplotlib que provee controles interactivos dentro del canvas de una figura: `Slider`, `Button`, `CheckButtons`, `RadioButtons`. No requieren construir una aplicación Qt o Tk completa, pero están limitados al espacio del gráfico.

FuncAnimation Clase de `matplotlib.animation` que crea animaciones llamando repetidamente a una función de actualización. El resultado se puede mostrar en vivo o exportar como GIF (requiere pillow) o MP4 (requiere ffmpeg).

GridSpec Clase de `matplotlib.gridspec` para layouts de subplots con tamaños desiguales. Permite definir `height_ratios` y `width_ratios` para que algunos paneles sean más grandes que otros, y posicionar cada Axes en celdas arbitrarias de la grilla.

twin axes Técnica de matplotlib para superponer dos variables con escalas distintas en el mismo panel. `ax2 = ax1.twinx()` crea un eje Y derecho que comparte el eje X con el izquierdo. Útil para comparar velocidad y temperatura en la misma serie temporal.

PyQt5 Librería para construir aplicaciones de escritorio completas en Python usando el framework Qt5. Permite crear ventanas con menús, botones y tablas, y embeber matplotlib como un widget (`FigureCanvasQTAgg`). Más potente que `matplotlib.widgets` pero más complejo. Los scripts PyQt5 deben correrse fuera de Spyder.

27.6 Formatos y estándares

NetCDF (`.nc`) Formato científico para datos multidimensionales con coordenadas etiquetadas (tiempo, profundidad, latitud, longitud). Estándar en oceanografía y meteorología. Se lee con `xarray`.

xarray Dataset Estructura de datos para archivos NetCDF. Equivalente a un diccionario de arrays NumPy con coordenadas nombradas. Permite seleccionar por nombre de coordenada: `ds['temp'].sel(ocean_time='2024-01-01')`.

coordenadas sigma (σ) Sistema de coordenadas verticales usado en modelos oceánicos (CROCO, ROMS) donde $\sigma = 0$ es la superficie y $\sigma = -1$ es el fondo. Los niveles siguen la forma del fondo en vez de ser planos horizontales fijos.

STAC (*SpatioTemporal Asset Catalog*) Estándar JSON para catálogos de datos geoespaciales. Permite buscar imágenes satelitales por área, fecha y colección mediante una API REST, sin descargar los archivos.

MGRS (*Military Grid Reference System*) Sistema de grilla mundial que divide la superficie terrestre en celdas de 100 km × 100 km identificadas por un código alfanumérico (ej. `19HBB`). Sentinel-2 organiza sus imágenes por tile MGRS.

GCP (*Ground Control Points*, puntos de control terrestre) Pares de puntos con coordenadas conocidas en píxeles y en el mundo real. Se usan para georreferenciar imágenes: ajustar la transformación píxel → coordenada geográfica.

OCR (*Optical Character Recognition*, reconocimiento óptico de caracteres) Proceso de extraer texto desde una imagen. En el manual se usa para digitalizar valores de profundidad desde cartas batimétricas escaneadas, combinando OpenCV (preprocesamiento) y Tesseract (reconocimiento).

ADCP (*Acoustic Doppler Current Profiler*) Instrumento oceanográfico que mide velocidad y dirección de corrientes a múltiples profundidades usando el efecto Doppler sobre partículas en suspensión. Genera matrices de datos (tiempo × profundidad).