

GeoFlow

Geoenergy systems simulation

2025



DARTS:
**Una guía práctica para la
simulación y explotación de
yacimientos geotérmicos**

M.I. Luis Armando García Navarrete

ar.garcia.navarrete@gmail.com

+52 5541409846

Ciudad de México, México

Contenido

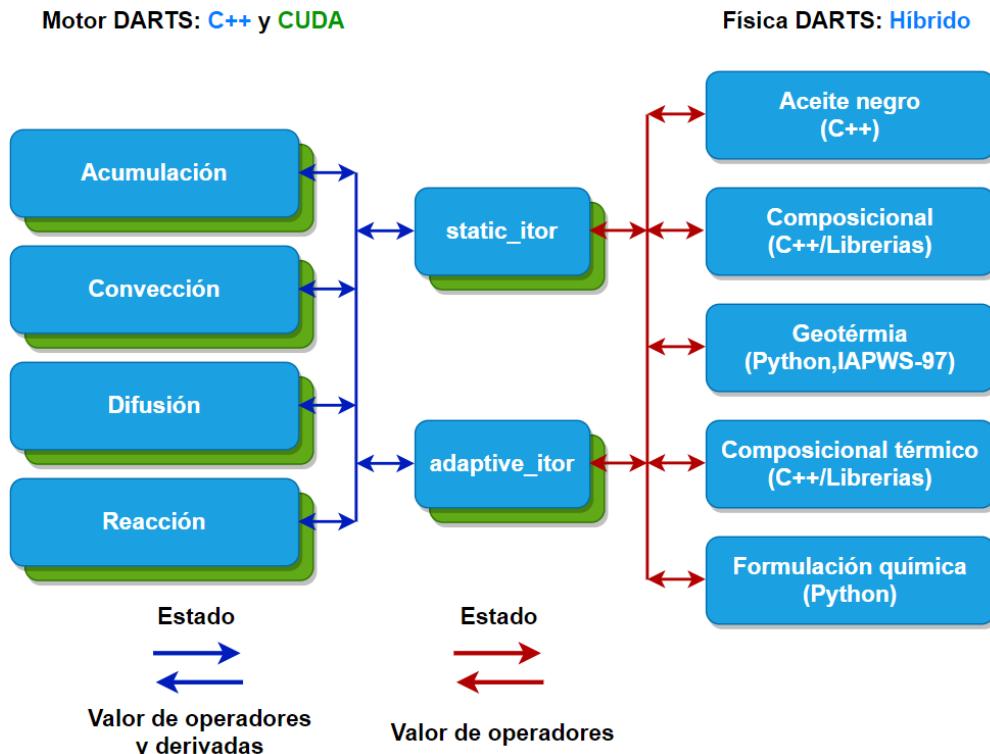
Introducción	2
Enlaces y recursos	4
Instalación	5
Python.....	5
Visual Studio Code	7
Extensiones	8
Virutal enviroment.....	15
Jupyter notebook (opcional).....	22
JSON.....	24
DARTS	29
Gmsh y Paraview.....	30
Estructura y ejemplos	31
Ejemplo 1	32
Ejemplo 2	49
Ejemplo 3	54
Ejemplo 4	58
Ejemplo 5	69
Ejemplo 6	74
Ejemplo 7	78
Ejemplo 8	81
Ejemplo 9	86
Ejemplo 10	91
Ejemplo 11	95
Ejemplo 12	102
Ejemplo 13	107
Ejemplo 14	114
Ejemplo 15	118
Ejemplo 16	121
Ejemplo 17	124
Ejemplo 18	138
Ejemplo 19	141

Introducción

La siguiente guía tiene como fin mostrar el funcionamiento general del software DARTS (“Delft Advanced Research Terra Simulator”) para la simulación numérica de yacimientos geotérmicos. Fue desarrollada como parte del proyecto PAPPIT DGAPA-PAPIIT IA104424, en colaboración con el Instituto de Energía Renovables (IER) de la Universidad Nacional Autónoma de México (UNAM) y es de uso libre. La guía tiene un enfoque práctico, por lo que se han preparado diferentes ejemplos que muestran las funcionalidades básicas de la herramienta. Todos los ejemplos pueden descargarse libremente del repositorio (<https://github.com/GeoFlow-git/DARTS-guide>). Los ejemplos tienen como fin mostrar los componentes que se requieren para correr una simulación en DARTS, además de mostrar casos de interés en el contexto de la explotación de un campo geotérmico. Varios ejemplos utilizan las herramientas Gmsh y Paraview, para la creación de mallas y la visualización de resultados, respectivamente. Aunque estas herramientas no son estrictamente necesarias para poder utilizar DARTS, son de gran ayuda para poder implementar y comprender algunas de las funciones avanzadas de DARTS (por ejemplo, en el uso del pre-procesador usado por DARTS para la creación de mallas que incorporan una representación discreta de la red de fracturas). A continuación, se muestra la lista completa de los ejemplos y una pequeña descripción de cada ejemplo:

#	Dim.	Malla	Descripción
1	1D	Malla estructurada	Construcción del modelo, clases y funciones, datos del pozo
2	1D	Malla estructurada	Encadenar corridas y plotear a diferentes tiempos
3	1D	Malla estructurada	Efecto de la viscosidad
4	1D	Malla estructurada	Evaporación flash
5	1D	Malla estructurada	Yacimiento saturado de vapor al 100%
6	1D	Malla estructurada	Curvas de permeabilidad relativa
7	2D	Malla estructurada	Introducción a Paraview, escribir archivos .vts
8	2D	Malla estructurada	Perforaciones, daño y activación de pozos
9	2D	Malla estructurada	Establecer fronteras abiertas. Correr scripts de Python en Paraview.
10	3D	Malla estructurada	Definir condiciones iniciales y propiedades mediante la profundidad
11	2D	Malla estructurada	Simulación en estado natural y yacimiento con casquete de vapor
12	3D	Malla estructurada	Fuente de calor
13	2D	Malla No estructurada	Introducción a Gmsh.
14	3D	Malla No estructurada	Definir condiciones iniciales y propiedades mediante capas
15	3D	Malla No estructurada	Geometrías complejas y refinamiento de malla
16	3D	Malla No estructurada	Geometrías complejas + fallas
17	2D	Malla No estructurada	Introducción al modelado de Fracturas
18	2D	Malla No estructurada	Uso del pre-procesador para simplificar de malla
19	2D	Malla No estructurada	Red de fracturas

Como se mencionó anteriormente, esta guía tiene un enfoque práctico, por lo que no debe considerarse un manual de usuario en sentido estricto. Se recomienda que el lector consulte la documentación oficial de DARTS, así como publicaciones especializadas que explican en detalle su funcionamiento interno, antes de ejecutar cualquiera de los ejemplos presentados aquí. Esto permitirá una comprensión más sólida del entorno de simulación y evitará errores comunes derivados de una interpretación superficial. No obstante, a continuación se presenta una descripción general concisa del simulador, con el objetivo de contextualizar su estructura básica y los componentes principales que intervienen en su operación. El simulador DARTS es un software relativamente nuevo, desarrollado por la universidad de DELFT en Países Bajos enfocada a la simulación de flujo en medio poroso, cuya aplicación se extiende a yacimientos de gas y aceite, acuíferos, proyectos de captura de CO₂, entre otros. En DARTS se implementa el método de volumen finito totalmente implícito, además de una aproximación de dos puntos para el flujo. Se usan las tablas IAPWS-IF97 para calcular las propiedades termodinámicas del fluido. Además de la discretización convencional en espacio y tiempo, DARTS utiliza una discretización “física” usando el método OBL (Operator Based Linearization). DARTS permite el modelado de fracturas mediante un esquema DFM (Discrete fracture-Matrix), además de usar un pre-procesador de código abierto para la creación de mallas conformantes. La parte central de DARTS está programada en C++, además de usar una interfaz en Python para mayor flexibilidad.



La estructura general de DARTS se muestra en la siguiente figura. En el lado izquierdo se muestran los cuatro términos físicos disponibles que definen el motor de simulación. A la derecha, se enumeran los cinco modelos principales involucrados en el paquete físico. Dependiendo del tipo de simulación que se quiera realizar, se utilizará una combinación de estos.

Enlaces y recursos

La siguiente lista contiene enlaces de diferentes recursos que creemos pueden ser útiles para aprender más sobre el funcionamiento de DARTS, además del repositorio en Github que contiene los ejemplos usados en esta guía.

Página de Darts dentro de [OpenResSim](#):

<https://openressim.com/simulator/open-darts>

Repositorios Gitlab/Github:

<https://gitlab.com/open-darts>

<https://github.com/GeoFlow-git/DARTS-guide>

Versiones:

<https://pypi.org/project/open-darts/#files>

Página de DARTS en la Universidad de Delft:

<https://darts.citg.tudelft.nl/>

Documentación:

<https://open-darts.gitlab.io/open-darts/>

Presentación:

https://darts.citg.tudelft.nl/assets/open_darts/7_darts.pdf

Referencias:

<https://darts.citg.tudelft.nl/resources/>

Videos tutoriales y webinars:

<https://www.youtube.com/@darts9764>

<https://www.youtube.com/watch?v=-zR-yvIXkWA>

<https://www.youtube.com/watch?v=9rnTrZjavEE&t=1741s>

<https://www.youtube.com/watch?v=9broz78NUQc&t=16s>

<https://www.youtube.com/watch?v=x-SDD84Q-5k>

Gmsh

<https://gmsh.info/>

Paview

<https://www.paraview.org/>

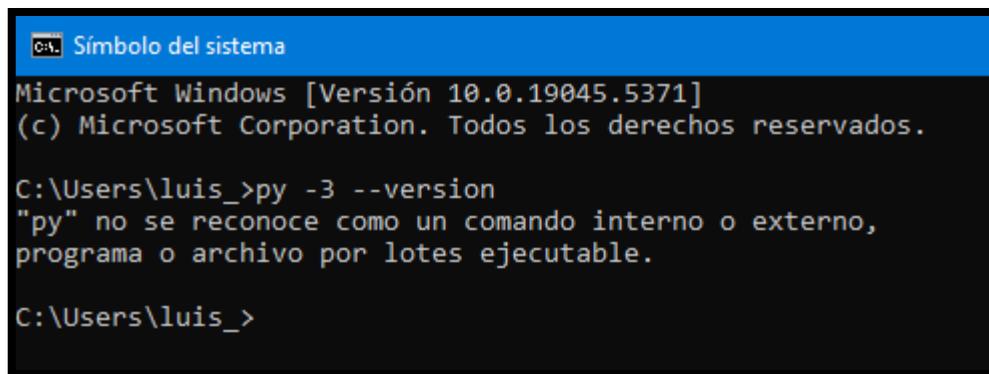
Instalación

Para poder utilizar el simulador DARTS, es necesario instalar y configurar diferentes herramientas. Esta sección contiene las instrucciones para instalar las diferentes herramientas que nosotros recomendamos utilizar para poder manipular DARST de forma sencilla y eficiente.

Python

El primer paso es instalar Python. Para verificar si ya se tiene instalado en Windows, podemos seguir los siguientes pasos:

- Abrir la línea de comandos y presión **Win + R**, posteriormente escribimos **cmd** y presionamos **Enter**
- Escribimos el comando **py -3 --version**



```
Símbolo del sistema
Microsoft Windows [Versión 10.0.19045.5371]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\luis_>py -3 --version
"py" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.

C:\Users\luis_>
```

Si no tenemos Python instalado, podemos descargarlo en <https://www.python.org/downloads/>



Python Releases for Windows

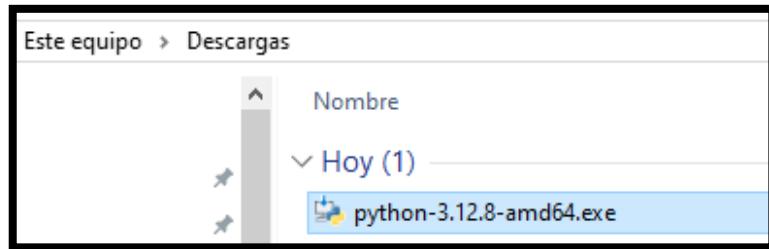
- [Latest Python 3 Release - Python 3.13.1](#)

Stable Releases

- [Python 3.12.8 - Dec. 3, 2024](#)

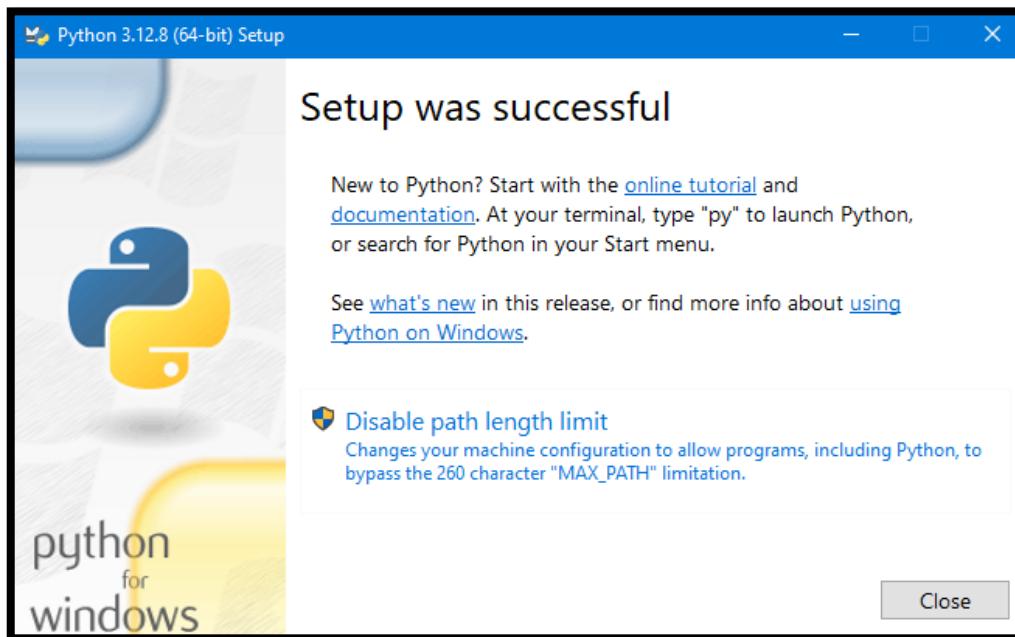
Note that Python 3.12.8 cannot be used on Windows 7 or earlier.

- [Download Windows installer \(64-bit\)](#)
- [Download Windows installer \(32-bit\)](#)
- [Download Windows installer \(ARM64\)](#)
- [Download Windows embeddable package \(64-bit\)](#)
- [Download Windows embeddable package \(32-bit\)](#)
- [Download Windows embeddable package \(ARM64\)](#)
- [Python 3.13.1 - Dec. 3, 2024](#)



Abrimos el archivo ejecutable e instalamos Python.





Una vez instalado, podemos verificar la instalación:

A screenshot of a Windows Command Prompt window titled "Símbolo del sistema". The window shows the following text:

```
Microsoft Windows [Versión 10.0.22631.5039]
(c) Microsoft Corporation. Todos los derechos reservados.

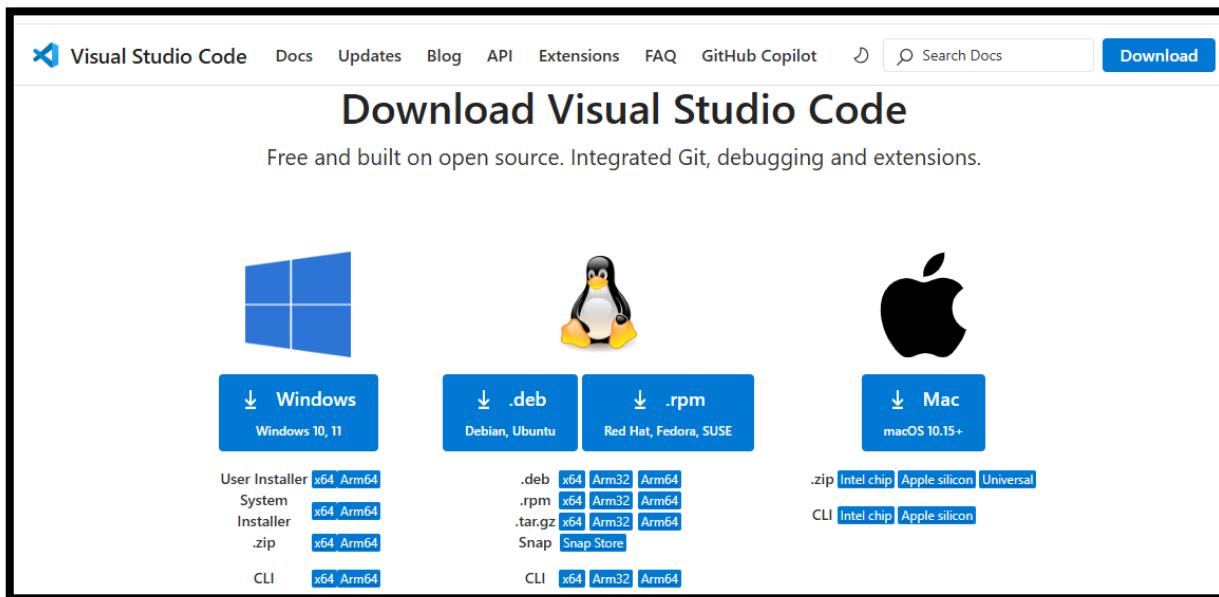
C:\Users\luis_>py --version
Python 3.12.8

C:\Users\luis_>python --version
Python 3.12.8

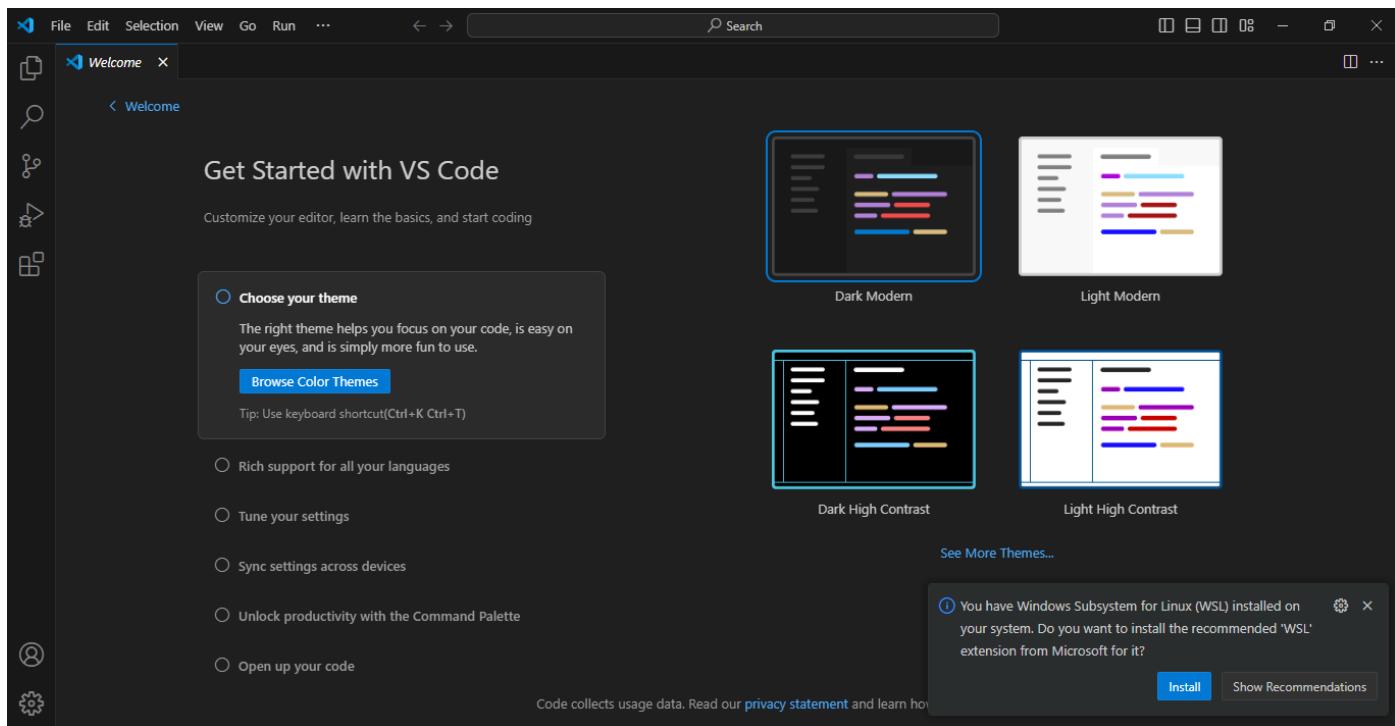
C:\Users\luis_>
```

Visual Studio Code

El siguiente paso es utilizar un editor de código. Aun que se puede utilizar cualquier editor de código deseado, recomendamos usar Visual Studio Code, ya que tiene muchas funcionalidades que nos serán útiles para poder usar y explorar DARTS de forma sencilla. Para descargarlo, nos dirigimos al siguiente link <https://code.visualstudio.com/download>

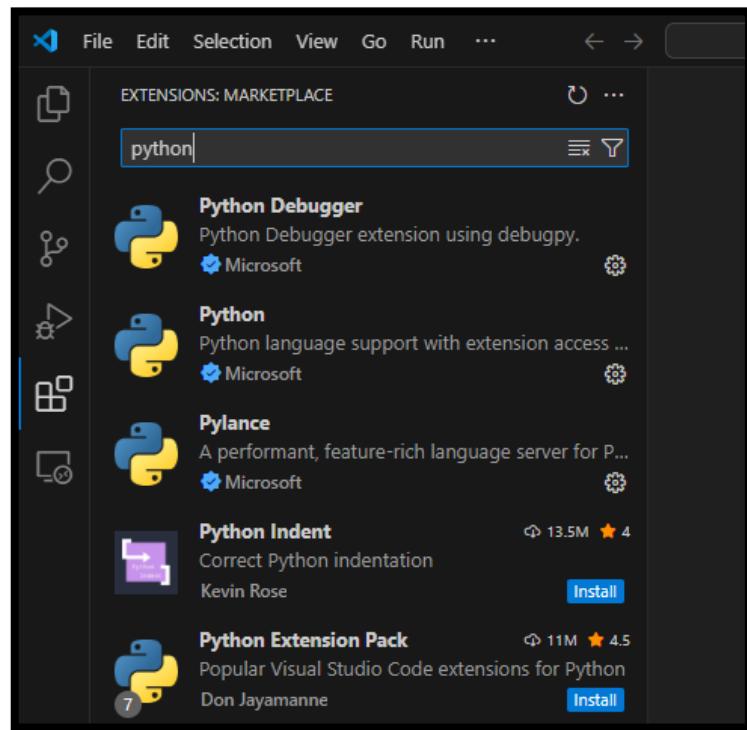


Después de instalar y abrir la aplicación:

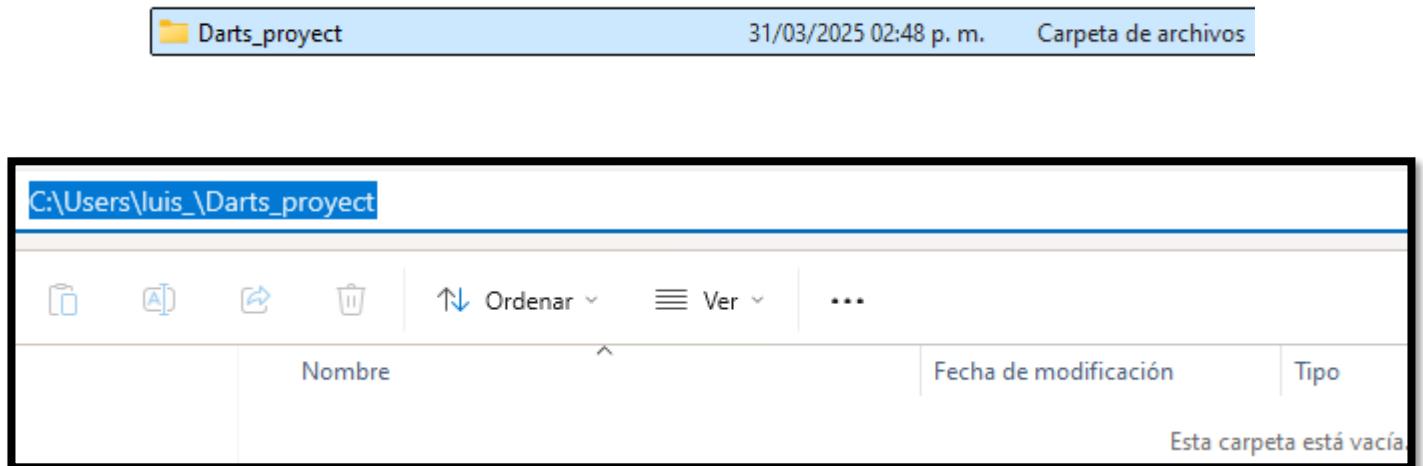


Extensiones

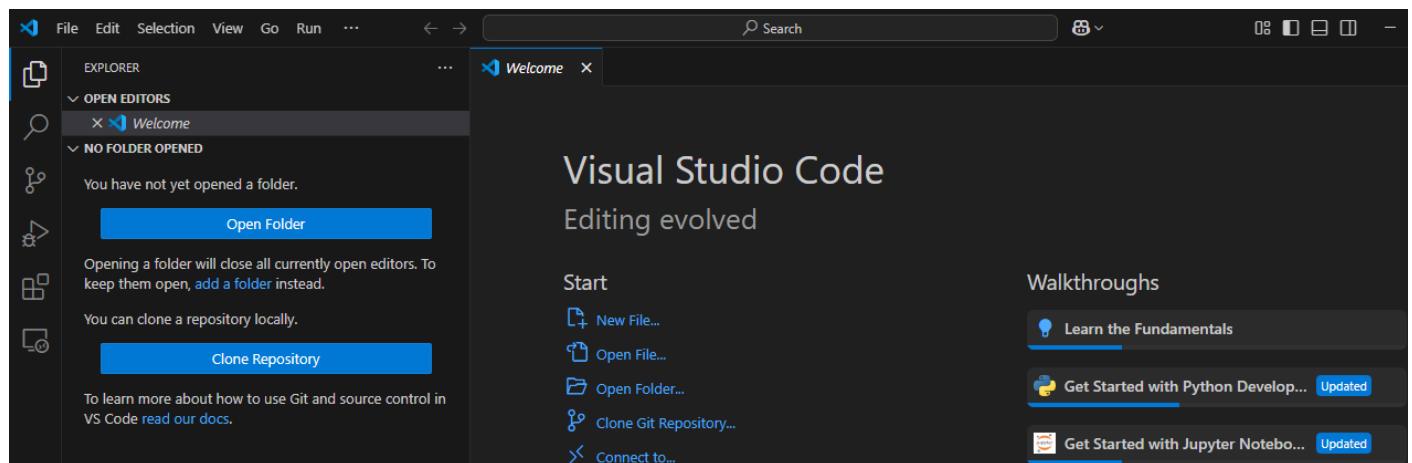
El siguiente paso es instalar las extensiones. En el menú de la izquierda, dar click en la 5º opción (extensiones:) y buscar “Python”. Debemos de instalar las primeras dos opciones:



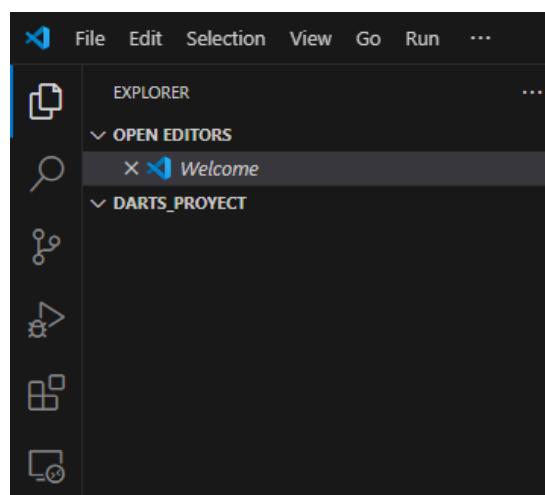
De esta forma ya podemos crear y correr scripts de Python (extensión .py) en VSC. Si queremos usar alguna librería (como numpy) primero hay que instalarla. Sin embargo, la forma recomendada para trabajar es crear un ambiente virtual (virtual environment) donde instalaremos todas las librerías que usaremos en nuestro proyecto, incluidas las librerías típicamente usadas en python como “numpy” así como las librerías propias de DARTS. Antes de crear el ambiente virtual, el primer paso es crear una carpeta para nuestro proyecto, donde se alojarán nuestros archivos de trabajo, así como el ambiente virtual como tal:



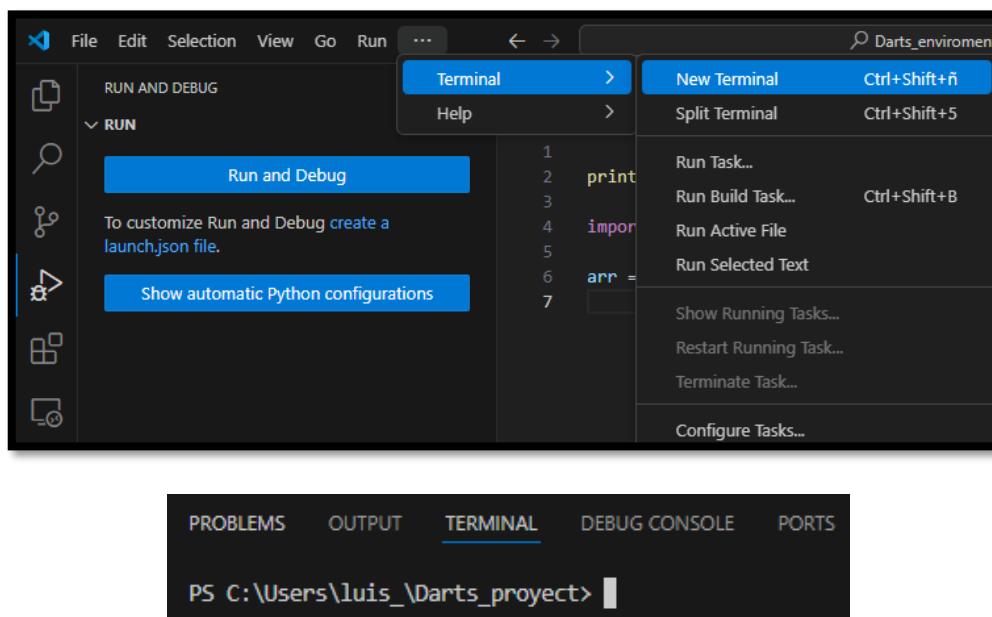
Ahora, al abrir VSC nos pedirá seleccionar una carpeta:



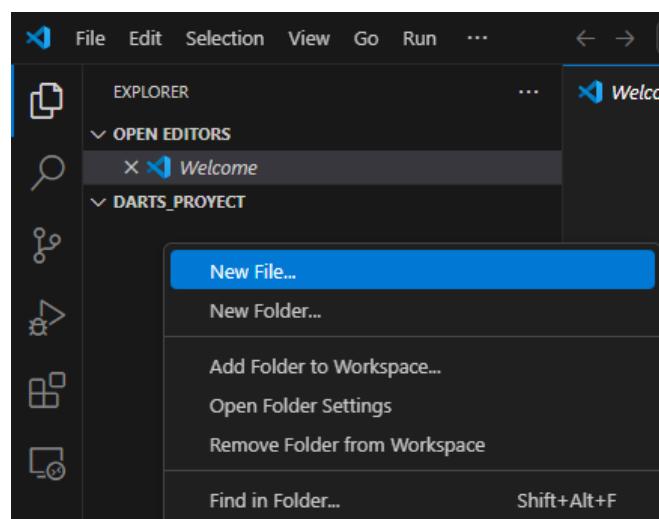
Escogemos la carpeta recién creada. Vemos que aparece el nombre de la carpeta como el directorio de trabajo actual:



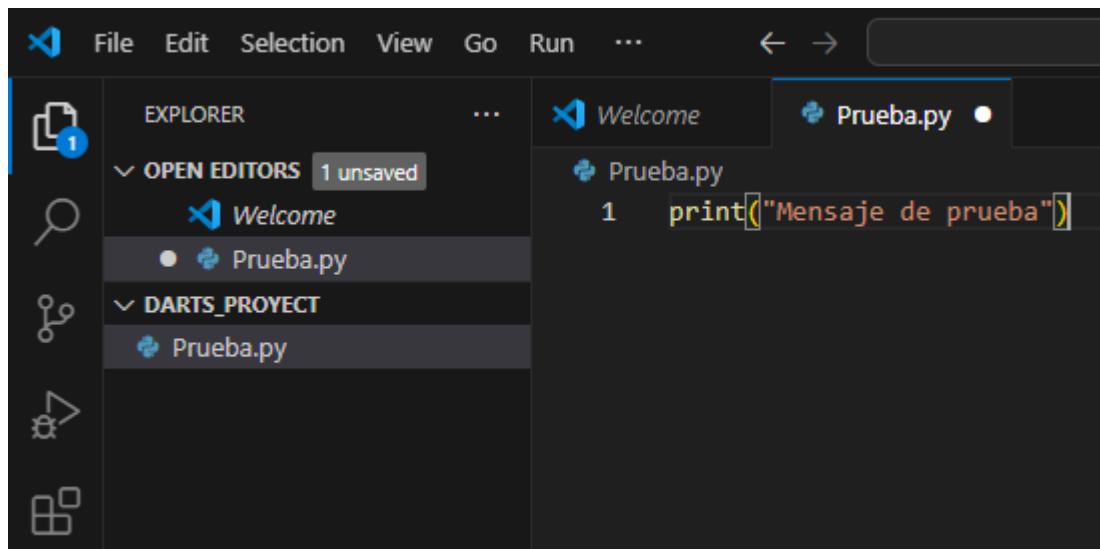
Si abrimos una nueva terminal, veremos que, de nuevo, nos ubicamos en la carpeta deseada:



Ya podemos crear un script de Python. Damos click derecho debajo del directorio de trabajo y seleccionamos "New file":



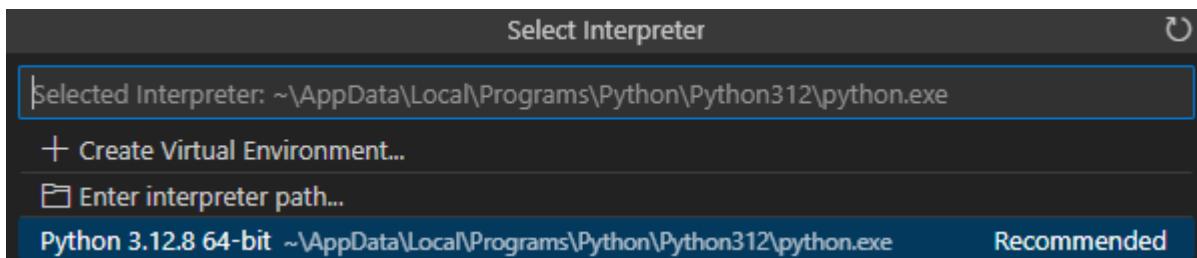
Creamos un archivo en lenguaje Python (extensión .py). Como ya instalamos las extensiones de Python necesarias, VSC reconocerá que se trata de un archivo de Python y podrá correrlo sin problema.



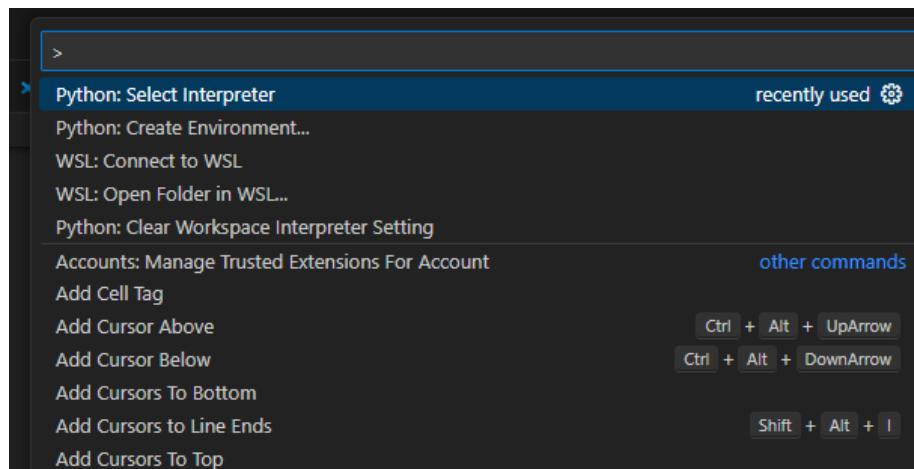
VSC escoge un intérprete de Python automáticamente al reconocer que se trata de un archivo “.py”. En la esquina inferior derecha, nos indicara el intérprete:

Ln 2, Col 1 Spaces: 4 UTF-8 CRLF {} Python 3.12.8 64-bit

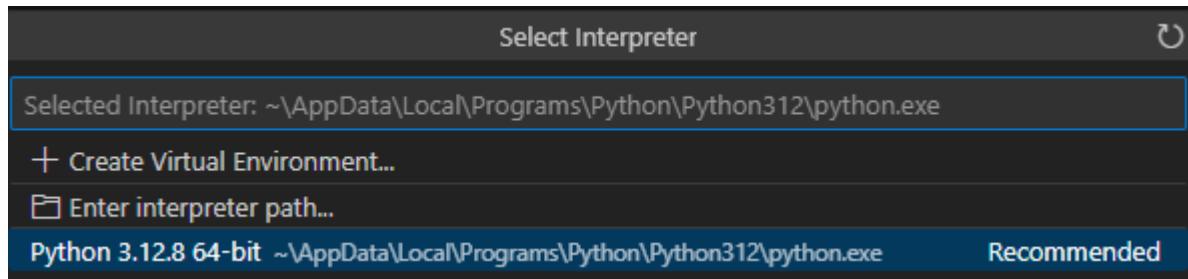
Si damos click nos manda a seleccionar el intérprete. Entre las opciones disponibles, deberá aparecer la ruta del archivo ejecutable de Python (dentro de la ruta donde instalamos Python previamente)



Además, si intentamos correr el archivo la primera vez es posible que nos pida seleccionar un intérprete de Python y el “debugger”. Para seleccionar el intérprete de Python, tecleamos **Ctrl + Shift + P** y seleccionamos la 1º opción:



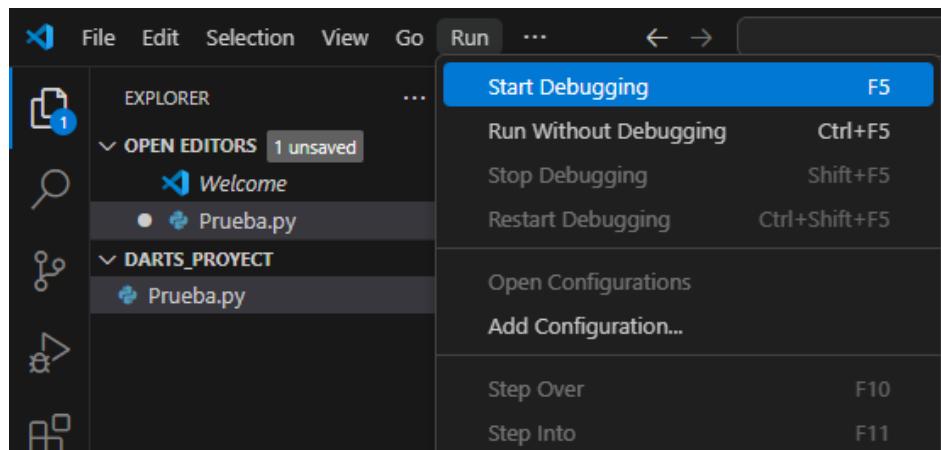
Entre las opciones disponibles, deberá aparecer la ruta del archivo ejecutable de Python (dentro de la ruta donde instalamos python previamente)



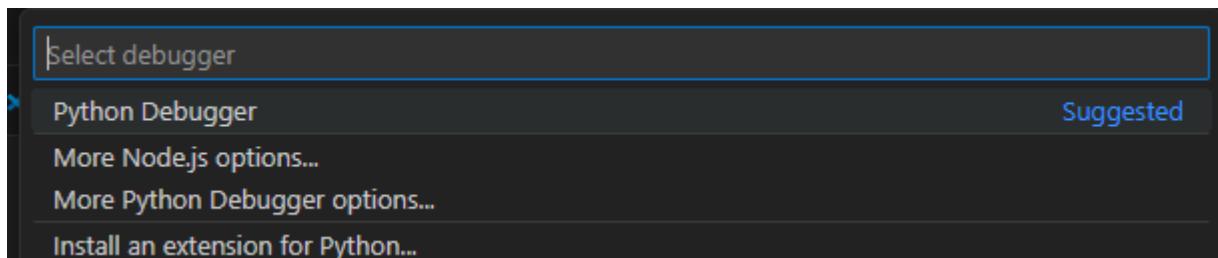
Si abrimos una nueva terminal, podemos verificar el ejecutable de Python que se está usando mediante:

```
PS C:\Users\luis_\Darts_proyect> which python
/cygdrive/c/Users/luis_/AppData/Local/Programs/Python/Python312/python
```

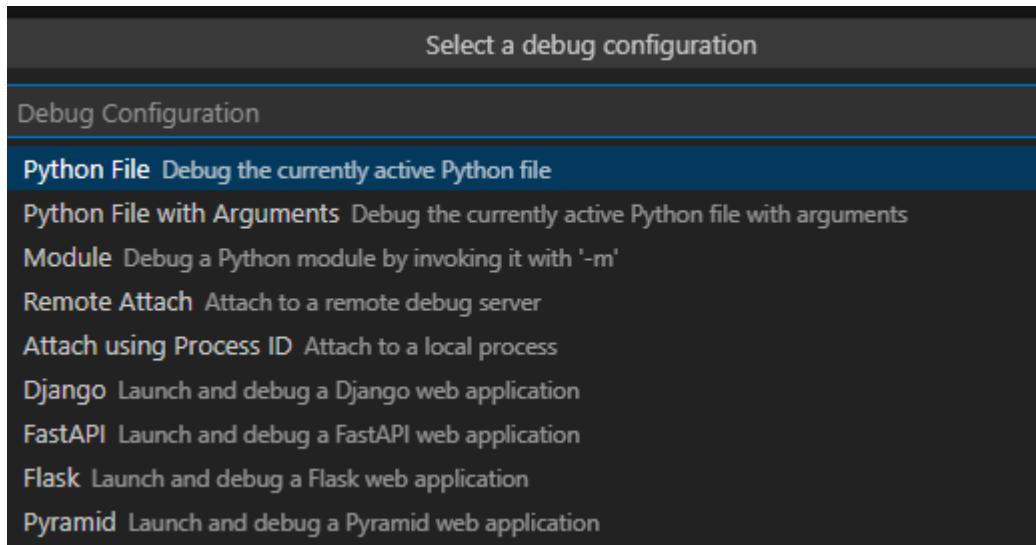
Además, al intentar correr el script nos pedirá escoger el “debugger” (cuya extensión acabamos de descargar):



Seleccionamos la 1º opción:



Escogemos la 1º opción:



En la terminal de VSC se vera el mensaje:

```
Run ... ← →
Welcome Prueba.py X
Prueba.py
1 print("Mensaje de prueba")

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

PS C:\Users\luis_\Darts_proyect> & 'c:\User: on.debugpy-2025.4.1-win32-x64\bundled\libs\de
Mensaje de prueba
PS C:\Users\luis_\Darts_proyect>
```

Vemos que en la primera línea indica la ubicación del ejecutable de Python que se está usando:

```
PS C:\Users\luis_\Darts_proyect> & 'c:\User: on.debugpy-2025.4.1-win32-x64\bundled\libs\de
```

Si queremos usar alguna librería, primero debemos instalarla. Por ejemplo, si intentamos utilizar la librería “numpy” sin instalarla previamente, marcara error:

```
Welcome Prueba.py 1
Prueba.py > ...
1 print("Mensaje de prueba")
2
3 import numpy as np
4
5 arreglo=np.array([1,2,3,4])

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

PS C:\Users\luis_\Darts_proyect> & 'c:\User: on.debugpy-2025.4.1-win32-x64\bundled\libs\de
numpy
PS C:\Users\luis_\Darts_proyect>
```

```

Welcome Prueba.py 1
Prueba.py > ...
1   print("Mensaje de prueba")
2
3   import numpy as np

Exception has occurred: ModuleNotFoundError
No module named 'numpy'

File "C:\Users\luis_\Darts_proyect\Prueba.py", line 3, in <module>
    import numpy as np
ModuleNotFoundError: No module named 'numpy'

```

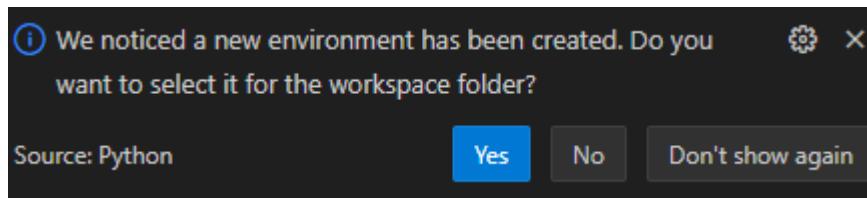
Aunque podemos instalar librerías de forma usual, lo recomendable es crear un ambiente virtual para trabajar de forma ordenada.

Virtual environment

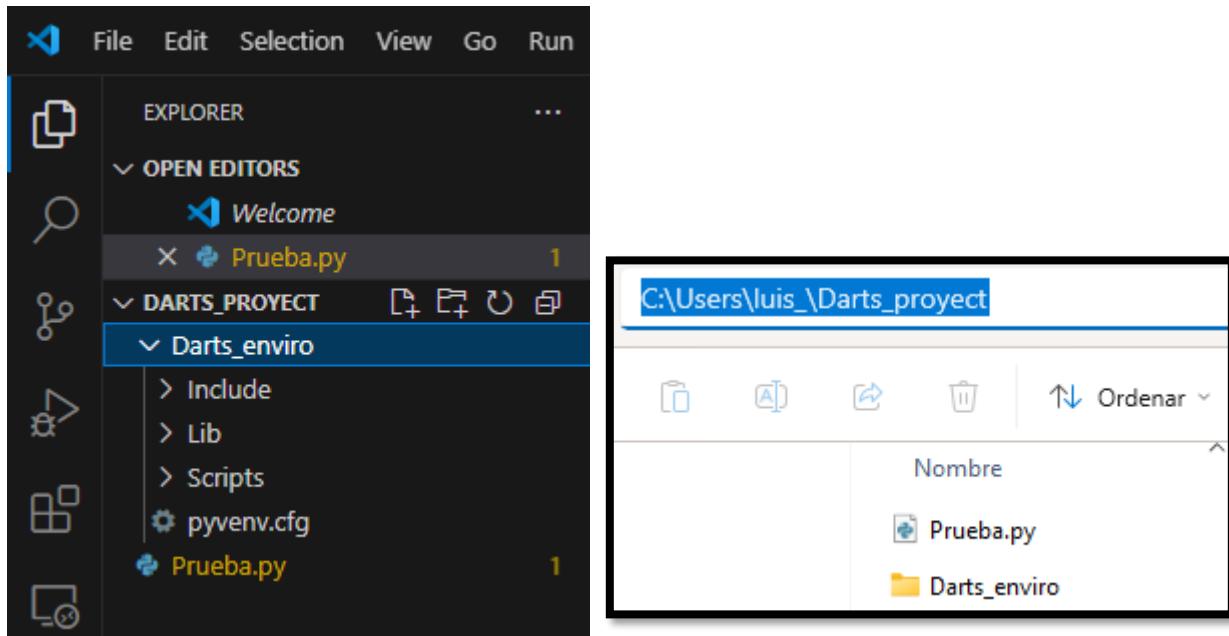
Vamos a la terminal y creamos el ambiente virtual y le damos nombre mediante el siguiente comando (el nombre del ambiente virtual se escogió como “Darts_enviro”, pero puede ser el que sea):

```
PS C:\Users\luis_\Darts_proyect> py -m venv Darts_enviro
```

Al crearlo, nos aparece el siguiente mensaje. Damos click en aceptar para que siempre que abramos a la carpeta de trabajo, se seleccione el ambiente virtual por default



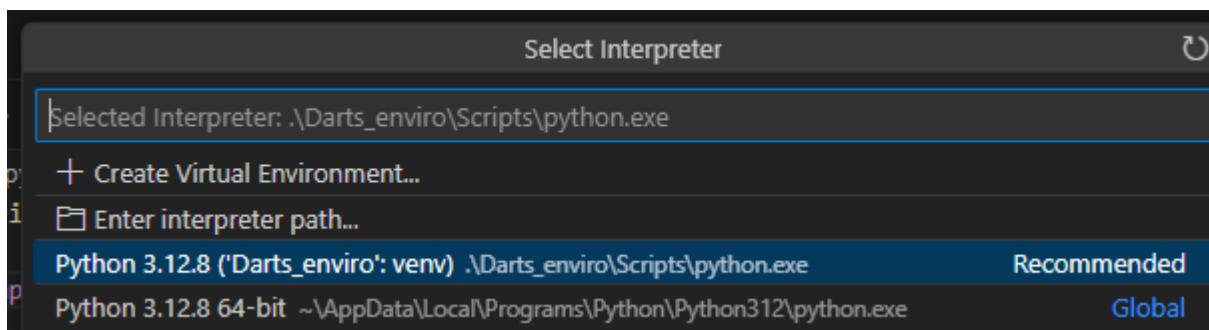
Además, vemos que se creó la carpeta del ambiente virtual. Esta carpeta contiene el ejecutable de Python además de las librerías instaladas dentro del ambiente virtual:

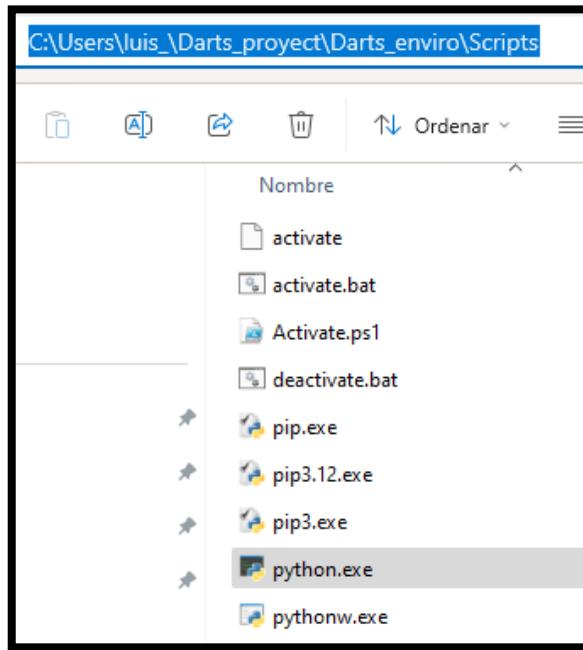


Vemos que dentro de la carpeta del ambiente virtual creado, tambien se crea el archivo `pyvenv.cfg`. Este archivo indica el python “base” y la ruta del ambiente virtual. Si lo abrimos:

```
home = C:\Users\luis_\AppData\Local\Programs\Python\Python312
include-system-site-packages = false
version = 3.12.8
executable = C:\Users\luis_\AppData\Local\Programs\Python\Python312\python.exe
command = C:\Users\luis_\AppData\Local\Programs\Python\Python312\python.exe -m venv C:\Users\luis_\Darts_proyect\Darts_enviro
```

Si en algun momento cambiamos el nombre de la carpeta que alberga el ambiente virtual, tendriamos tambien que modificar la ruta que aparece en este archivo (ultima linea). Si volvemos a visualizar la lista de intérpretes (**Ctrl + Shift + P**), vemos que ya existe la ruta del nuevo ejecutable de Python (junto al original, llamado “Global”):

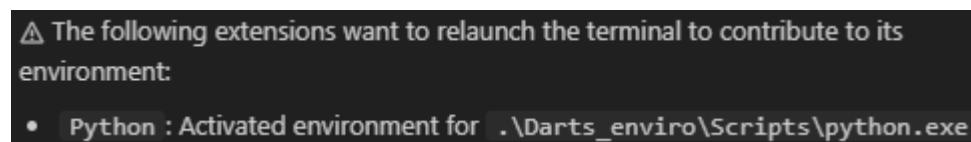




Además, vemos el siguiente mensaje en la terminal.



Si nos paramos sobre el sin dar click, vemos que nos indica que debemos reiniciar la terminal:



También podemos verificar parámetros sobre “Power Shell” del lado derecho de la terminal:



Una vez abierto el script y colocando el cursor sobre alguna línea de este, podemos de nuevo verificar el entorno observando la parte inferior derecha de VSC:



Si corremos un archivo de python, podemos ver en la terminal la ruta del ejecutable de python que se uso:

```

prueba.py x
prueba.py
1   print('Mensaje de prueba')
2   import numpy

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE PORTS
Python Debug Console + x ... x

PS C:\Users\luis_\Darts_proyect> & 'c:\Users\luis_\Darts_proyect\Scripts\python.exe' 'c:\Users\luis_\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '57349' '--' 'C:\Users\luis_\Darts_proyect\prueba.py'
● Mensaje de prueba
○ PS C:\Users\luis_\Darts_proyect> []

```

'c:\Users\luis_\Darts_proyect\Scripts\python.exe'

Si abrimos una nueva terminal, podemos verificar que ya se está usando el ejecutable de Python del ambiente virtual:

```

PS C:\Users\luis_\Darts_proyect> which python
/cygdrive/c/Users/luis_/Darts_proyect/Darts_enviro/Scripts/python
PS C:\Users\luis_\Darts_proyect>

```

NOTA: Hay veces que, aunque el ambiente virtual haya sido creado y este activado, la terminal y el ambiente virtual no se sincronizan correctamente. Por ejemplo, puede ser el caso en que, si queremos ver en la terminal la versión de Python que se está usando, hay veces que se muestra el ambiente global en lugar del ambiente virtual:

```

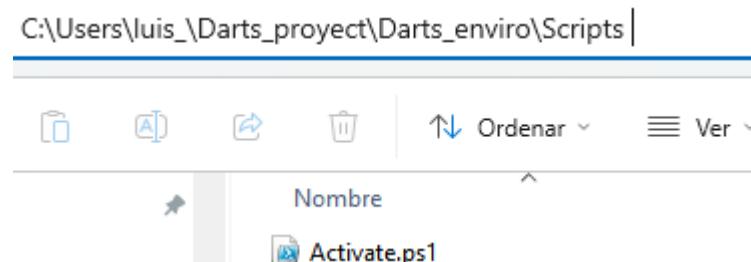
● PS C:\Users\luis_\Darts_proyect> which python
/cygdrive/c/Users/luis_/AppData/Local/Programs/Python/Python312/python

```

Pero vemos que el ambiente virtual si está activo:

Ln 4, Col 1 Spaces: 4 UTF-8 CRLF {} Python 3.12.8 ('Darts_enviro': venv)

Para sincronizar la terminal con el ambiente virtual, podemos correr el archivo "Activate.ps1" que se ubica dentro de la carpeta de ambiente virtual, en la terminal:



Ejecutamos en la terminal:

```

● PS C:\Users\luis_\Darts_proyect> C:\Users\luis_\Darts_proyect\Scripts\Activate.ps1

```

Puede ser que la 1º vez marque el siguiente error:

```
PS C:\Users\luis_\Darts_proyect> C:\Users\luis_\Darts_proyect\Darts_enviro\Scripts\Activate.ps1
C:\Users\luis_\Darts_proyect\Darts_enviro\Scripts\Activate.ps1 : No se puede cargar el archivo
C:\Users\luis_\Darts_proyect\Darts_enviro\Scripts\Activate.ps1 porque la ejecución de scripts está deshabilitada en este sistema. Para obtener
más información, consulta el tema about_Execution_Policies en https://go.microsoft.com/fwlink/?LinkID=135170.
En línea: 1 Carácter: 1
+ C:\Users\luis_\Darts_proyect\Darts_enviro\Scripts\Activate.ps1
+ ~~~~~
+ CategoryInfo          : SecurityError: (:) [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess
PS C:\Users\luis_\Darts_proyect>
```

Para solucionarlo, corremos VSC como administrador:



Y posteriormente ejecutamos la siguiente línea:

```
○ PS C:\Users\luis_\Darts_proyect> Set-ExecutionPolicy Unrestricted
```

Si lo volvemos a intentar, vemos que el ambiente virtual ya está activo y sincronizado con la terminal:

```
● PS C:\Users\luis_\Darts_proyect> C:\Users\luis_\Darts_proyect\Darts_enviro\Scripts\Activate.ps1
○ (Darts_enviro) PS C:\Users\luis_\Darts_proyect>
```

Vemos que ahora si se muestra la ruta del ejecutable de Python que corresponde al ambiente virtual:

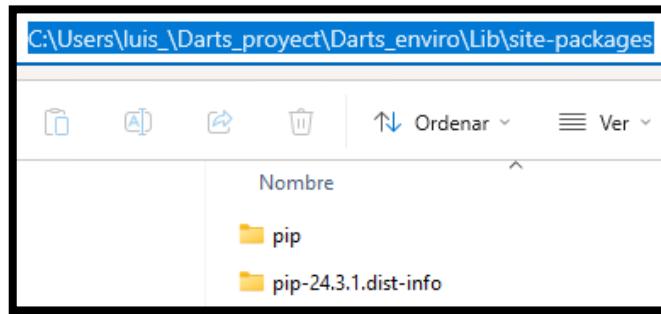
```
● (Darts_enviro) PS C:\Users\luis_\Darts_proyect> which python
/cygdrive/c/Users/luis_/Darts_proyect/Darts_enviro/Scripts/python
```

Ahora podemos instalar las librerías. Por ejemplo, podemos verificar las librerías o paquetes instalados mediante:

```
PS C:\Users\luis_\Darts_proyect> pip list
Package Version
-----
pip    24.3.1
PS C:\Users\luis_\Darts_proyect>
```

Las cuales se ubican en:





Antes de instalar alguna librería, podemos actualizar **pip** de Python mediante:

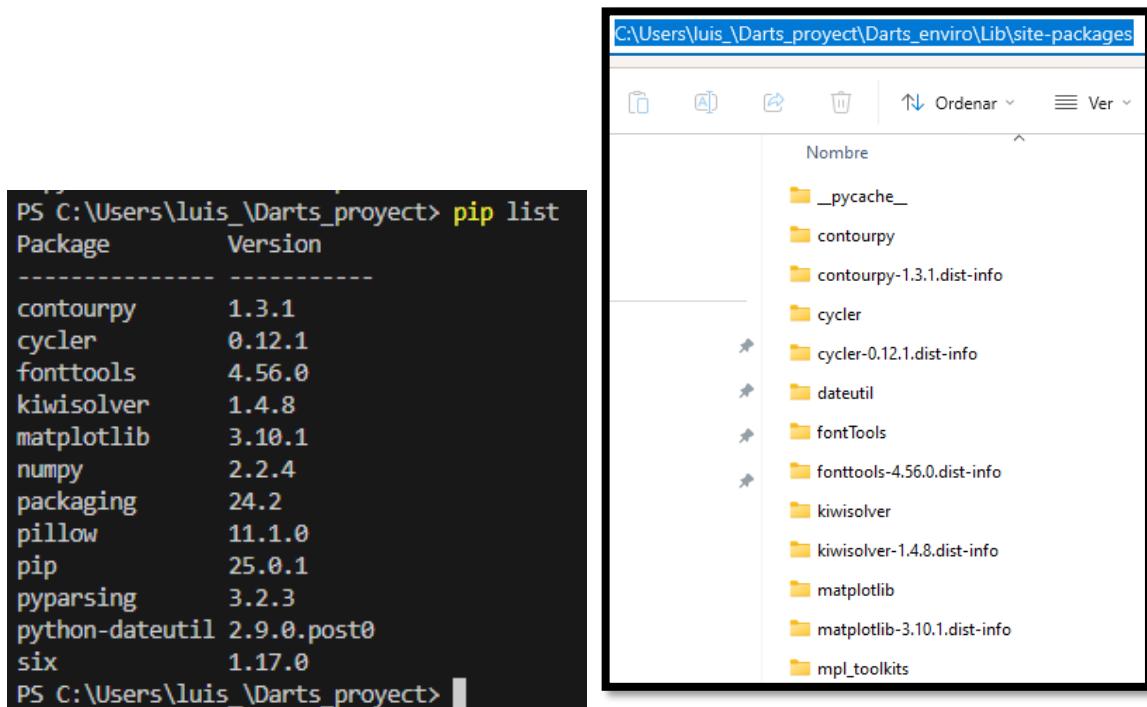
```
PS C:\Users\luis_\Darts_proyect> python.exe -m pip install --upgrade pip
Requirement already satisfied: pip in c:\users\luis_\darts_proyect\darts_enviro\lib\site-packages (24.3.1)
Collecting pip
  Using cached pip-25.0.1-py3-none-any.whl.metadata (3.7 kB)
Using cached pip-25.0.1-py3-none-any.whl (1.8 MB)
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 24.3.1
    Uninstalling pip-24.3.1:
      Successfully uninstalled pip-24.3.1
Successfully installed pip-25.0.1
```

Ahora sí, podemos instalar las librerías necesarias. Por ejemplo, instalamos **numpy** y **matplotlib**

```
PS C:\Users\luis_\Darts_proyect> pip install numpy
Collecting numpy
  Downloading numpy-2.2.4-cp312-cp312-win_amd64.whl.metadata (60 kB)
  Downloading numpy-2.2.4-cp312-cp312-win_amd64.whl (12.6 MB)
    12.6/12.6 MB 4.3 MB/s eta 0:00:01
```

```
PS C:\Users\luis_\Darts_proyect> pip install matplotlib
Collecting matplotlib
  Using cached matplotlib-3.10.1-cp312-cp312-win_amd64.whl.metadata (11 kB)
Collecting contourpy>=1.0.1 (from matplotlib)
  Using cached contourpy-1.3.1-cp312-cp312-win_amd64.whl.metadata (5.4 kB)
```

Verificamos mediante **pip list**. Vemos además que las librerías ya se ubican dentro de la ruta **\Lib\site-packages** del ambiente virtual:



Si volvemos a correr el script de prueba, vemos que no hay problemas:

The screenshot shows a VS Code interface with a Python file named `Prueba.py` open. The code contains:

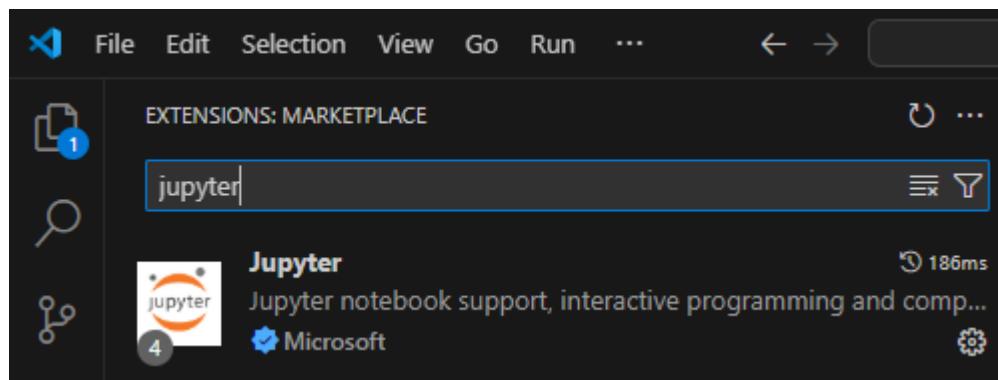
```
import numpy as np
arreglo=np.array([1,2,3,4])
print(arreglo)
```

The terminal below shows the execution of the script:

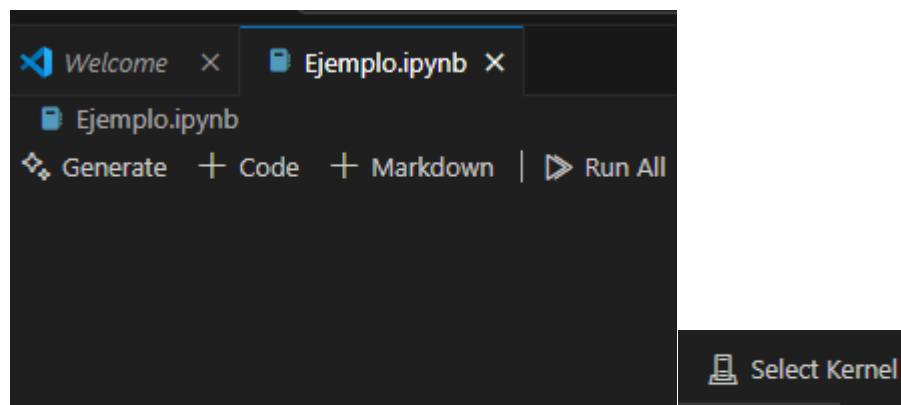
```
PS C:\Users\luis_\Darts_proyect> & 'c:\Users\luis_\Darts_proyect\on.debugpy-2025.4.1-win32-x64\bundled\libs\de
Mensaje de prueba
PS C:\Users\luis_\Darts_proyect> ^C
PS C:\Users\luis_\Darts_proyect>
PS C:\Users\luis_\Darts_proyect> c;; cd 'c:\Users\luis_\.vscode\extensions\ms-python.debug\Prueba.py'
Mensaje de prueba
[1 2 3 4]
PS C:\Users\luis_\Darts_proyect>
```

Jupyter notebook (opcional)

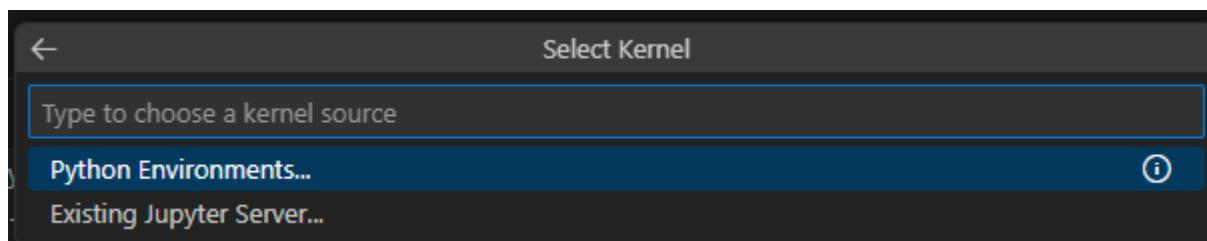
Si queremos usar un archivo de Jupyter notebook, primero debemos instalar la extensión correspondiente:

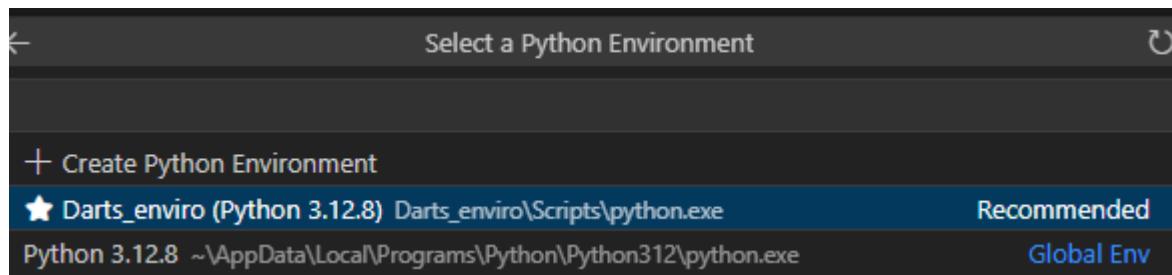


Una vez instalada, creamos un archivo con extensión (.ipynb). Antes que nada, debemos seleccionar un kernel usando la opción de lado derecho:

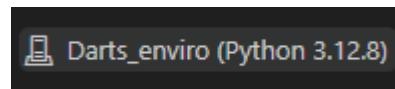


Nos pedirá escoger un intérprete o enviroment:

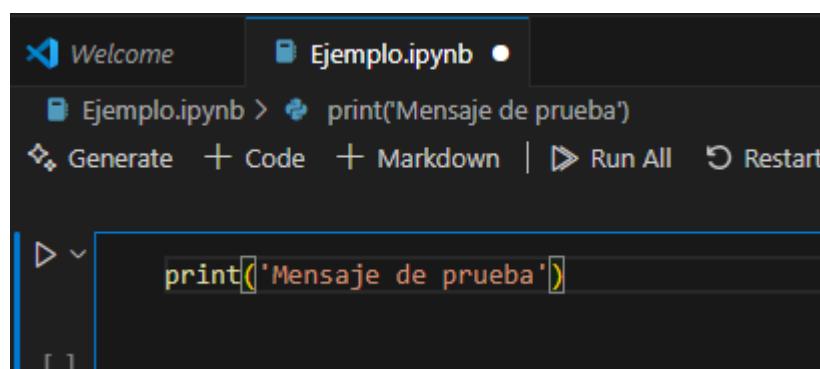




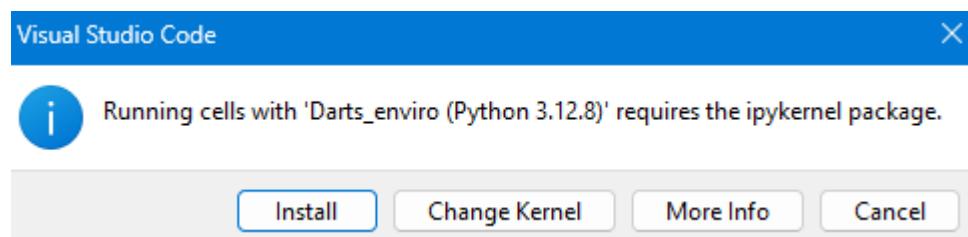
Una vez seleccionado el kernel, vemos que se ha actualiza el nombre del kernel de lado derecho:



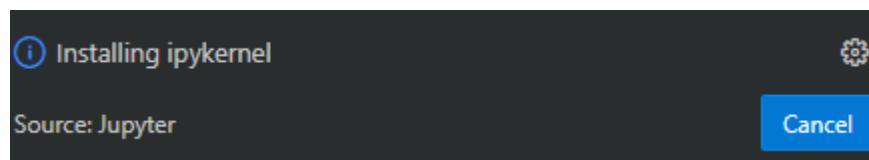
el siguiente paso es correr una celda. Agregamos una celda de código mediante **+ Code**:



Al intentar correr una celda la primera vez, nos pedirá si queremos instalar el paquete o librería de jupyter notebook:

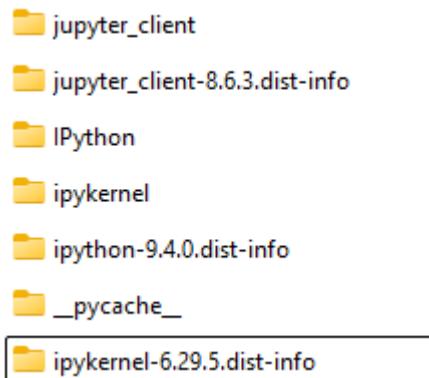


Al dar click en sí, veremos que se empieza a instalar.



Podemos corroborar que se haya instalado en la lista de paquetes

```
C:\Users\luis_\Darts_proyect\Darts_enviro\Lib\site-packages
```

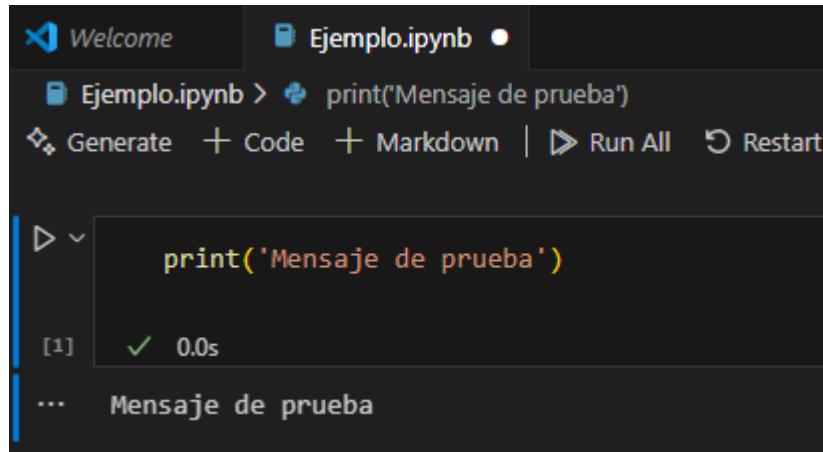


O usando de nuevo el comando

```
PS C:\Users\luis_\Darts_proyect> pip list
          Package           Version
-----
```

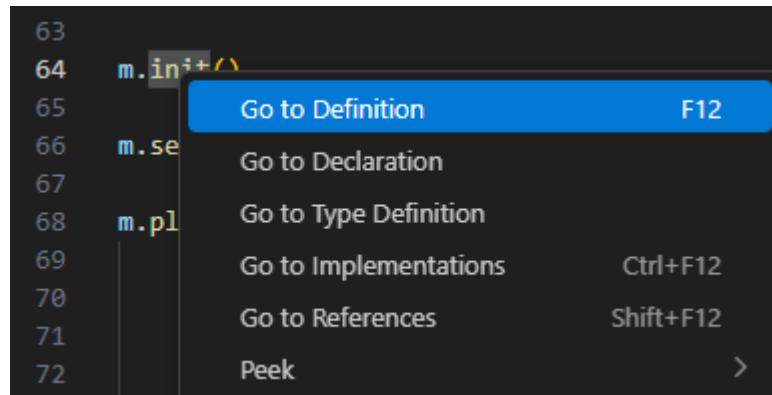
jupyter_client	8.6.3
jupyter_core	5.8.1

Al finalizar se correrá el código de la celda:



JSON

En esta sección presentamos como configurar un archivo de tipo **JSON**, la cual, aunque no es necesaria, es recomendada para poder conocer a fondo la forma en que funciona DARTS. Si queremos conocer detalle sobre una clase, método o función creada mediante DARTS, podemos dar click derecho sobre la propiedad que queremos conocer y posteriormente dar click en **Go to definition**. Por ejemplo, si queremos saber qué hace exactamente la función **init** del objeto **m** (el objeto principal en una simulación de DARTS):



Esto abrirá el archivo y la línea donde se define:

```

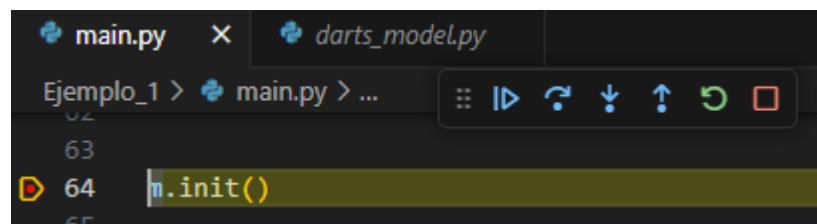
def __init__(self): ...

def init(
    self,
    descr_type: str = 'tpfa',
    platform: str = 'cpu',
    restart: bool = False,
    verbose: bool = False,
    itor_mode: str = 'adaptive',
    itor_type: str = 'multilinear',
    is_barycentric: bool = False,
):
    """
        Function to initialize the model, which includes:
        - initialize well (perforation) position
        - initialize well rate parameters
        - initialize reservoir initial conditions
        - initialize well control settings
        - define list of operator interpolators for accumulation-flux regions and wells
        - initialize engine
    """

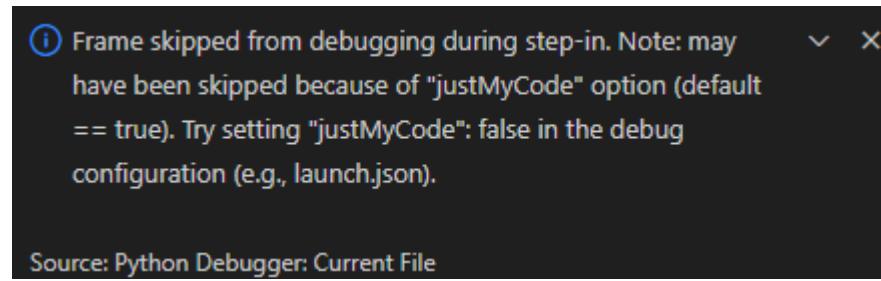
```

Para poder explorar las clases y funciones de DARTS en tiempo de ejecución, podemos agregar un **breakpoint** sobre la línea que queremos investigar. Al correr el script automáticamente parara en la línea de **breakpoint**.

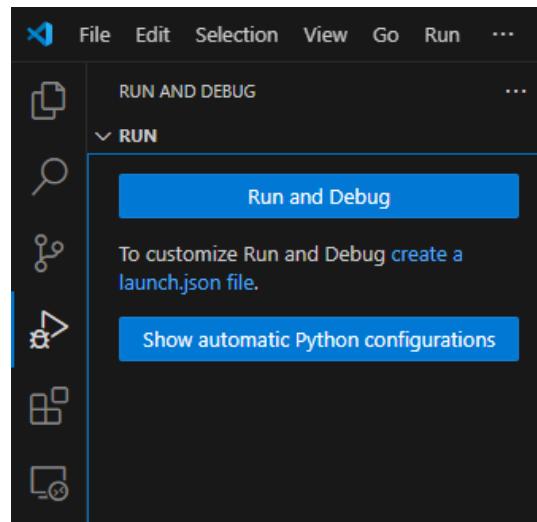
En el menú de la derecha, damos click en



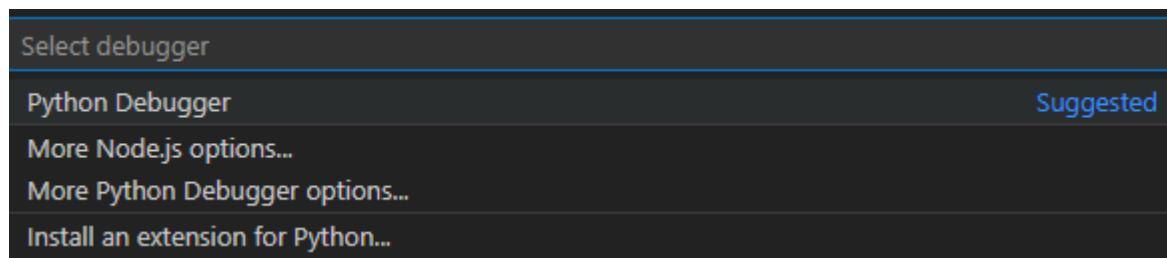
Sin embargo, marcará un error/advertencia, indicando que solo podemos explorar dentro de las funciones de nuestro script:



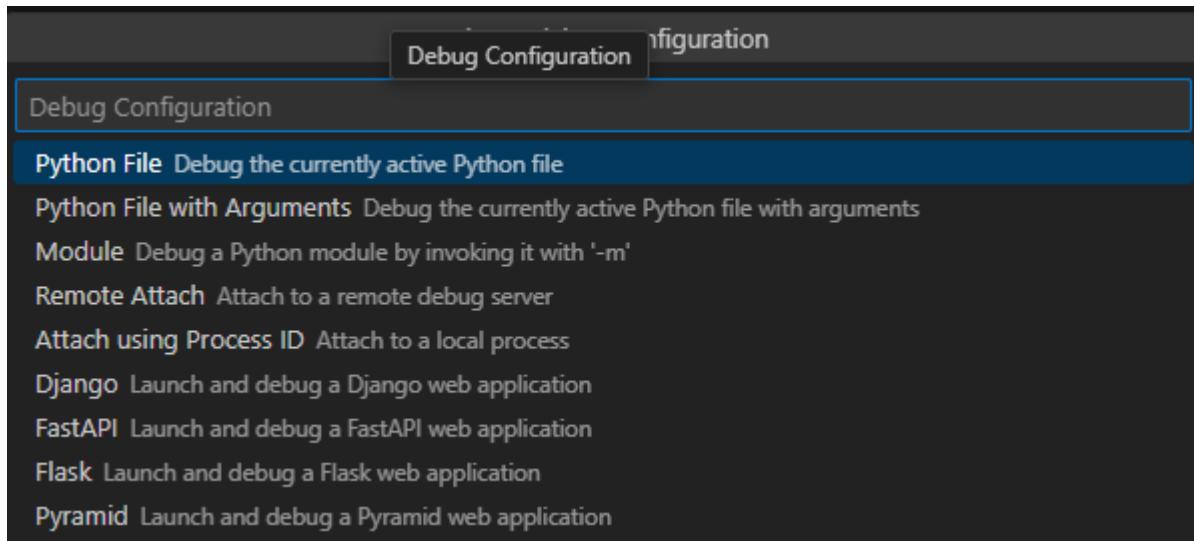
Para cambiar esto, tenemos que desactivar la opción **justmycode** mediante el archivo **Launc.json**. Para esto, primero debemos crear dicho archivo. Damos click del lado izquierdo en la opción de **Debugger**. Luego, damos click en **create a launch.json file**



Seleccionamos el debugger de python:

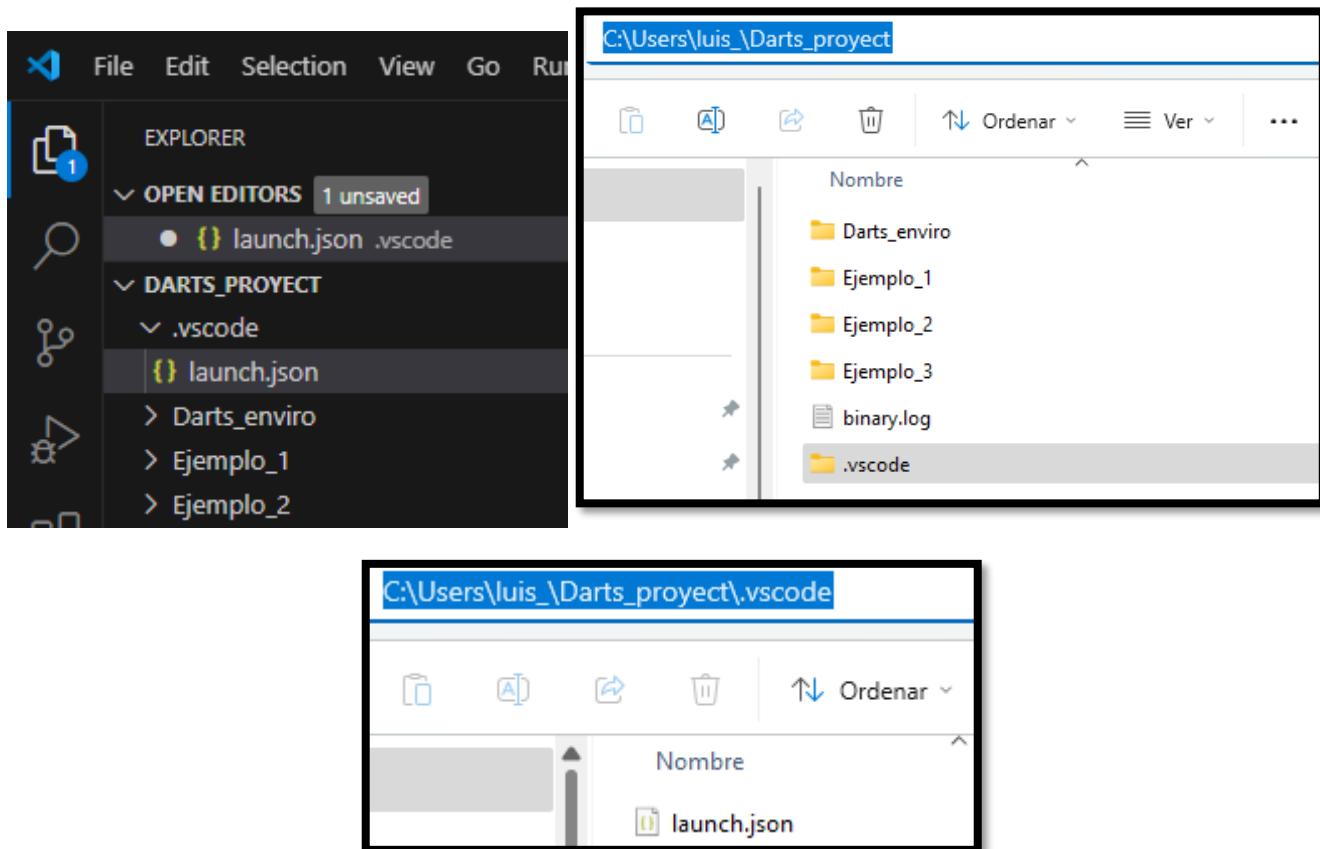


Y damos click en la primera opción:



Esto creará y abrirá el archivo **launch.json** dentro de nuestro proyecto:

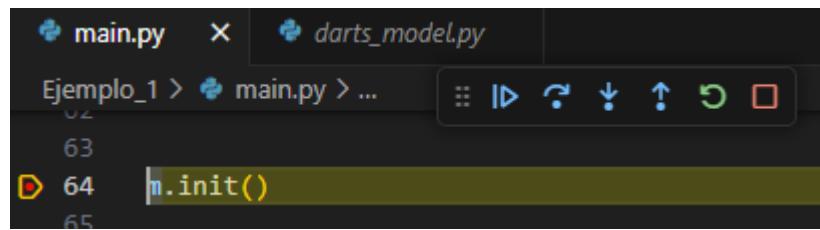
```
{} launch.json ● .vscode > {} launch.json > JSON Language Features > [ ] configurations
1  {
2      // Use IntelliSense to learn about possible attributes.
3      // Hover to view descriptions of existing attributes.
4      // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
5      "version": "0.2.0",
6      "configurations": [
7          {
8              "name": "Python Debugger: Current File",
9              "type": "debugpy",
10             "request": "launch",
11             "program": "${file}",
12             "console": "integratedTerminal"
13         }
14     ]
15 }
16 }
```



Agregamos la siguiente línea para desactivar la opción **justmycode** y guardamos el archivo:

```
{} launch.json ●
.vscode > {} launch.json > Launch Targets > {} Python Debugger: Current File
1  {
2      // Use IntelliSense to learn about possible attributes.
3      // Hover to view descriptions of existing attributes.
4      // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
5      "version": "0.2.0",
6      "configurations": [
7
8          {
9              "name": "Python Debugger: Current File",
10             "type": "debugpy",
11             "request": "launch",
12             "program": "${file}",
13             "console": "integratedTerminal",
14             "justMyCode": false
15         }
16     ]
17 }
```

Con esto, al momento de debuggear podemos acceder a las funciones y clases de las librerías de DARTS (y no solo a las funciones de nuestro script). Repetimos



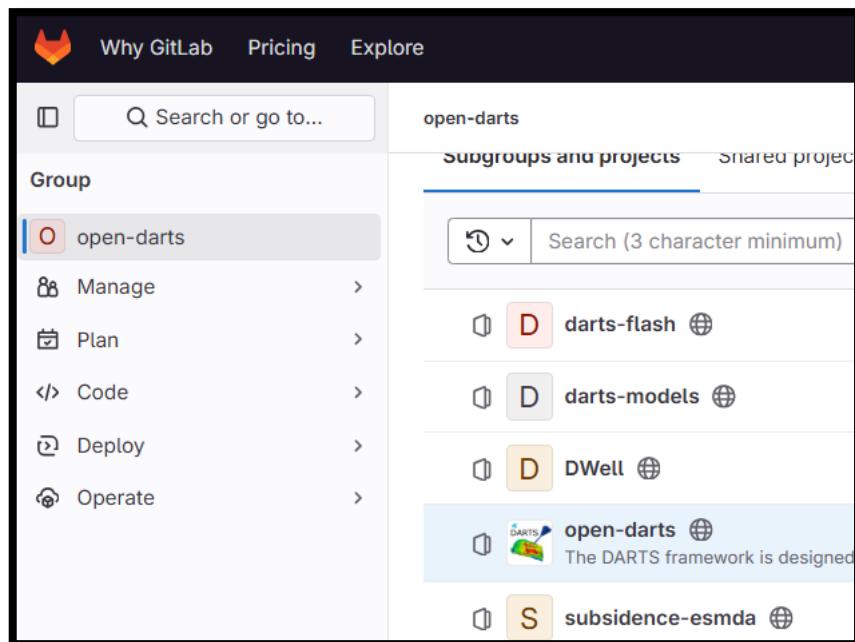
Si damos click en , podemos acceder a las clases, funciones y métodos en tiempo de ejecución.

DARTS

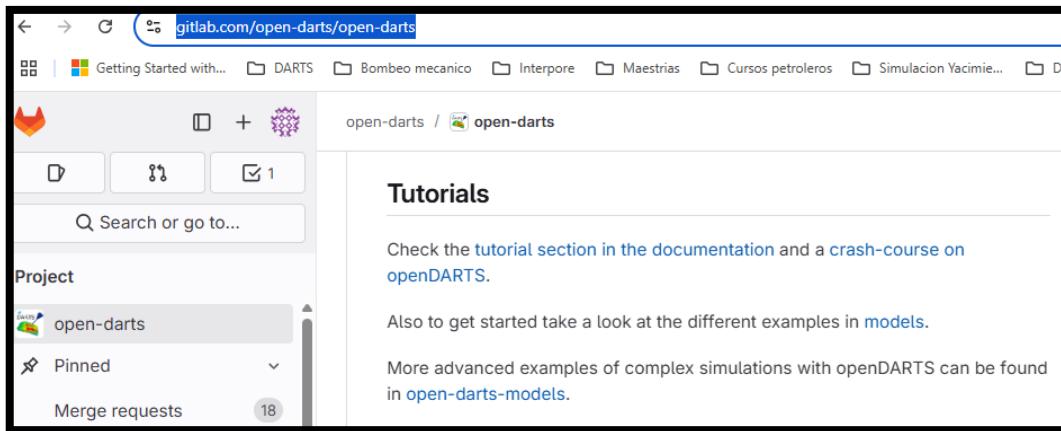
El siguiente paso es instalar DARTS mediante [pip install open-darts](#):

```
PS C:\Users\luis_\Darts_proyect> pip install open-darts
Collecting open-darts
  Downloading open_darts-1.2.2-cp312-cp312-win_amd64.whl.metadata (3.9 kB)
Requirement already satisfied: numpy in c:\users\luis_\darts_proyect\darts_enviro\lib\site-packages (from open-darts) (2.2.4)
Collecting numba (from open-darts)
  Downloading numba-0.61.0-cp312-cp312-win_amd64.whl.metadata (2.8 kB)
```

Una vez instalado podemos empezar usarlo. En el repositorio de Gitlab vienen varios ejemplos:



Por ejemplo:



Ejemplos sobre CCS (Captura y almacenamiento de CO₂):

https://gitlab.com/open-darts/darts-models/-/tree/development/teaching/CCS_workshop?ref_type=heads

Ejemplos sobre Geotermia):

https://gitlab.com/open-darts/darts-models/-/tree/development/publications/24_geothermal_chapter?ref_type=heads

https://gitlab.com/open-darts/darts-models/-/tree/development/teaching/EAGE?ref_type=heads

En las versiones más nuevas de DARTS, la librería DARTS-Flash se instala automáticamente cuando se instala DARTS, por lo que no se tiene que instalar por separado usando **pip**. Esta librería se ubica en **Darts_enviro\Lib\site-packages**



Algunos ejemplos que utilizan esta librería se pueden encontrar en los siguientes enlaces:

<https://pypi.org/project/open-darts-flash/>

https://gitlab.com/open-darts/darts-flash/-/blob/development/README.md?ref_type=heads

Gmsh y Paraview

Como se mencionó anteriormente, **Gmsh** y **Paraview** son dos herramientas que no son estrictamente necesarias para poder utilizar **DARTS** pero se recomienda su uso, por lo que varios de los ejemplos de esta guía utilizan dichos softwares. En la sección de enlaces se muestran los enlaces donde se puede descargar e instalar ambas herramientas. Aunque esta guía no requiere tener experiencia previa con estas dos herramientas, si se recomienda que el lector revise la documentación y se familiarice con su uso.

Estructura y ejemplos

Para utilizar DARTS, es necesario configurar el modelo. Esto generalmente incluye algunos scripts de Python, llamados por lo general **main.py** y **model.py**.

- **main.py** suele gestionar la inicialización de un modelo, la ejecución de la simulación, el avance en el tiempo y los tipos de salida, como archivos **vtk**, datos temporales en archivos de Excel o gráficos.
- **model.py** define el dominio de la simulación, también llamado **reservorio**, la malla computacional y las propiedades estáticas asignadas a ellos. Además, contiene la descripción de la física y todos los parámetros necesarios para definirla.

Por lo general, los archivos están organizados de manera que **main.py** actúa como un script de inicio donde se crea el modelo. Luego, llama al constructor del modelo o a otros métodos sobrecargados en **model.py**, que posteriormente inicializan el modelo con un **reservorio**, una **física** asociada al reservorio, una **condición inicial** y una **condición de frontera**. También pueden existir archivos adicionales utilizados para inicializar un reservorio, por ejemplo, el archivo de distribución de permeabilidad. También puede incluir una carpeta de mallas o simplemente archivos **.geo**, **.msh**, **.grdecl** colocados en la carpeta del proyecto. Estos archivos definen las **mallas computacionales** utilizadas en el modelo. Despues de ejecutar la simulación, el programa generará varios archivos de salida en la carpeta del proyecto, incluyendo:

- **Carpeta vtk**: Contiene la solución dinámica de **temperatura** y **presión** en todas las celdas de la malla, por ejemplo, archivos *solution_ts.vts**, así como la malla de las celdas, e.g. **mesh.vts**.
- **darts_time_data.pkl**: Archivo en formato **pickle** que almacena los resultados dinámicos de la simulación, incluyendo **presión en fondo de pozo (BHP)**, **temperatura**, **tasas de inyección/producción de agua**, y el **volumen acumulado de agua producida** en los pozos de inyección y producción durante el tiempo de simulación.
- **time_data.xlsx**: Archivo de **Excel** con los resultados dinámicos de la simulación, incluyendo **BHP**, **temperatura**, **tasas de inyección/producción de agua** y el **volumen acumulado de agua producida** en los pozos de inyección y producción.

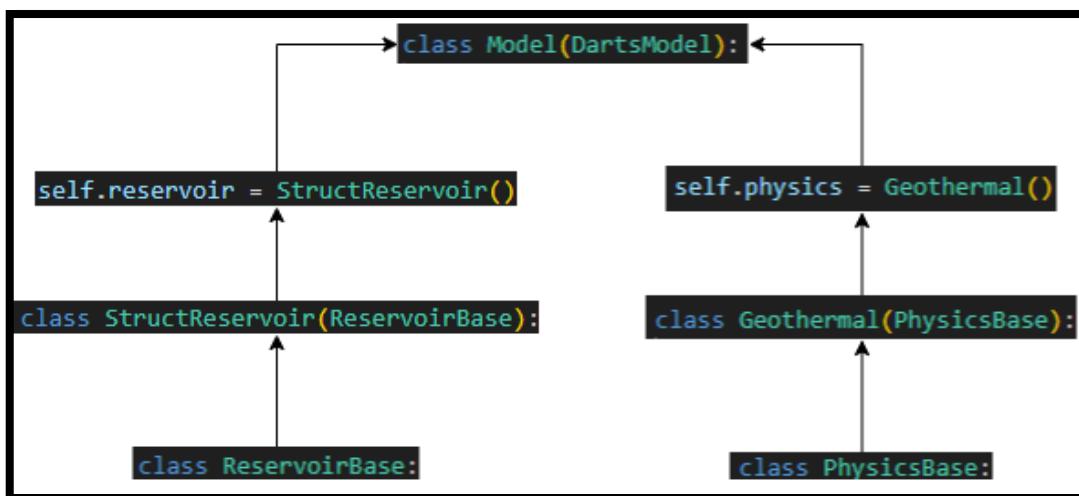
Ejemplo 1

En este ejemplo se mostrarán los elementos básicos que se requieren para correr una simulación en DARTS. Comenzaremos con explicar el contenido del archivo [model.py](#). El primer paso es cargar las librerías, modelos y clases propias de DARTS, incluyendo:

- La clase [DartsModel](#), que sirve como base para crear nuestro modelo.
- Las clases para crear un objeto de tipo [reservoir](#). ([StructReservoir](#) en este caso).
- Las clases para crear un objeto de tipo [Physics](#). ([Geothermal](#) en este caso).
- La clase [PropertyContainer](#), para calcular propiedades secundarias.

```
import numpy as np
from darts.models.darts_model import DartsModel
from darts.reservoirs.struct_reservoir import StructReservoir
from darts.physics.geothermal.physics import Geothermal
from darts.physics.geothermal.property_container import PropertyContainer
from darts.engines import value_vector, sim_params, well_control_iface
```

El siguiente paso es crear el modelo. La forma en que funciona DARTS es mediante clases y mediante el concepto de herencia. **La herencia** en Python es un principio de la programación orientada a objetos (OOP) que permite a una clase (llamada **clase hija** o **subclase**) heredar atributos y métodos de otra clase (llamada **clase padre** o **superclase**). Por ejemplo, en este caso se crea la clase [Model](#), la cual se crea a partir de la clase padre [DartsModel](#). La clase [DartsModel](#), a su vez, utiliza otras clases para crear el objeto deseado. Por ejemplo, si se desea simular un yacimiento geotérmico con malla estructurada, se utilizarán las clases [StructReservoir](#) y [Geothermal](#) para crear características del objeto de tipo [Model](#). A su vez, estas dos clases se crean mediante clases padre o base: [ReservoirBase](#) y [PhysicsBase](#), respectivamente.



De esta forma, se define la clase [Model](#) a partir de la clase base (o clase padre) [DartsModel](#)

```
class Model(DartsModel): ...
```

Este modelo tendrá varias funciones para establecer los parámetros principales de la simulación. La mayoría de estas funciones son heredadas a partir de las clases padre, por lo que, más que crearlas, serán modificadas:

```

< class Model(DartsModel):
>     def __init__(self, n_points=64): ...
>     def set_physics(self, n_points): ...
>     def set_initial_conditions(self): ...
>     def set_reservoir(self): ...
>     def set_wells(self): ...
>     def set_well_controls(self): ...
>     def plot_init_phase_with_p_t(self,P_ini,T_ini,P_iny,T_iny,P_prod,T_prod): ...

```

`def __init__(self, n_points=64):`: Aquí se definen todos los argumentos que tendrá un objeto de tipo **Model** al ser inicializado. Como argumento de entrada se define el número de puntos que se utilizaran en el método OBL.

```

self.timer.node["initialization"].start() # se inicia un contador de tiempo
                                         # para determinar el tiempo de inicializacion

self.set_reservoir()                      # Se llama la funcion "set_reservoir" (definida mas abajo)
                                         # con la cual se establecen las caracteristicas del yacimiento
self.set_physics(n_points)                # Se llama la funcion "set_physics" (definida mas abajo)
                                         # con la cual se establece la fisica del problema

```

Se definen algunos parámetros de la simulación. **Algunos valores se pueden modificar más adelante**

```

                                         # 1 [s]= 1.1574e-5 [days]
self.set_sim_params(first_ts=1.1574e-5, mult_ts=2, max_ts=100, runtime=3650, tol_newton=1e-2, tol_linear=1e-4,
                     it_newton=20, it_linear=40, newton_type=sim_params.newton_global_chop,
                     newton_params=value_vector([1]))

```

DARTS utiliza días para las unidades de tiempo.

- first_ts= tamaño del primer paso de tiempo
- mult_ts= Multiplicador de paso de tiempo
- max_ts= Tamaño máximo para el paso de tiempo
- runtime= Tiempo de simulación
- tol_newton= Tolerancia al método de Newton
- tol_linear= Tolerancia al solucionador lineal
- it_newton= máximo número de iteraciones del método de newton
- it_linear= máximo número de iteraciones del resolvedor lineal

- newton_type= ??
- newton_params= ??

`def set_physics(self, n_points):` Se define la física del problema. Se establecen los límites o rango de las variables principales usadas en el método OBL. Las unidades deben de ser en [bar] para la presión y [KJ/Kmol] para la entalpia.

```
self.min_p=0.          #  bar
self.max_p=500.        #  bar
self.min_e= 1.*18.01528 # KJ/kg ---- KJ/Kmol
self.max_e= 3200.*18.01528 # KJ/kg ---- KJ/Kmol
```

Se utiliza la clase **Geothermal** para definir la física del modelo

```
self.physics = Geothermal(self.timer, n_points=n_points,
                           min_p=self.min_p, max_p=self.max_p,
                           min_e=self.min_e, max_e=self.max_e, cache=False)
```

Se crea un objeto de la clase **PropertyContainer**. Se especifican las propiedades secundarias que queremos se calculen durante la simulación, a partir de las variables principales (presión y entalpia). Posteriormente, se asigna el objeto a la física del problema

- W=agua (water)
- S=vapor (steam)

```
property_container = PropertyContainer()

property_container.output_props = {'Enthalpy_W [KJ/kmol]': lambda: property_container.enthalpy[0],
                                   'Enthalpy_S [KJ/kmol]': lambda: property_container.enthalpy[1],
                                   'Enthalpy_W [KJ/kg]': lambda: property_container.enthalpy[0]/18.01528,
                                   'Enthalpy_S [KJ/kg]': lambda: property_container.enthalpy[1]/18.01528,
                                   'temp [K]': lambda: property_container.temperature,
                                   'temp [°C]': lambda: property_container.temperature-273.15,
                                   'sat_W': lambda: property_container.saturation[0],
                                   'sat_S': lambda: property_container.saturation[1]
                                  }

self.physics.add_property_region(property_container)
```

Nota: la entalpia usada como variable principal en la simulación es la entalpia de la mezcla (kJ/kmol) y se trata de la entalpia de la mezcla:

$$h_{\text{mezcla}} = (1 - x) h_f(T) + x h_g(T) = h_f(T) + x h_{fg}(T)$$

- $h_f(T)$: entalpía específica del líquido saturado (agua).
- $h_g(T)$: entalpía específica del vapor saturado.
- (Opcional) $h_{fg}(T) = h_g - h_f$.

X= fracción másica del vapor (no confundir con “sat_S”, que representa la fracción volumétrica del vapor)

Por otro lado, las entalpias calculadas como parte de las variables secundarias se refieren a la entalpía de cada fase.

El siguiente paso es establecer las condiciones iniciales del problema:

```
def set_initial_conditions(self): ...
    # initialization with constant pressure and temperature
    self.input_distribution = {"pressure": 150.,                      # [bar]
                               "temperature": 300. + 273.15  # [c]-----[k]
                               }

self.physics.set_initial_conditions_from_array(mesh=self.reservoir.mesh,
                                                input_distribution=input_distribution)
```

Aunque DARTS utiliza entalpia en lugar de temperatura, se puede inicializar el problema usando temperatura como dato de entrada y DARTS calculará la entalpia correspondiente usando este dato y el dato de presión. Para saber la entalpia correspondiente, podemos calcularla mediante el método `evaluate` de `enthalpy_ev` del objeto `property_containers`:

```
state_init = value_vector([input_distribution['pressure']])           # Initial presion   (bar)
enth_init = self.physics.property_containers[0].enthalpy_ev['total'].evaluate(state_init,
                                input_distribution['temperature'])
```

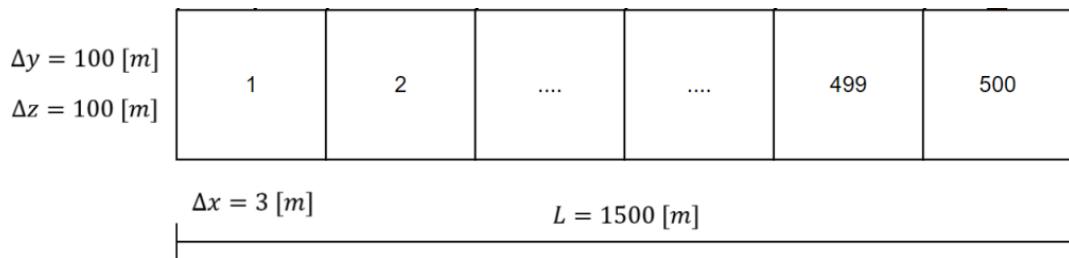
De igual forma, se evalúa la saturación de agua (porcentaje de agua o vapor) del estado inicial:

```
estado=np.array([state_init[0],enth_init ])
Sw_init=self.physics.property_containers[0].saturation_ev['water'].evaluate(estado)
```

Se imprimen las condiciones iniciales:

```
print('\n-----INITIAL CONDITIONS :')
print('\n')
print('Initial pressure (bar)=',input_distribution['pressure'])
print('Initial temperature (C)=',input_distribution['temperature']-273.15)
print('Initial enthalpy ([kJ/kg])=',enth_init/18.01528)
print('Initial enthalpy ([kJ/kmol])=',enth_init)
print('Initial Saturation (water)=',Sw_init)
print('\n')
```

def set_reservoir(self):: se define la geometría del yacimiento y sus propiedades. Para este primer ejemplo, se utiliza una sola dimensión y 500 celdas:



```
self.Long=1500. # [m]

(self.nx, self.ny, self.nz) = (500, 1, 1) # numero de celdas en cada direccion
dx=self.Long/self.nx

self.nb = self.nx * self.ny * self.nz      # numero de celdas totales
```



```
perm = np.ones(self.nb) * 1. # Se crea el arreglo que contiene los valores
# de permeabilidad para cada celda
perm[:self.nx*self.ny] =5. # Permeabilidad de 5 [mD] para todas las celdas
```

Debido a que es una geometría sencilla, se utiliza la clase **StructReservoir** para crear la geometría y discretizarlo. Para casos más complejos, se deberá utilizar una clase diferente. Se agregan otros valores de la roca, incluyendo la porosidad, el Calor específico de la roca y la Conductividad térmica de la roca:

```
self.reservoir = StructReservoir(self.timer, nx=self.nx, ny=self.ny, nz=self.nz, dx=dx, dy=100., dz=100.,
permx=perm, permy=perm, permz=perm, poro=.01, depth=1.,
hcap=2470., rcond= 172.8 )
```

hcap	Calor específico de la roca	2470 [kJ/(m ³ *K)]
rcond	Conductividad térmica de la roca	172.8 [KJ/(m*día*K)]

def set_wells(self): Función donde se especifican los pozos

- cell_index: ubicación de los pozos (se especifica el # de la celda donde se ubica)
- well_radius: radio del pozo
- skin: daño
- Well_index: Indicie de productividad del pozo. Si se omite, se calculará a partir del radio del pozo y de otros parámetros geoestéticos y físicos (utiliza la permeabilidad de la celda)

- Well_indexD: Indicie de productividad térmico del pozo. Si se omite, su valor por default es cero (significa que no hay perdida de calor). Para que sea calculado, se debe de establecer su valor como 'None'
 - segment_direction: direction in which the segment perforates the reservoir block (default: 'z_axis')

El índice de productividad (well index o productivity index) se define como:

$$q = -J(P_i - P_{wf})$$

Donde q es el gasto volumétrico. A su vez, P_1 se define como:

$$r_p = 0.28 * \frac{\left[\left(\frac{k_y}{k_x} \right)^{1/2} * dx^2 + \left(\frac{k_x}{k_y} \right)^{1/2} * dy^2 \right]^{1/2}}{\left(\frac{k_y}{k_x} \right)^{1/4} + \left(\frac{k_x}{k_y} \right)^{1/4}}$$

La tasa de transferencia de calor (Q) es la cantidad total de energía térmica transferida por unidad de tiempo. El flujo de calor entre el pozo y el yacimiento tiene un componente advectivo y otro conductivo, los cuales se discretizan de manera análoga.

$$Q = Q_a + Q_c$$

Donde

$$Q_c = r_{cond} * W.I.D(T - T_{well})$$

donde Wh es el índice térmico del pozo, calculado con la misma aproximación de Peaceman que WI, pero utilizando la conductividad térmica en lugar de la permeabilidad

$$W.I.D = \frac{2 * \pi * dz}{\log\left(\frac{r_c}{r}\right) + s}$$

$$r_c = 0.28 * \frac{(dx^2 + dy^2)^{1/2}}{2}$$

Finalmente queda que:

$$Q_c = \left[\frac{KJ}{\text{día}} \right] = r_{cond} * W.I.D * (T - T_{well}) = \left[\frac{KJ}{m * \text{día} * k} \right] [m][k]$$

(rcond)	Conductividad térmica de la roca	172.8 [KJ/(m*día*K)]
---------	----------------------------------	----------------------

La información del pozo se ubica en:

```

    m = <model.Model object at 0x00000215C2BF83B0>
    reservoir = <darts.reservoirs.struct_reservoir.StructReser...
    wells = [<darts.engines.ms_well object at 0x00000215C2C6B...
        0 = <darts.engines.ms_well object at 0x00000215C2C6BAB0>
        perforations = [(0, 0, 5.925441989782517, 0.0)]

```

```
well.perforations = (well_block, res_block_local, well_index, well_indexD)
```

Con las siguientes líneas se acceden a la información de cada pozo:

```

print('Well 1, block:', self.reservoir.wells[0].perforations[0][1])
print('Well 1, well_index:', self.reservoir.wells[0].perforations[0][2])
print('Well 1, well_indexD:', self.reservoir.wells[0].perforations[0][3])

print('Well 2, block:', self.reservoir.wells[1].perforations[0][1])
print('Well 2, well_index:', self.reservoir.wells[1].perforations[0][2])
print('Well 2, well_indexD:', self.reservoir.wells[1].perforations[0][3])

```

Para este ejemplo en específico, los valores calculados para índice de productividad (well index) son:

```
Well 1, block: 0
Well 1, well_index: 5.925441989782517
Well 1, well_indexD: 0.0
Well 2, block: 499
Well 2, well_index: 5.925441989782517
Well 2, well_indexD: 0.0
```

`def set_well_controls(self):` Aquí se definen los valores en los que opera cada pozo.

Inyector:

```
self.P_iny=200 # [bar]
self.inj_temp= 100 # [c]
```

Productor:

```
self.P_prod=100 # [bar]
```

Se utilizan los siguientes métodos para establecer los pozos (si el pozo lleva por nombre “INJ”, se asignará como un pozo inyector):

```
for i, w in enumerate(self.reservoir.wells):
    if 'INJ' in w.name:
        self.physics.set_well_controls(wctrl=w.control,
                                         control_type=well_control_iface.BHP,
                                         is_inj=True,
                                         target=self.P_iny,
                                         phase_name='water',
                                         inj_temp=self.inj_temp + 273.15 # [c]----[k]
                                         )
```

El pozo productor se define mediante una presión de fondo constante:

```
else:
    self.physics.set_well_controls(wctrl=w.control,
                                    control_type=well_control_iface.BHP,
                                    is_inj=False,
                                    target=self.P_prod
                                    )
```

Estos son todos los métodos disponibles para definir los pozos (en el caso de usar una física de tipo Geothermal):

- BHP [bar]
- MOLAR_RATE [kmol/day]
- MASS_RATE

- VOLUMETRIC_RATE
- ADVECTIVE_HEAT_RATE

Para plotear los datos de entrada y los pozos en el diagrama de presión-entalpia del agua, se utiliza una función propia:

```
def plot_init_phase_with_p_t(self,P_ini,T_ini,P_iny,T_iny,P_prod,T_prod)
```

Esta función utiliza la librería CoolProp:

```
from CoolProp.CoolProp import PropsSI
```

Para instalar esta librería:

```
pip install CoolProp
```

Una vez creado el modelo, el siguiente paso es cargarlo, inicializarlo y correrlo. Todos estos pasos se hacen a través del archivo [main.py](#), el cual explicaremos a detalle a continuación. El primer paso es cargar la clase **Model** del archivo model.py (debe de estar en la misma carpeta)

```
from model import Model # Se carga la clase "Model" del archivo model.py
# (debe de estar en la misma carpeta)
```

La siguiente función se utiliza para definir la forma en que se presentan los resultados de simulación:

```
from darts.engines import redirect_darts_output # ,value_vector
#redirect_darts_output('dfm_model.log') # change output style
#redirect_darts_output('binary.log') # change output style
```

Una vez importada la clase, se utiliza para crear un objeto:

```
m = Model()
```

Se pueden corroborar los parámetros más importantes de nuestro modelo antes de correr:

<pre>print('\n-----COMPONENTS, PHASES AND VARIABLES :') print('\n') print('Phases=',m.physics.phases) print('Number of phases=',m.physics.nph) print('Number of components=',m.physics.nc) print('Main variables=',m.physics.vars) print('Number of variables=',m.physics.n_vars) print('Number of operators=',m.physics.n_ops)</pre>	<pre>Phases= ['water', 'steam'] Number of phases= 2 Number of components= 1 Main variables= ['pressure', 'enthalpy'] Number of variables= 2 Number of operators= 10</pre>
---	---

Se inicializa el modelo:

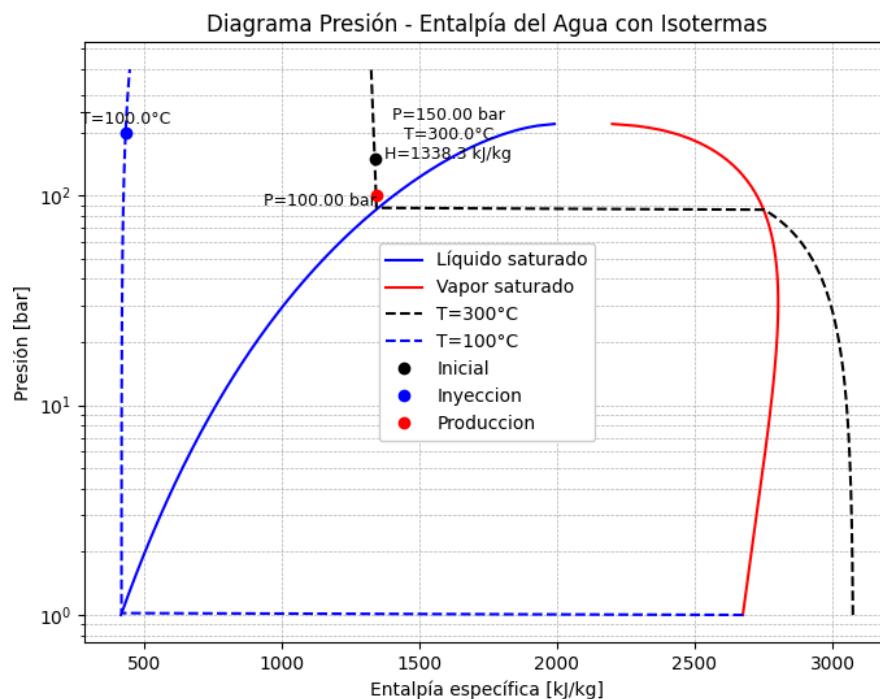
```
m.init()
```

Se define la carpeta de salida

```
m.set_output(output_folder='output')
```

Se llama la función para plotear el estado inicial, datos de entrada y los pozos en el diagrama de presión-entalpia del agua:

```
m.plot_init_phase_with_p_t( m.input_distribution['pressure'],
                             m.input_distribution['temperature'],
                             m.P_iny,
                             m.inj_temp + 273.15,
                             m.P_prod,
                             m.input_distribution['temperature'],
                             )
```



Se establece el tiempo de simulación:

```
end_time = 30 # End time of the simulation (years)
```

Se corre la simulación:

```
m.run(end_time*365) # End time of the simulation (days)
```

Al correr la simulación, mostrara en pantalla:

```
--RUN SIMULATION --

#1 (1.3986e-04, 8.6702e-03): lin 1 (0.0e+00)
-----
T = 1.1574e-05, DT = 1.1574e-05, NI = 1, LI = 1, RES = 6.7e-09 (2.3e-07), CFL=0.000 (ELAPSED 00:00:00 )
-----
# 1      T = 1.1574e-05  DT = 1.1574e-05  NI = 1  LI=1    DT_MULT= 2      dX=[0.404 0.619]
```

El primer paso de tiempo indica que hubo 1 iteración de Newton-Raphson, con una sola iteración lineal. También se muestra el tiempo total de simulación, el paso de tiempo usado y el residuo. Al ser el primer paso de tiempo, se utiliza el valor establecido previamente

- T=tiempo de simulación
- DT= paso de tiempo. Para este caso, se estableció como 1 segundo (1 segundo= 1.1574e-5 [days])
- NI= Newton – Raphson iterations
- LI= Lineal iterations
- RES= Residual
- DT_MULT= Multiplicador de paso de tiempo. Para este caso, se estableció como 2

Cuando el paso de tiempo alcanza el límite establecido (100 días en este caso), dejará de duplicarse y se mantendrá el valor límite. Por ejemplo:

```
T = 194.179, DT = 97.0897, NI = 2, LI = 2, RES = 7.9e-03 (5.3e-08), CFL=0.000 (ELAPSED 00:00:00 )
-----
# 24      T = 194.179      DT = 97.0897      NI = 2  LI=2    DT_MULT= 2      dX=[1.180000e-01 3.589551e+03]
          #1 (2.3519e-01, 8.2208e-06): lin 1 (0.0e+00)
          #2 (1.6526e+00, 3.8246e-06): lin 1 (0.0e+00)
          #3 (1.7432e-02, 4.9460e-08): lin 1 (0.0e+00)
-----
T = 294.179, DT = 100, NI = 3, LI = 3, RES = 7.1e-08 (9.6e-08), CFL=0.000 (ELAPSED 00:00:00 )
```

Al final de la simulación, indicará el número total de pasos de tiempo usados y de iteraciones:

TS= Total timesteps

```
T = 10950, DT = 55.8205, NI = 2, LI = 2, RES = 2.4e-06 (5.0e-08), CFL=0.000 (ELAPSED 00:00:01 )
-----
# 132      T = 10950      DT = 55.8205      NI = 2  LI=2    DT_MULT= 2      dX=[4.60000e-02 1.32386e+02]
TS = 132(0), NI = 246(0), LI = 246(0)
```

Los siguientes comandos imprimen en pantalla estadísticos de la simulación

```
m.print_timers()  
m.print_stat()
```

```
Total elapsed 49.212000 sec  
    initialization 0.004000 sec  
        connection list generation 0.001000 sec  
    newton update 0.006000 sec  
    output 1.449000 sec  
        exporting_property_array 0.000000 sec  
        output_well_time_data 0.000000 sec  
        saving_reservoir_data 0.032000 sec  
        saving_well_data 1.414000 sec  
        vtk_output 0.000000 sec  
simulation 1.255000 sec  
    jacobian assembly 1.010000 sec  
        interpolation 0.922000 sec  
            property 0 interpolation 0.000000 sec  
            reservoir 0 interpolation 0.952000 sec  
                body generation 0.906000 sec  
                point generation 0.906000 sec  
well controls interpolation 0.069000 sec  
    body generation 0.068000 sec  
        point generation 0.068000 sec  
well initialization 0.002000 sec  
    body generation 0.002000 sec  
        point generation 0.002000 sec  
well interpolation 0.053000 sec  
    body generation 0.053000 sec  
        point generation 0.053000 sec  
linear solver setup 0.001000 sec  
    SUPERLU 0.001000 sec  
linear solver solve 0.154000 sec  
    SUPERLU 0.135000 sec  
newton update 0.000000 sec
```

```
Total steps 132 (0) newton 246 (0) linear 246 (0)
---OBL Statistics---
Number of operators: 10
Number of points: 64
Number of interpolations: 1895000
Number of points generated: 129 (3.149%)
Total elapsed 1.255000 sec
    jacobian assembly 1.010000 sec
        interpolation 0.922000 sec
            property 0 interpolation 0.000000 sec
            reservoir 0 interpolation 0.952000 sec
                body generation 0.906000 sec
                    point generation 0.906000 sec
            well controls interpolation 0.069000 sec
                body generation 0.068000 sec
                    point generation 0.068000 sec
            well initialization 0.002000 sec
                body generation 0.002000 sec
                    point generation 0.002000 sec
            well interpolation 0.053000 sec
                body generation 0.053000 sec
                    point generation 0.053000 sec
linear solver setup 0.001000 sec
    SUPERLU 0.001000 sec
linear solver solve 0.154000 sec
    SUPERLU 0.135000 sec
newton update 0.000000 sec
    composition correction 0.000000 sec
```

Los resultados de la simulación se pueden obtener a partir de `m.physics.engine.X`, aunque solo contiene la información de las variables principales para cada una de las celdas de simulación (presión y entalpia, para este caso). Además, incluye datos de los pozos. Por ejemplo, para este caso en particular donde se tienen 500 celdas:

```
# con la siguiente linea se obtiene los resultados, pero solamente los
# que corresponden a las variables principales (presion y entalpia)
resultados= np.array(m.physics.engine.X, copy=False)
print(resultados.shape)
print(resultados.ndim)
```

Esto imprime:

```
(1008, )
1
```

Presión	500
Entalpia	500
Pozos	8
¿?	
Total	1008

Si visualizamos los valores del pozo, vemos que contiene valores de presión y entalpia de cada pozo:

```
print(resultados[-8:]) # Muestra los últimos 8 valores del array
```

```
[ 200.          7821.66771828  199.99952118  7821.66771997
 100.          24144.78593363  100.00019441  24144.78593363]
```

Para extraer los valores de presión y entalpia ($m.nb$ =número de celdas del modelo=500):

```
# Extraer los valores de presión (índices impares)
pressure = resultados[::m.nb*2:2] # Del índice 0 al 999, tomando cada 2 valores

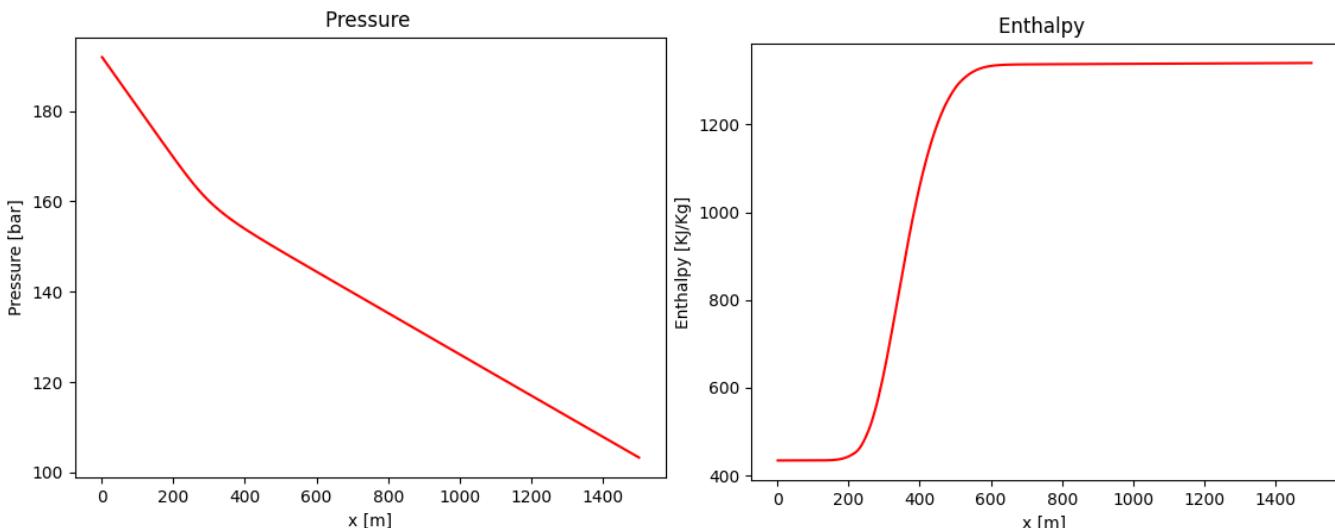
# Extraer los valores de enthalpy (índices pares)
enthalpy = resultados[1::m.nb*2:2] # Del índice 1 al 999, tomando cada 2 valores
```

Para plotear los valores de presión y entalpia:

```
x= np.linspace( m.Long/(2*m.nx), m.Long, num=m.nx)

plt.title("Pressure")
plt.plot(x, pressure, color="red")
plt.xlabel("x [m]")
plt.ylabel("Pressure [bar]")
plt.savefig('Pressure.png', format='png')
plt.show()

plt.title("Enthalpy")
plt.xlabel("x [m]")
plt.ylabel("Enthalpy [KJ/Kg]")
plt.plot(x, enthalpy / 18.015, color="red")
plt.savefig('enthalpy.png', format='png')
plt.show()
```



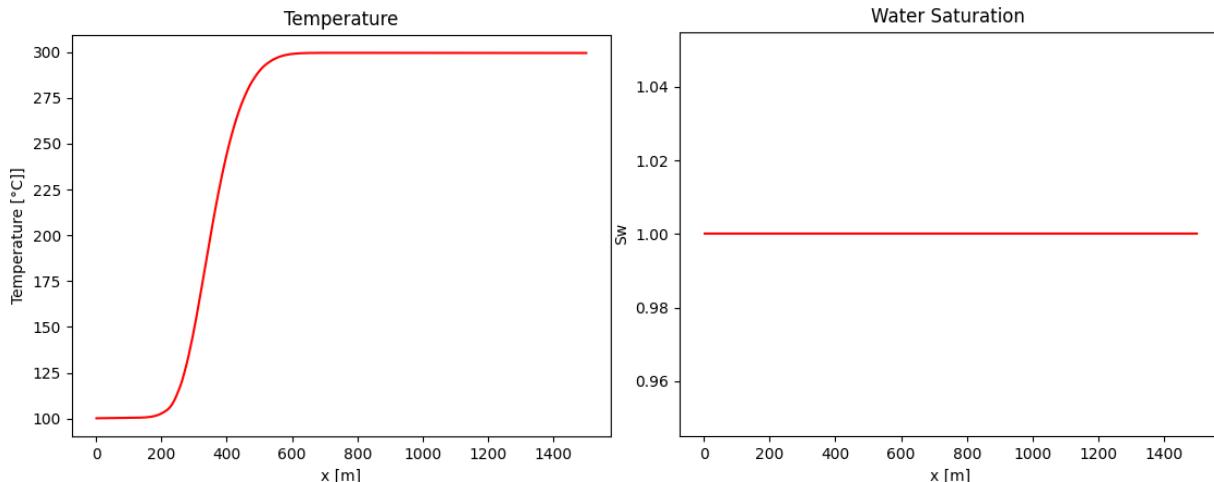
Las propiedades secundarias (definidas en “Model.py”) se obtienen de la siguiente forma:

```
tiempo_final, estado_final = m.output.output_properties(output_properties = m.physics.property_operators[0].props_name)
```

Para plotear:

```
plt.title("Temperature")
plt.xlabel("x [m]")
plt.ylabel("Temperature [°C]")
plt.plot(x, estado_final['temp [°C]'][1], color="red")
plt.savefig('Temperature.png', format='png')
plt.show()

plt.title("Water Saturation ")
plt.xlabel("x [m]")
plt.ylabel("Sw")
plt.plot(x, estado_final['sat_W'][1], color="red")
plt.savefig('Water Saturation.png', format='png')
plt.show()
```



Se pueden guardar los perfiles de presión y entalpia en un archivo Excel:

```
df = pd.DataFrame(pressure)
df2= pd.DataFrame(enthalpy/ 18.015)
writer = pd.ExcelWriter('Perfiles.xlsx')
df.to_excel(writer, sheet_name = 'Pressure')
df2.to_excel(writer, sheet_name = 'Enthalpy')
writer.close()
```

Con los siguientes comandos se escribe un archivo Excel que contiene la información resumida de los pozos, junto a la información de la celda que lo contiene (NOTA: Se han detectado errores en este archivo, revisarlo y manejarlo con cuidado)

```

td = pd.DataFrame.from_dict(m.physics.engine.time_data)
td.to_pickle("darts_time_data.pkl")
writer = pd.ExcelWriter('well_resume.xlsx')
td.to_excel(writer, 'Sheet1')
writer.close()

```

La columna ‘D’ muestra los pasos de tiempo (en días). En las columnas ‘F’, ‘I’, ‘P’ y ‘Q’ podemos ver los datos de la presión de fondo de ambos pozos y la presión del bloque que lo contiene. En la siguiente imagen se muestran los últimos tiempos.

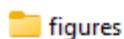
D	F	I	P	Q
time	INJ : BHP (bar)	INJ : p 0 reservoir P (bar)	PRD : p 0 reservoir P (bar)	PRD : BHP (bar)
10694.17949	200	191.8825278	103.2971737	100
10794.17949	200	191.8967157	103.2914206	100
10894.17949	200	191.910833	103.2856967	100
10950	200	191.9187857	103.2825087	100

Podemos ver que la presión de fondo y la presión de la celda difieren, debido al valor del índice de productividad. Para el pozo inyector, la presión de fondo tiene que ser mayor que la presión de la celda para que el fluido pueda ser inyectado en el yacimiento. En contraste, la presión de fondo del pozo productor resulta menor que la presión del yacimiento, ya que el fluido fluye hacia el pozo productor. Con las siguientes líneas podemos escribir una serie de archivos que contienen la información completa de los pozos (pero no contiene datos de ninguna celda):

```

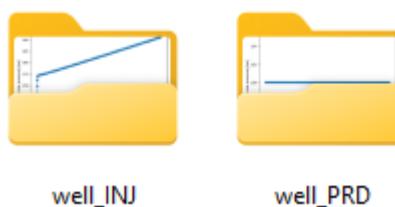
m.output.store_well_time_data( save_output_files=True)
m.output.plot_well_time_data()

```

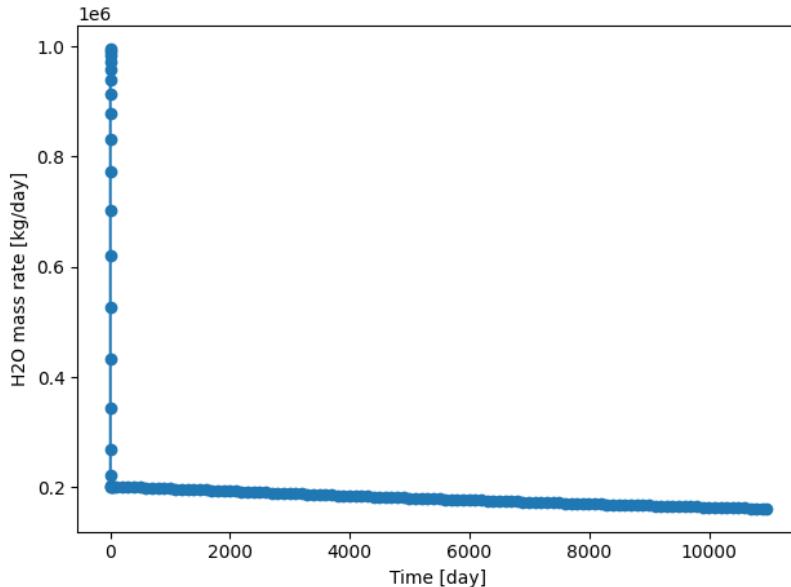


- reservoir_solution.h5
- well_data.h5
- well_time_data.pkl
- well_time_data.xlsx

Dentro de la carpeta ‘figures’ se crea una carpeta para cada pozo, la cual contiene las gráficas de diferentes variables ploteadas contra el tiempo



Por ejemplo, la gráfica de gasto masico vs tiempo para el pozo inyector:



Para este ejemplo, se establecerá el valor del índice de productividad para cada pozo (en lugar de que se calcule automáticamente). Para el pozo inyector, se establece un valor de 2, mientras que el pozo productor tendrá un valor de 100. (el valor calculado es de 5.925)

```
# add well
self.reservoir.add_well("INJ", wellbore_diameter=0.3048)
self.reservoir.add_perforation("INJ", cell_index=(1, 1, 1), skin=0.0,
| | | | well_radius=0.1524,well_index=2, multi_segment=False) #

# add well
self.reservoir.add_well("PRD", wellbore_diameter=0.3048)
self.reservoir.add_perforation("PRD", cell_index=(self.nb, 1, 1), skin=0.0,
| | | | well_radius=0.1524, well_index=100, multi_segment=False)
```

Si vemos la información de los pozos y las celdas:

D	F	I	P	Q
time	INJ : BHP (bar)	INJ : p 0 reservoir P (bar)	PRD : p 0 reservoir P (bar)	PRD : BHP (bar)
10694.17949	200	178.2016576	100.1771376	100
10794.17949	200	178.2328076	100.1768851	100
10894.17949	200	178.2638395	100.1766335	100
10950	200	178.2813522	100.1764938	100

Vemos que, respecto al ejemplo anterior, la diferencia de presiones en el pozo inyector aumentó (debido a que el valor del índice de productividad se redujo), mientras que la diferencia de presiones en el pozo productor disminuyó (ambas presiones son casi iguales), debido al alto valor del índice de productividad.

Ejemplo 2

También podemos correr un mismo modelo en repetidas ocasiones. Darts automáticamente guarda los resultados al final de la simulación y los utiliza como los datos de entrada para la siguiente corrida. Primero, creamos las gráficas de presión y entalpia.

```
x= np.linspace( m.Long/(2*m.nx), m.Long, num=m.nx)

# Crear figuras antes del bucle
fig1, ax1 = plt.subplots() # Figura para presión
ax1.set_title("Pressure")
ax1.set_xlabel("x [m]")
ax1.set_ylabel("Pressure [bar]")

fig2, ax2 = plt.subplots() # Figura para entalpía
ax2.set_title("Enthalpy")
ax2.set_xlabel("x [m]")
ax2.set_ylabel("Enthalpy [KJ/Kg]")
```

Posteriormente corremos

```
for t in range(24): # Por 24 horas
    m.run(0.0416667) # 1 hora
    resultados = np.array(m.physics.engine.X, copy=False)
    # Extraer los valores de presión (índices impares)
    pressure = resultados[::m.nb * 2:2]
    # Extraer los valores de entalpía (índices pares)
    enthalpy = resultados[1::m.nb * 2:2]
    # Agregar curva de presión
    ax1.plot(x, pressure, label=f'Hora {t+1}', alpha=0.7)
    # Agregar curva de entalpía
    ax2.plot(x, enthalpy / 18.015, label=f'Hora {t+1}', alpha=0.7)
```

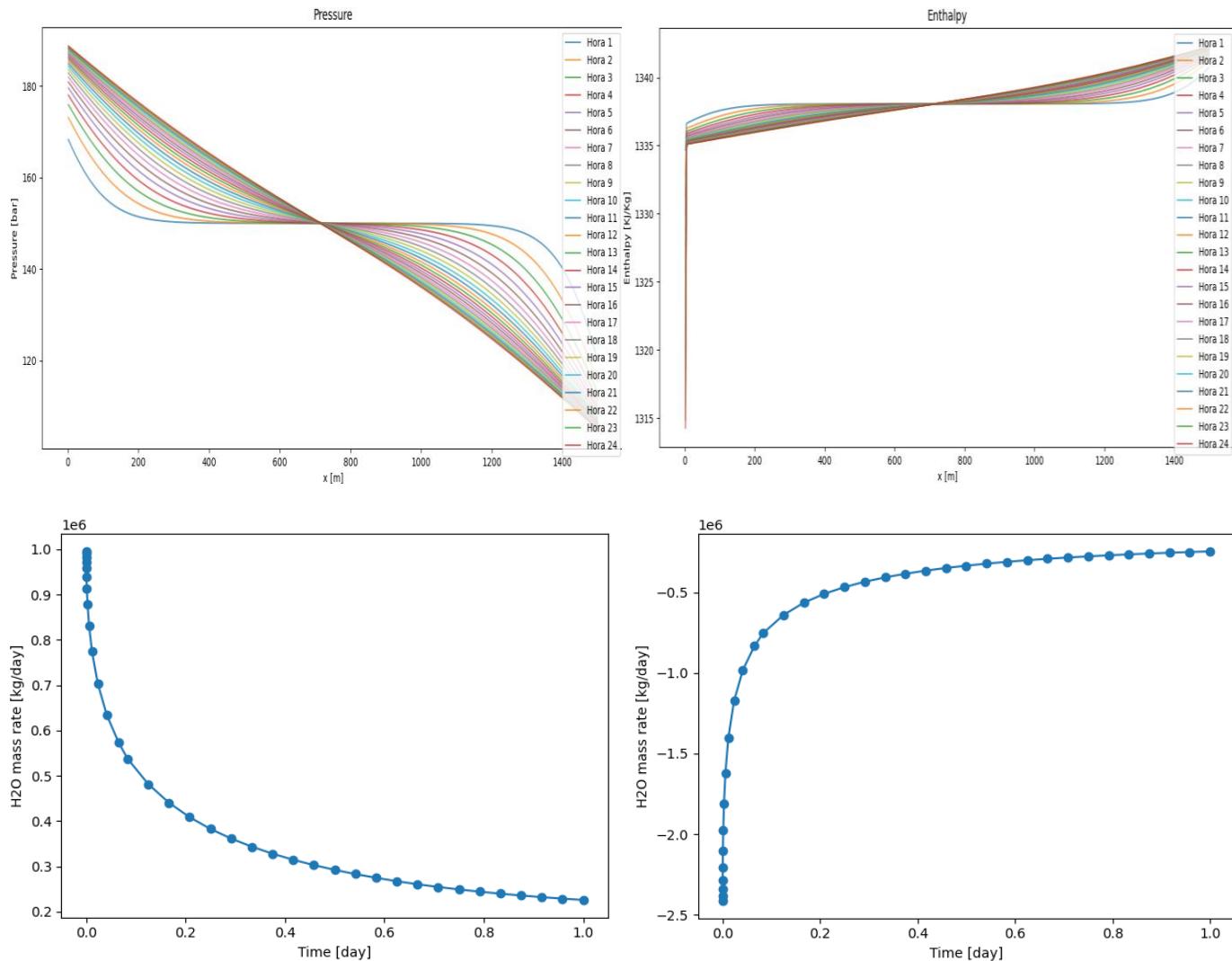
Se muestran las gráficas y se guardan:

```
# Mostrar las gráficas después del bucle
ax1.legend()
fig1.savefig('pressure.png', format='png')
plt.show()

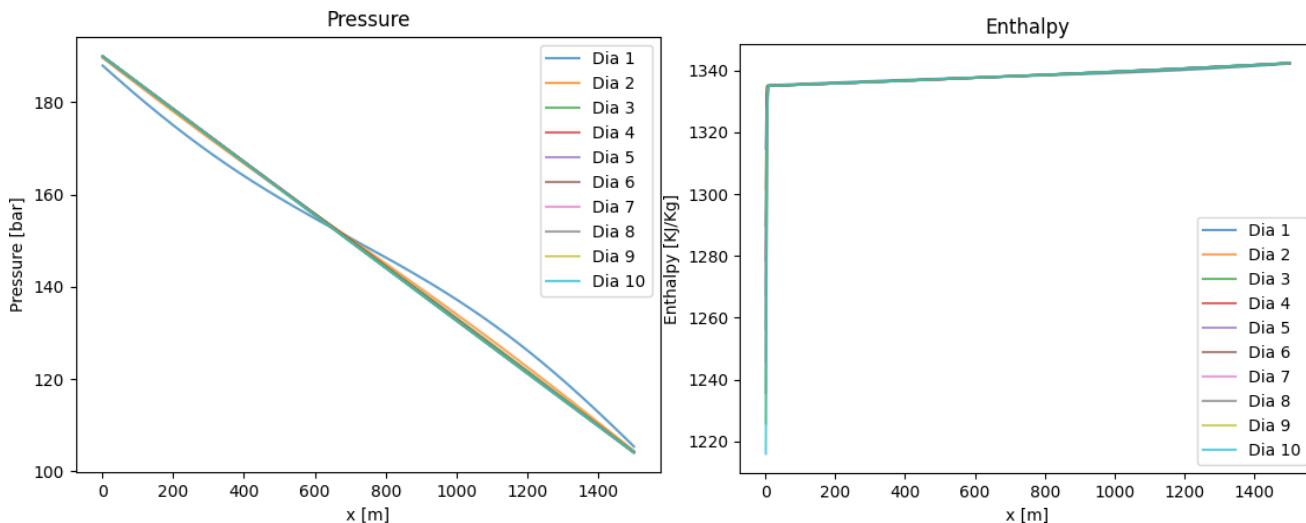
ax2.legend()
fig2.savefig('enthalpy.png', format='png')
plt.show()
```

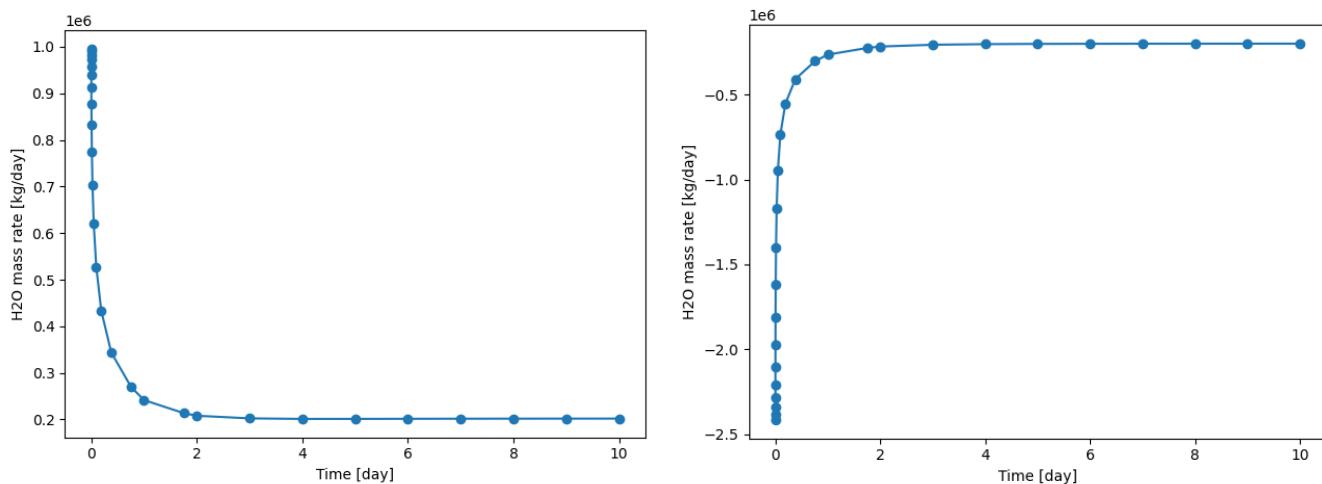
Se muestran los perfiles de presión y entalpia, además del gasto masico de agua para cada pozo.

Para 1 día:

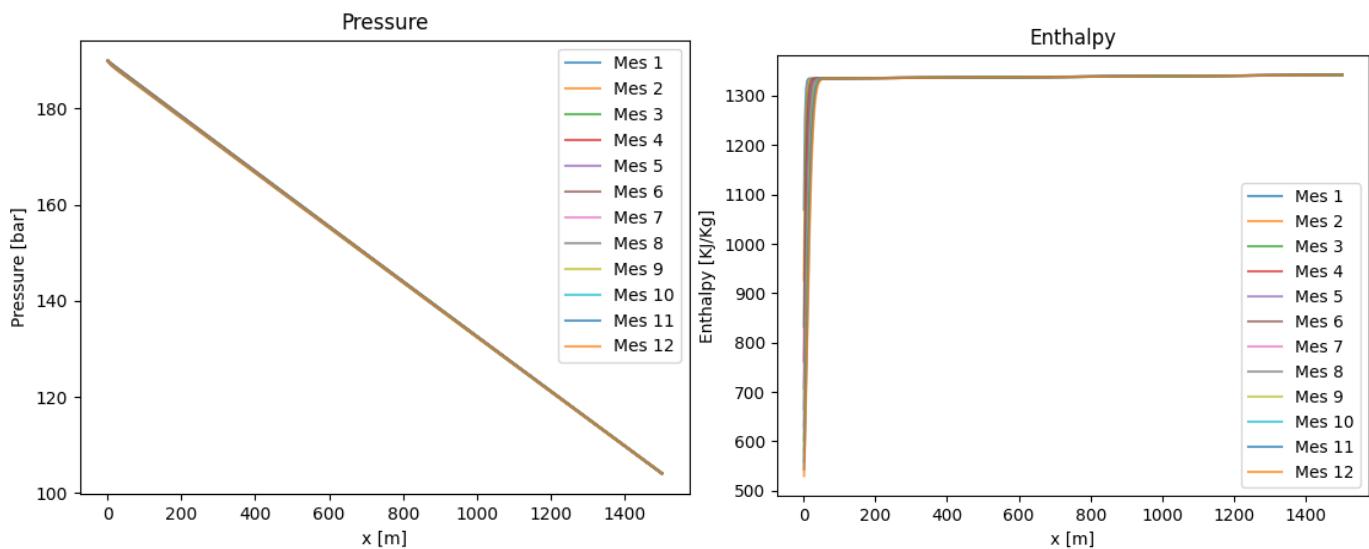


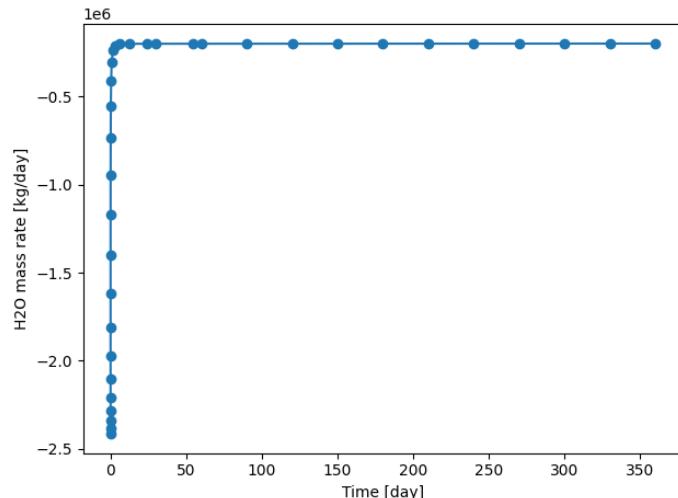
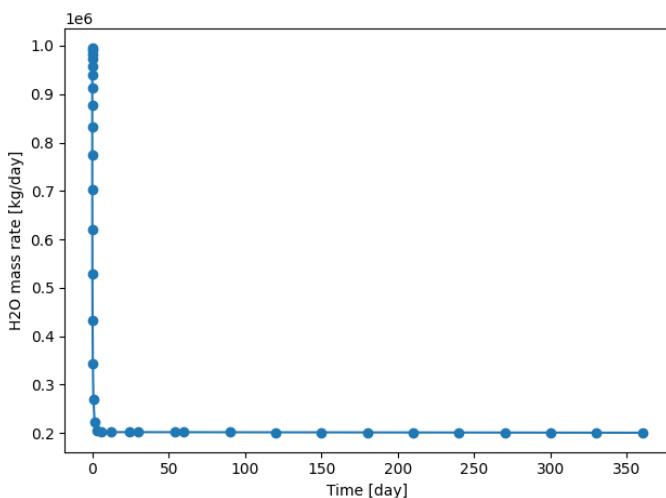
Por 10 días:



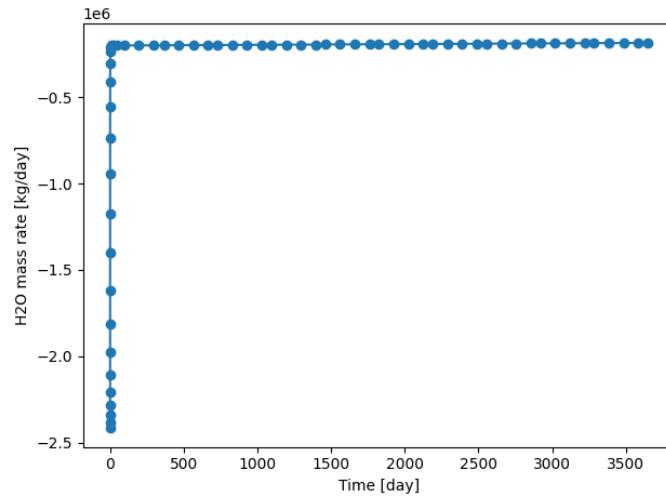
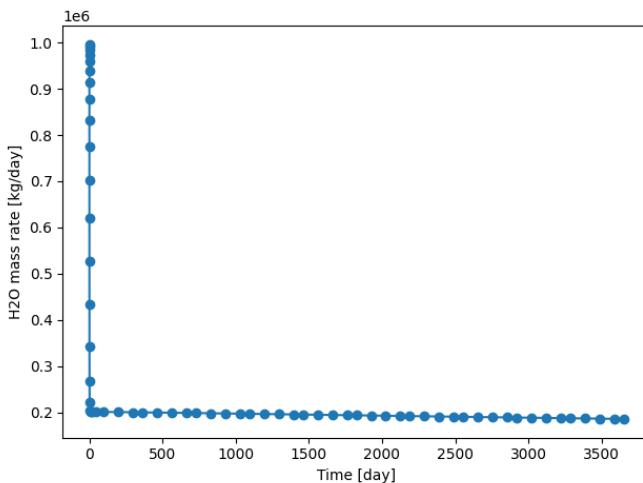
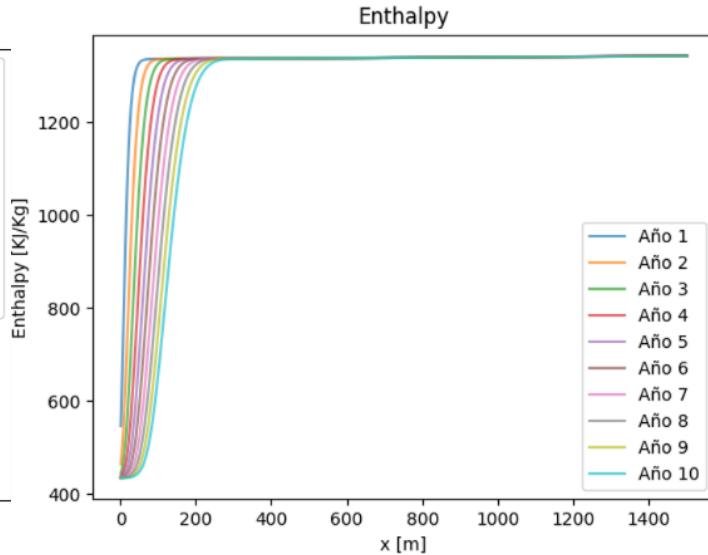
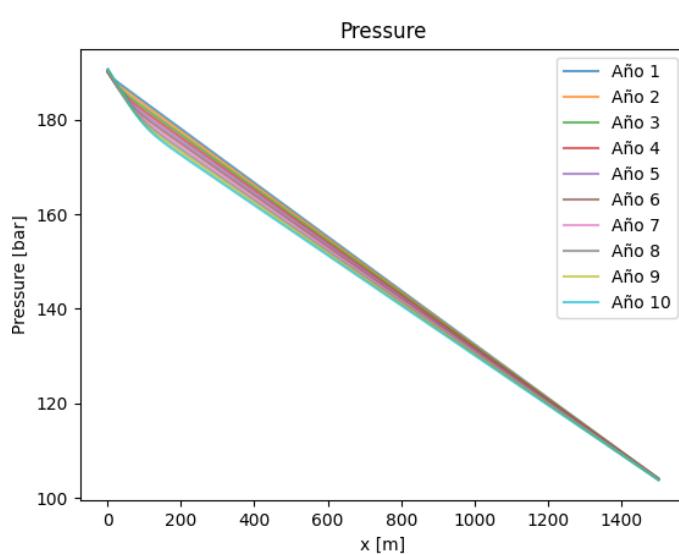


Por 1 año:

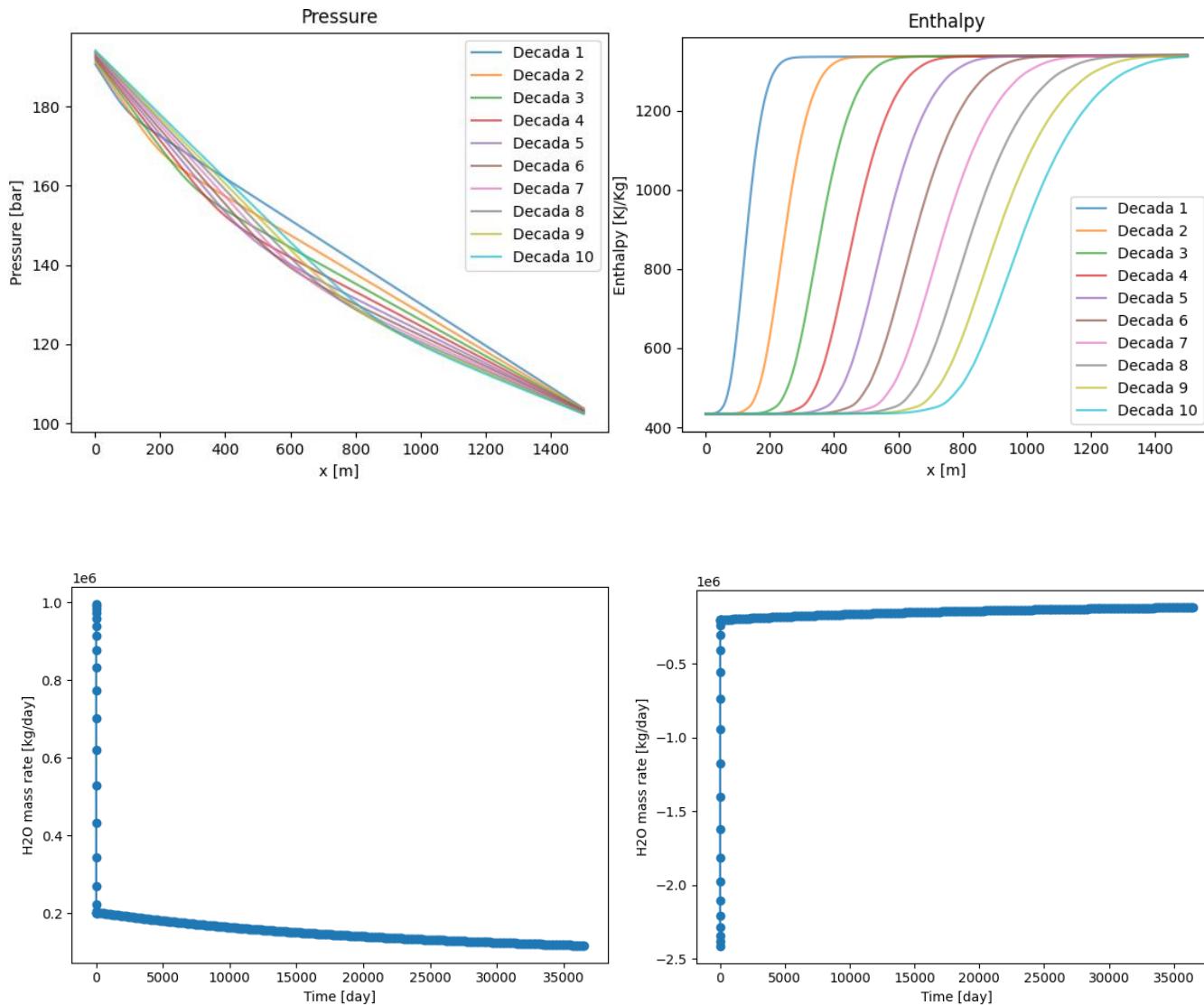




Por 10 años:

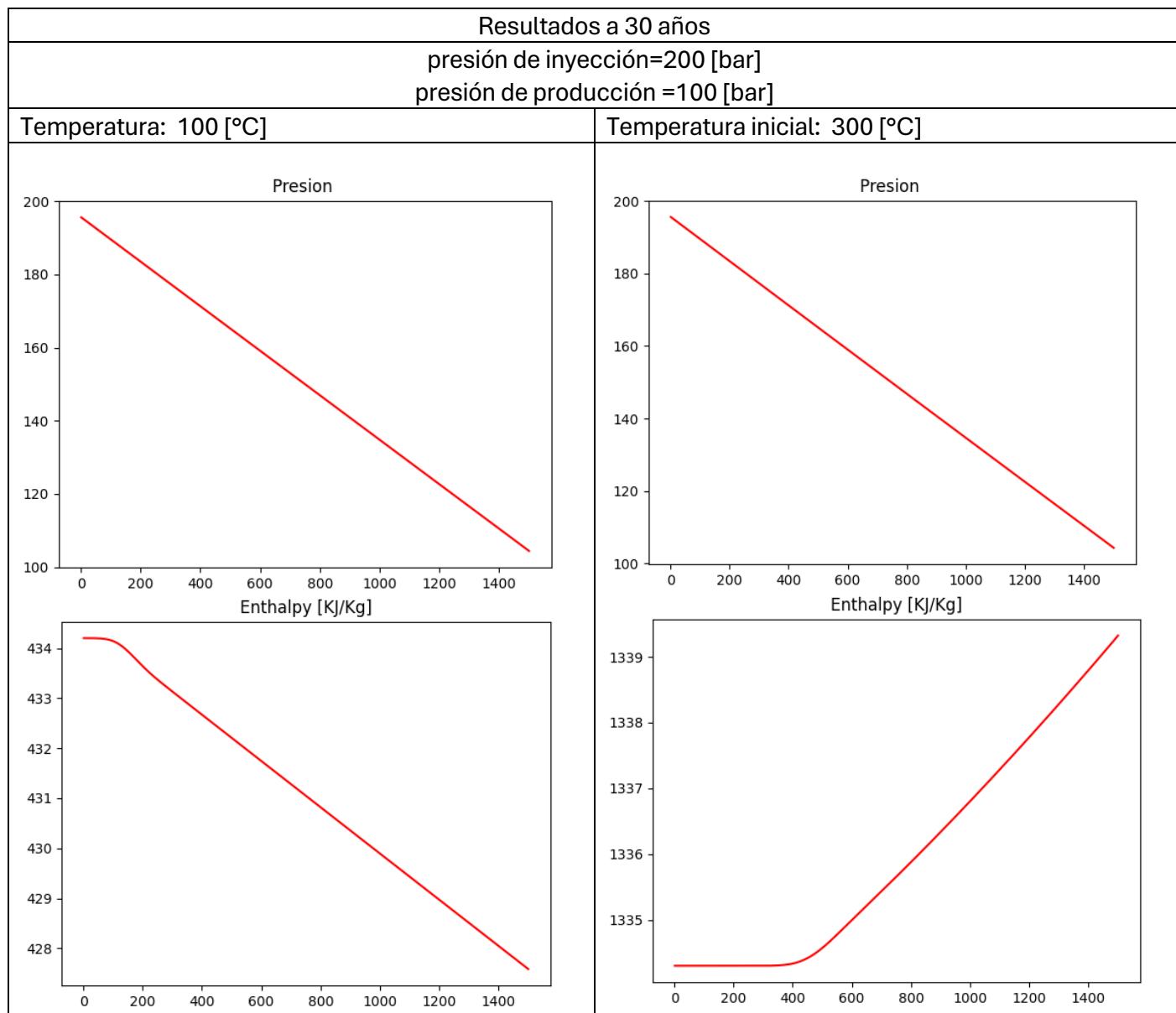
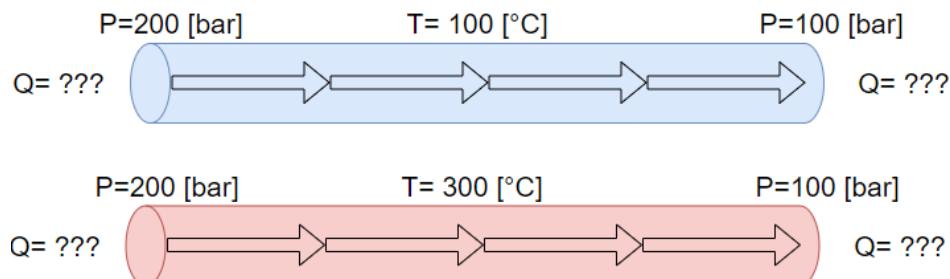


Por 100 años:



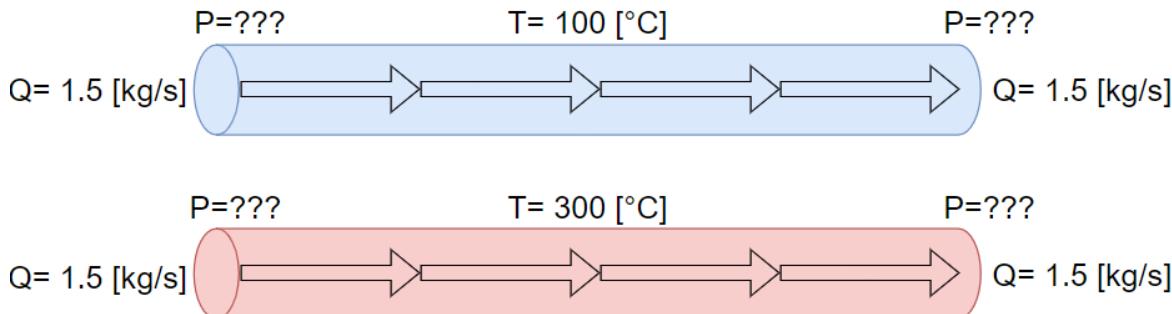
Ejemplo 3

En este ejemplo, veremos cómo afecta la temperatura al gradiente de presión y al gasto calculado. Para esto, el pozo inyector inyectara agua a la misma temperatura que el yacimiento. Se proponen 2 casos: uno de baja temperatura y otro de alta temperatura:



Agua inyectada=117,502.1733 [kg/day]	Agua inyectada=213,222.2931 [kg/day]
Agua producida=-116,958.2778 [kg/day]	Agua producida=-213,213.4094 [kg/day]

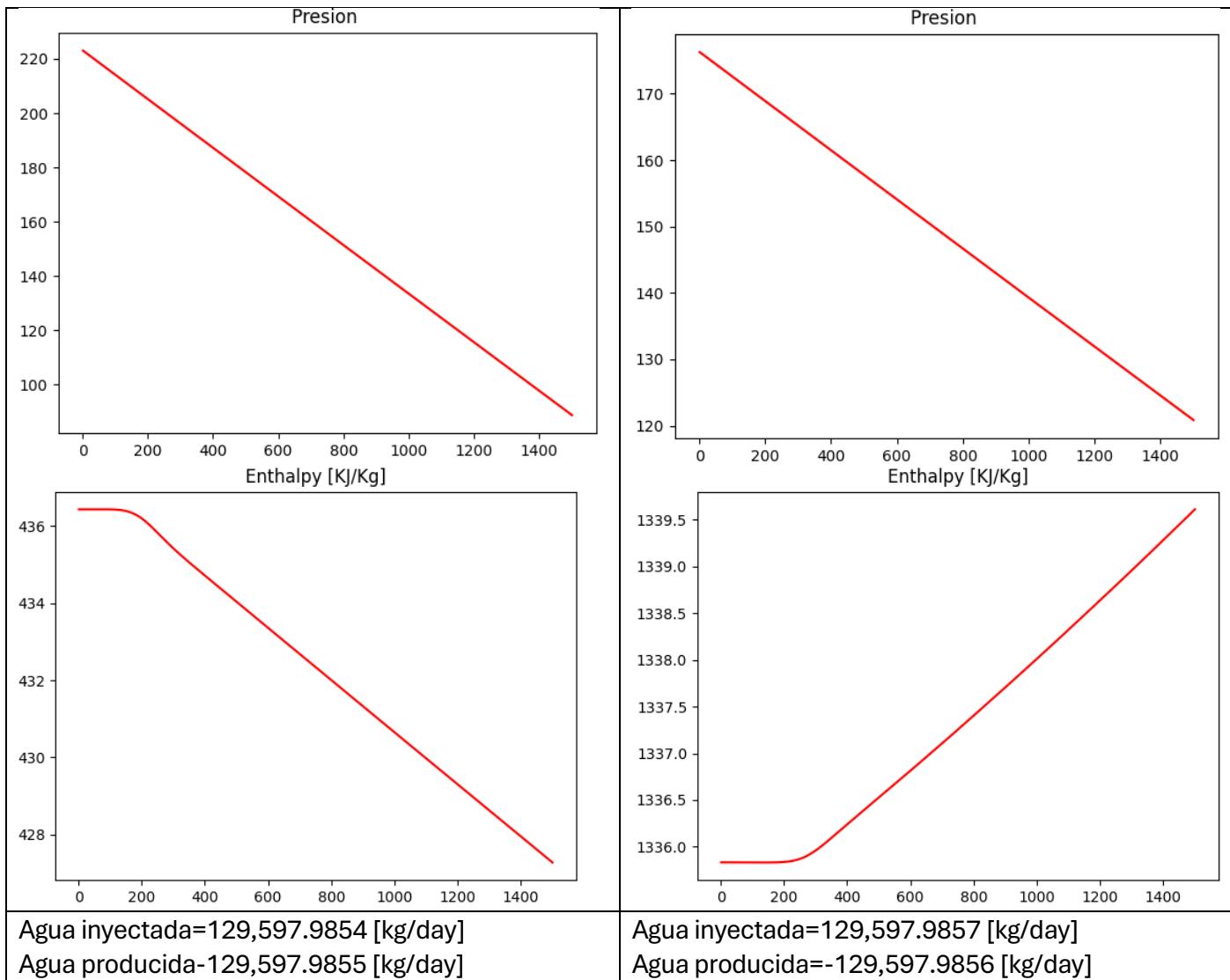
Podemos utilizar un gasto constante, y ver cómo se comporta el gradiente de presión:



```
# # Gasto constante
self.Mass_iny=1.5 # [Kg/s]
mol_rate_iny=self.Mass_iny*4795.928784 # [Kg/s] -----> [kmol/day]
self.inj_temp= 100 # [c]
```

```
for i, w in enumerate(self.reservoir.wells):
    if 'INJ' in w.name:
        self.physics.set_well_controls(wctrl=w.control,
                                        control_type=well_control_iface.MOLAR_RATE,
                                        is_inj=True,
                                        target=mol_rate_iny,
                                        phase_name='water',
                                        inj_temp=self.inj_temp + 273.15 # [c]----[k]
                                         )
```

Resultados a 30 años	
Masa de inyección =129,600.0 [kg/day]	
Temperatura: 100 [°C]	Temperatura inicial: 300 [°C]



La diferencia del gradiente y/o gasto se debe a la viscosidad del fluido. Podemos obtener la viscosidad calculada agregando dicha variable al objeto ‘property_container’:

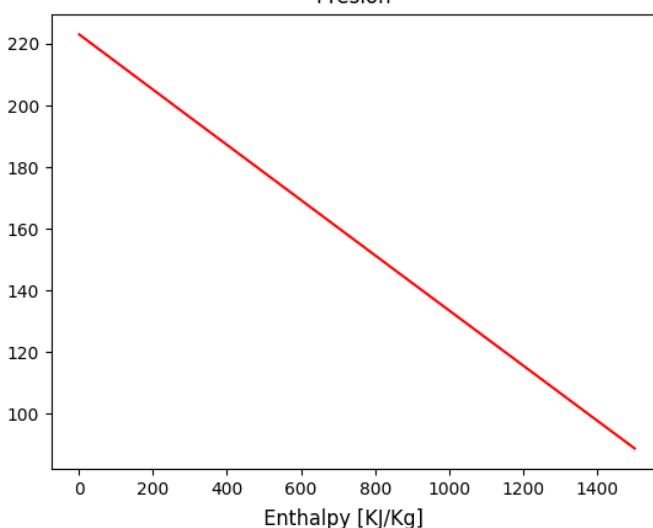
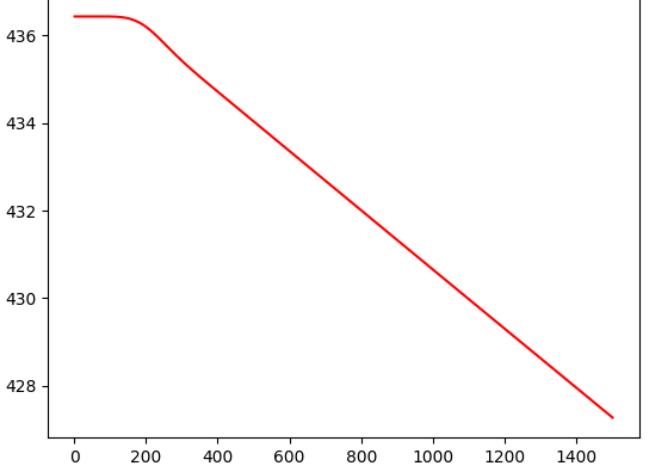
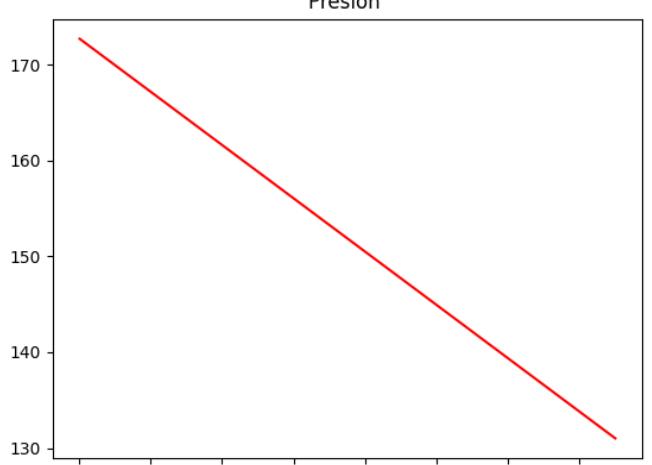
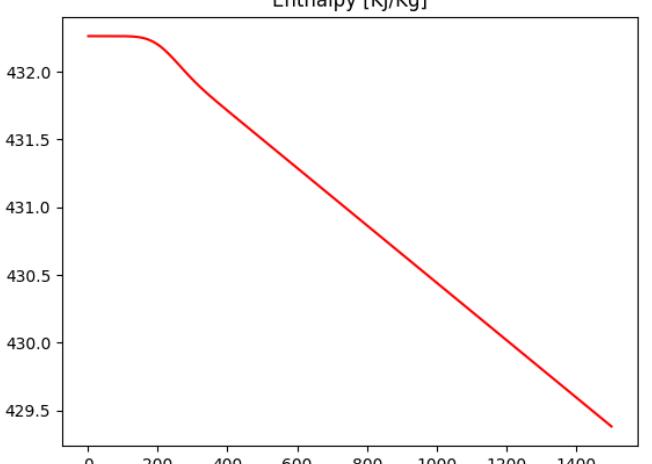
```
property_container.output_props = { 'Visco': lambda: property_container.mu[0],
                                     'temp [K]': lambda: property_container.temperature,
                                     'temp [°C]': lambda: property_container.temperature-273.15,
                                     'sat_W': lambda: property_container.saturation[0],
                                     'sat_S': lambda: property_container.saturation[1]
                                    }
```

P= 150 [bar]	0.285757 [cP]
T= 100 [°C]	
P= 150 [bar]	0.0883696 [cP]
T= 300 [°C]	

Para definir un valor constante de viscosidad, se utiliza la función ‘viscosity_ev’:

```
property_container.viscosity_ev = dict([('water', ConstFunc(0.08836)), # 0.08836 [cP]
                                         ('steam', iapws_steam_viscosity_evaluator())])
```

Podemos hacer una simulación usando la misma temperatura, pero utilizando una viscosidad diferente.

Resultados a 30 años Masa de inyección =129,600.0 [kg/day] Masa de producción =129,600.0 [kg/day]	
Temperatura: 100 [°C] Viscosidad del agua= 0.285757 [cP]	Temperatura: 100 [°C] Viscosidad del agua= 0.0883696 [cP]
 	 
Agua inyectada=129,597.9854 [kg/day] Agua producida=129,597.9855 [kg/day]	Agua inyectada=129,597.9857 [kg/day] Agua producida=-129,597.9856 [kg/day]

Ejemplo 4

Para este ejemplo, disminuiremos la presión inicial y la presión del pozo productor con respecto al ejemplo anterior, con la finalidad de realizar una vaporización flash en las cercanías del pozo productor:

```
# initialization with constant pressure and temperature
self.input_distribution = {"pressure": 100.,                      # [bar]
| | | | | "temperature": 300. + 273.15 # [c]-----[k]
| | | | }
```

El pozo productor

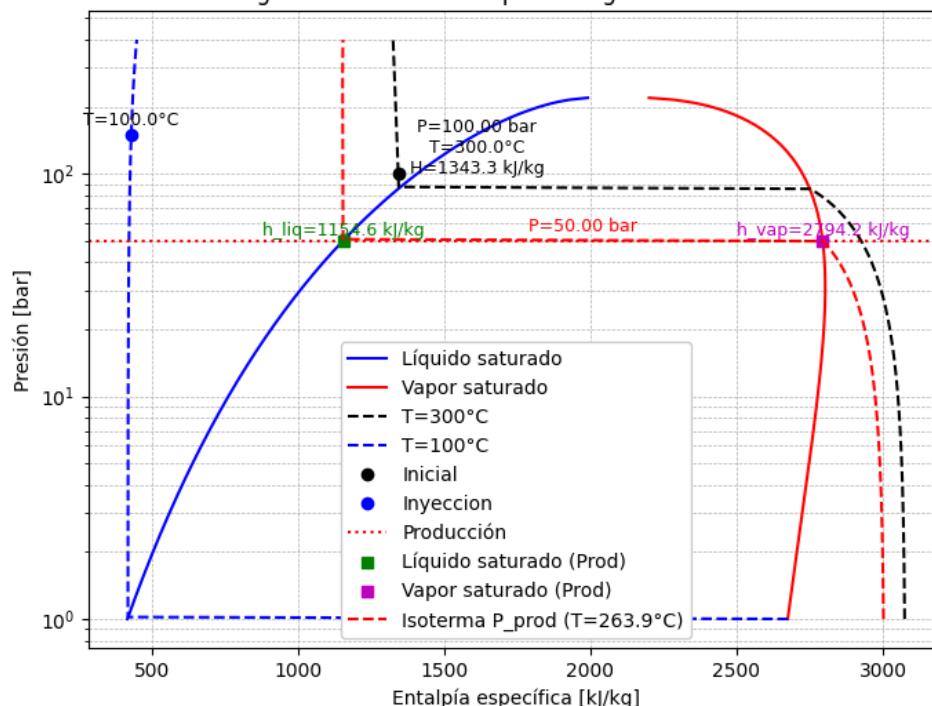
```
self.P_prod=50 # [bar]
```

Se incrementará el indicie de productividad para que las presiones en las celdas sean muy parecidas a las presiones de fondo de los pozos:

```
# add well
self.reservoir.add_well("INJ", wellbore_diameter=0.3048)
self.reservoir.add_perforation("INJ", cell_index=(1, 1, 1), skin=0.0,
| | | | | | | well_radius=0.1524, multi_segment=False,well_index=100)

# add well
self.reservoir.add_well("PRD", wellbore_diameter=0.3048)
self.reservoir.add_perforation("PRD", cell_index=(self.nb, 1, 1), skin=0.0,
| | | | | | | well_radius=0.1524, multi_segment=False,well_index=100)
```

Diagrama Presión - Entalpía del Agua con Isotermas



```
m.run(1) # End time of the simulation (days)
```

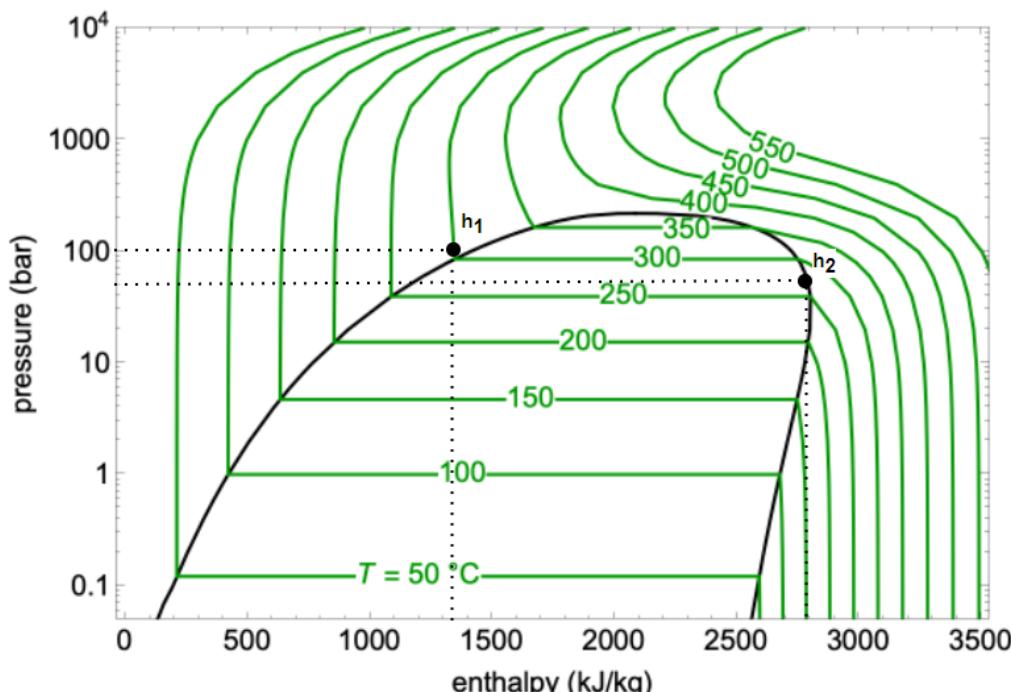
Cuando la presión disminuye sobre el agua líquida:

1. **El punto de ebullición desciende:** A menor presión, el agua hierve a menor temperatura.
 2. **Parte del líquido se evapora:** Para que ocurra la evaporación, se necesita energía (calor latente de vaporización).
 3. **La energía para la evaporación puede provenir del propio líquido**, si no hay aporte de calor externo.
Esto significa que:
 - El líquido cede energía interna para que parte de él pase a vapor.
 - Como consecuencia, **su temperatura desciende**.
- Parte de la energía interna (calor sensible) se convierte en **calor latente de vaporización**, necesario para romper las fuerzas intermoleculares y formar vapor.
 - Si el sistema no recibe calor del entorno, la energía proviene de sí mismo → **enfriamiento**.

Si no entra calor de fuera, la energía para vaporizar sale de la propia entalpia del líquido, y baja su temperatura (o se mantiene a la temperatura de saturación en la mezcla). Esa es la razón física del enfriamiento flash.

Si la presión cae a 50 bar y si solo hubiera agua (sin roca), ¿podría vaporizarse toda el agua? Para que esto suceda, la entalpia inicial debe de ser suficiente. Es decir, debe ser igual o mayor al estado final

$$h_1 > h_2$$



$$h_1 = 1343 \left[\frac{kJ}{kg} \right]$$

$$h_2 = (1 - x) * h_f(P_2) + x * h_g(P_2)$$

$$h_2 = (1 - 1) * h_f(P_2) + 1 * h_g(P_2)$$

$$h_2 = h_g(P_2) = 2794.2 \left[\frac{kJ}{kg} \right]$$

No hay suficiente energía. ¿Cuál sería la proporción de la mezcla? ¿A qué temperatura estaría el fluido?

$$h_1 = h_2$$

$$h_1 = (1 - x) * h_f(P_2) + x * h_g(P_2)$$

$$h_1 = h_f(P_2) - x * h_f(P_2) + x * h_g(P_2)$$

$$h_1 = h_f(P_2) + x (-h_f(P_2) + h_g(P_2))$$

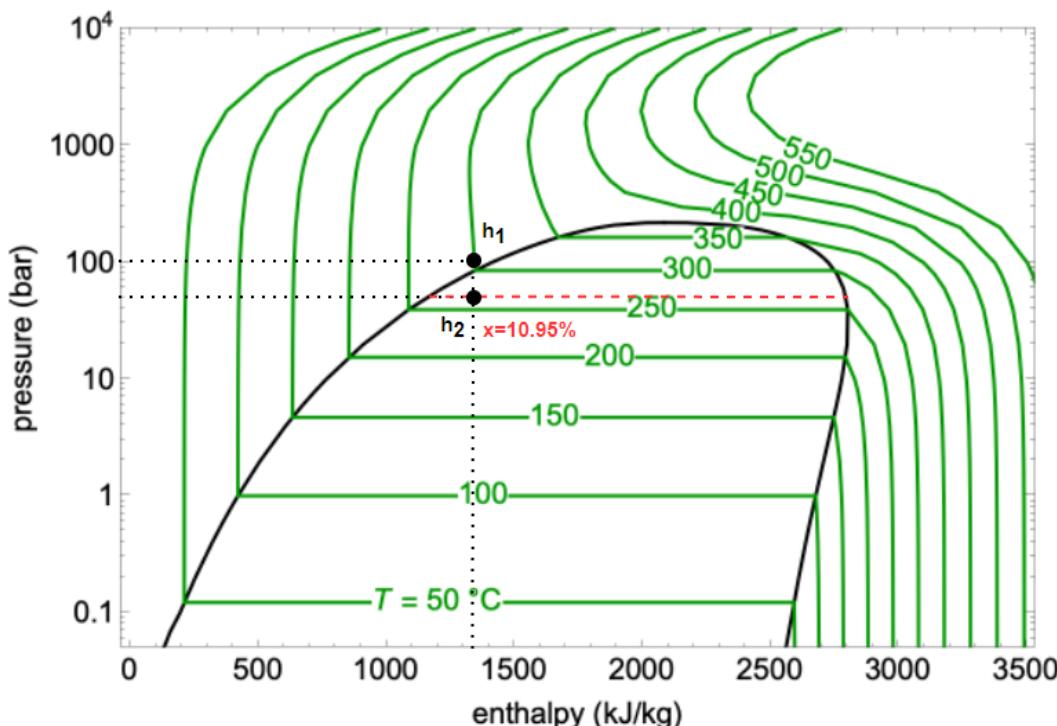
$$x (h_g(P_2) - h_f(P_2)) = h_1 - h_f(P_2)$$

$$x = \frac{h_1 - h_f(P_2)}{h_g(P_2) - h_f(P_2)}$$

$$h_f(P_2) = 1164.52 \left[\frac{kJ}{kg} \right]$$

$$h_g(P_2) = 2794.2 \left[\frac{kJ}{kg} \right]$$

$$x = \frac{1343 - 1164.52}{2794.2 - 1164.52} = \frac{178.48}{1629.68} = 0.1095 = 10.95\% \text{ de vapor}$$



No podría vaporizarse toda el agua, se tendría mezcla. La temperatura de la mezcla sería igual a la temperatura de saturación que corresponde a la presión:

$$T_s = T(P_2) = T(50) = 537.6[k] = 264.45^{\circ}C$$

¿Qué pasaría si consideramos la roca? ¿Podría vaporizarse toda el agua? Para que toda el agua se vaporice, la enérgica inicial del sistema debe de ser lo suficientemente alta. Es decir, la energía inicial debe de ser igual o mayor que la energía final del sistema.

$$E_1 > E_2$$

Se asume que el estado final es 100% vapor y, además, que la temperatura de todo el sistema es igual a la temperatura de saturación (incluyendo la temperatura de la roca):

$$\begin{aligned} E_1 &= E_{1,w} + E_{1,r} = m_w h_w + V_r C_r T_1 \\ E_2 &= E_{2,w} + E_{2,r} = m_w [(1 - x)h_f(P_2) + xh_g(P_2)] + V_r C_r T_s \\ E_2 &= m_w [(1 - 1) * h_f(P_2) + 1 * h_g(P_2)] + V_r C_r T_s \\ E_2 &= m_w * h_g(P_2) + V_r C_r T_s \end{aligned}$$

Datos:

$$T_1 = 300^{\circ}C$$

$$T_s = 264.45^{\circ}C$$

$$V_{celda} = 100 * 100 * 3 = 30000 [m^3]$$

$$V_r = V_{celda} * (1 - \emptyset) = V_{celda} * (1 - .01) = 29,700 [m^3]$$

$$V_{poroso} = V_{celda} * \emptyset = 30000 [m^3] * .01 = 300 [m^3]$$

$$V_w = V_{poroso} * S_w = 300 [m^3] * 1 = 300 [m^3]$$

$$m_w = 300 [m^3] * 715.509 \left[\frac{kg}{m^3} \right] = 214,652.7 [kg]$$

$$C_r = 2470 \left[\frac{kJ}{m^3 k} \right]$$

Sustituyendo:

$$E_1 = 214,652.7 [kg] * 1343 \left[\frac{kJ}{kg} \right] + 29,700 [m^3] * 2470 \left[\frac{kJ}{m^3 k} \right] * (573.15[k])$$

$$E_1 = 288,278,576.1 [kJ] + 4.2045 \times 10^{10} [kJ] = 4.233 \times 10^{10} [kJ]$$

$$E_2 = 214,652.7 [kg] * 2794.2 \left[\frac{kJ}{kg} \right] + 29,700 [m^3] * 2470 \left[\frac{kJ}{m^3 k} \right] * (537.6[k])$$

$$E_2 = 599,782,574.3 [kJ] + 3.943 \times 10^{10} [kJ]$$

$$E_2 = 4.003 \times 10^{10} [kJ]$$

$$E_1 > E_2$$

Hay energía de sobra en el sistema (la roca aporta la mayor parte) para vaporizar toda el agua. Como todo el líquido ya se ha vaporizar y aún hay energía disponible, significa que cualquier aumento de energía va a aumentar su temperatura. Para conocer cuanto subirá la temperatura del vapor al agregar una cierta cantidad de energía se utiliza:

$$\Delta E = m \cdot c_{p,vapor} \cdot \Delta T$$

Donde:

$$c_{p,vapor} = \text{calor específico del vapor a presión constante } [kJ/(kg \cdot K)]$$

$$c_{p,vapor} = 2 [kJ/(kg \cdot K)]$$

Es un valor aproximado para vapor de agua a alta temperatura ($\approx 300^{\circ}\text{C}$) y presiones moderadas (50 bar). Para conoce la temperatura final del sistema, se hace el balance energético:

$$E_1 = E_2$$

$$E_1 = E_{1,w} + E_{1,r} = m_w h_w + V_r C_r T_1$$

$$E_2 = E_{2,w} + E_{2,r} = m_w * h_g(P_2) + m_w * c_{p,vapor}(T_f - T_s) + V_r C_r T_2$$

Sustituyendo:

$$m_w h_w + V_r C_r T_1 = m_w * h_g(P_2) + m_w * c_{p,vapor}(T_2 - T_s) + V_r C_r T_2$$

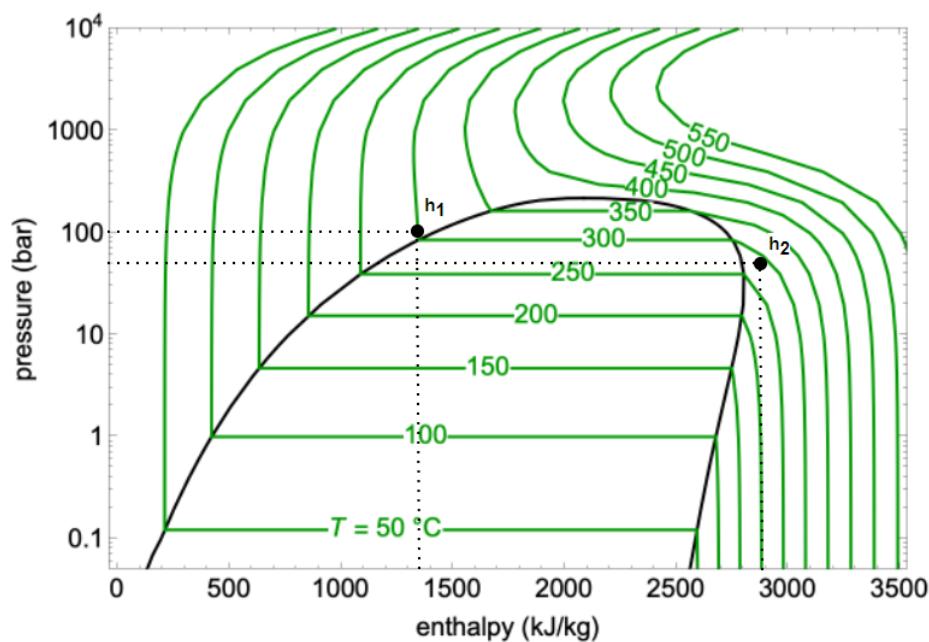
$$m_w h_w + V_r C_r T_1 - m_w * h_g(P_2) = m_w * c_{p,vapor} * T_2 - m_w * c_{p,vapor} * T_s + V_r C_r T_2$$

$$m_w h_w + V_r C_r T_1 - m_w * h_g(P_2) + m_w * c_{p,vapor} * T_s = (m_w * c_{p,vapor} + V_r C_r) T_2$$

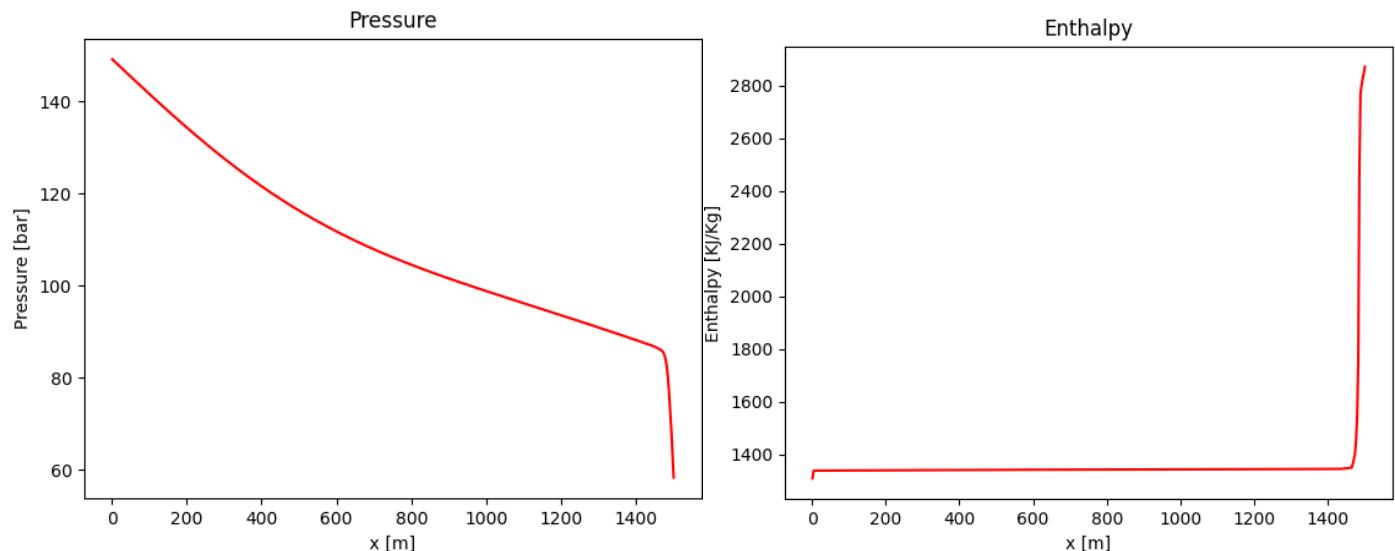
$$T_2 = \frac{m_w h_w + V_r C_r T_1 - m_w * h_g(P_2) + m_w * c_{p,vapor} * T_s}{m_w * c_{p,vapor} + V_r C_r}$$

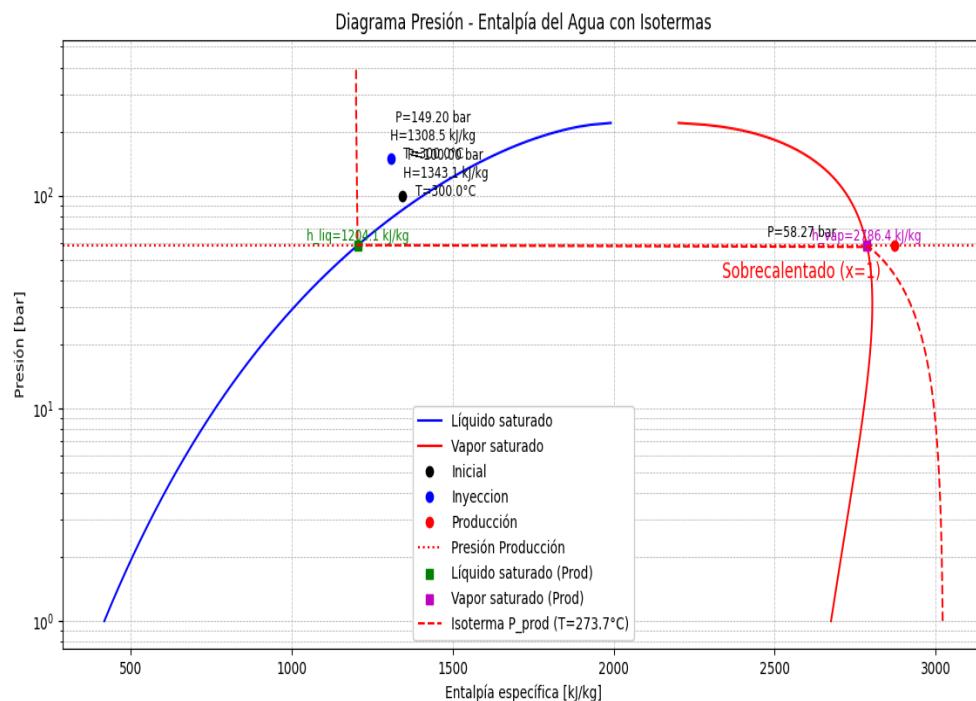
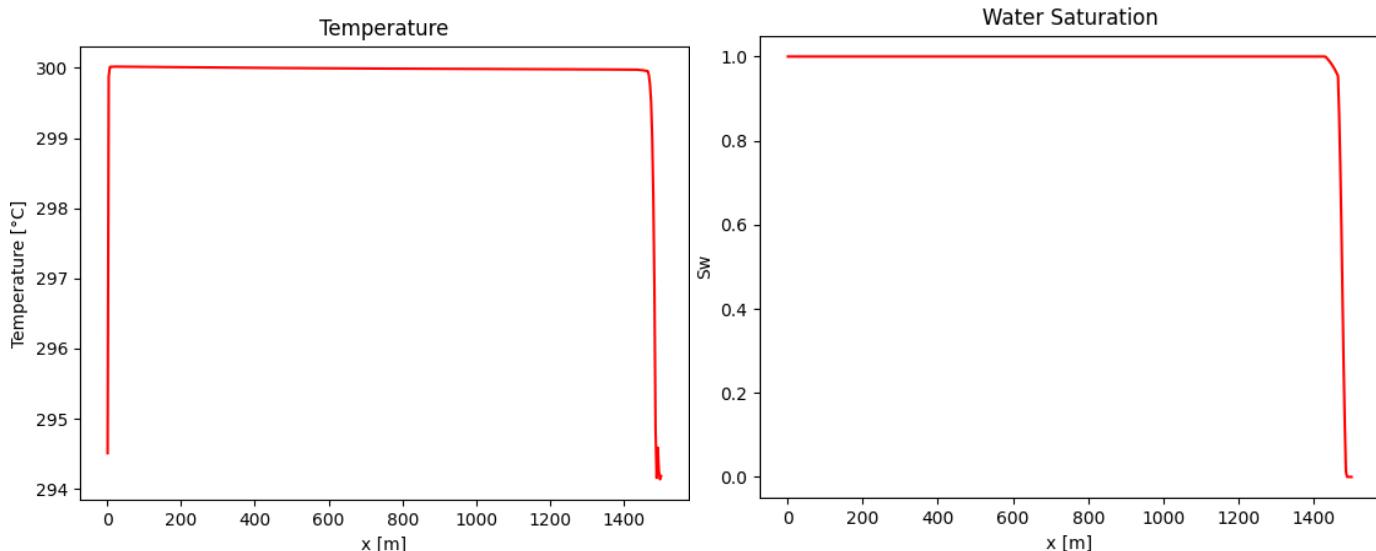
$$T_2 = \frac{4.233 \times 10^{10} [kJ] - 599,782,574.3 [kJ] + 214,652.7 [kg] * 2 \left[\frac{kJ}{kg * K} \right] * 537.6 [k]}{214,652.7 [kg] * 2 \left[\frac{kJ}{kg * K} \right] + 29,700 [m^3] * 2470 \left[\frac{kJ}{m^3 k} \right]}$$

$$T_2 = \frac{4.1965 \times 10^{10} [kJ]}{73,788,305.4 \left[\frac{kJ}{k} \right]} = 568.7215 [k] = 295.5715 [{}^{\circ}\text{C}]$$



Comparamos el cálculo analítico con los resultados para 1 día de simulación:





Se recomienda correr una simulación usando un valor de $h_{cap}=200$ [kJ/m³k] y comparar los resultados de la simulación con los teóricos. Si usáramos este valor, se tendría que:

$$C_r = 200 \left[\frac{kJ}{m^3k} \right]$$

Sustituyendo:

$$E_1 = 214,652.7 [kg] * 1343 \left[\frac{kJ}{kg} \right] + 29,700 [m^3] * 200 \left[\frac{kJ}{m^3k} \right] * (573.15[k])$$

$$E_1 = 288,278,576.1 [kJ] + .3404x10^{10}[kJ] = .3692x10^{10}[kJ]$$

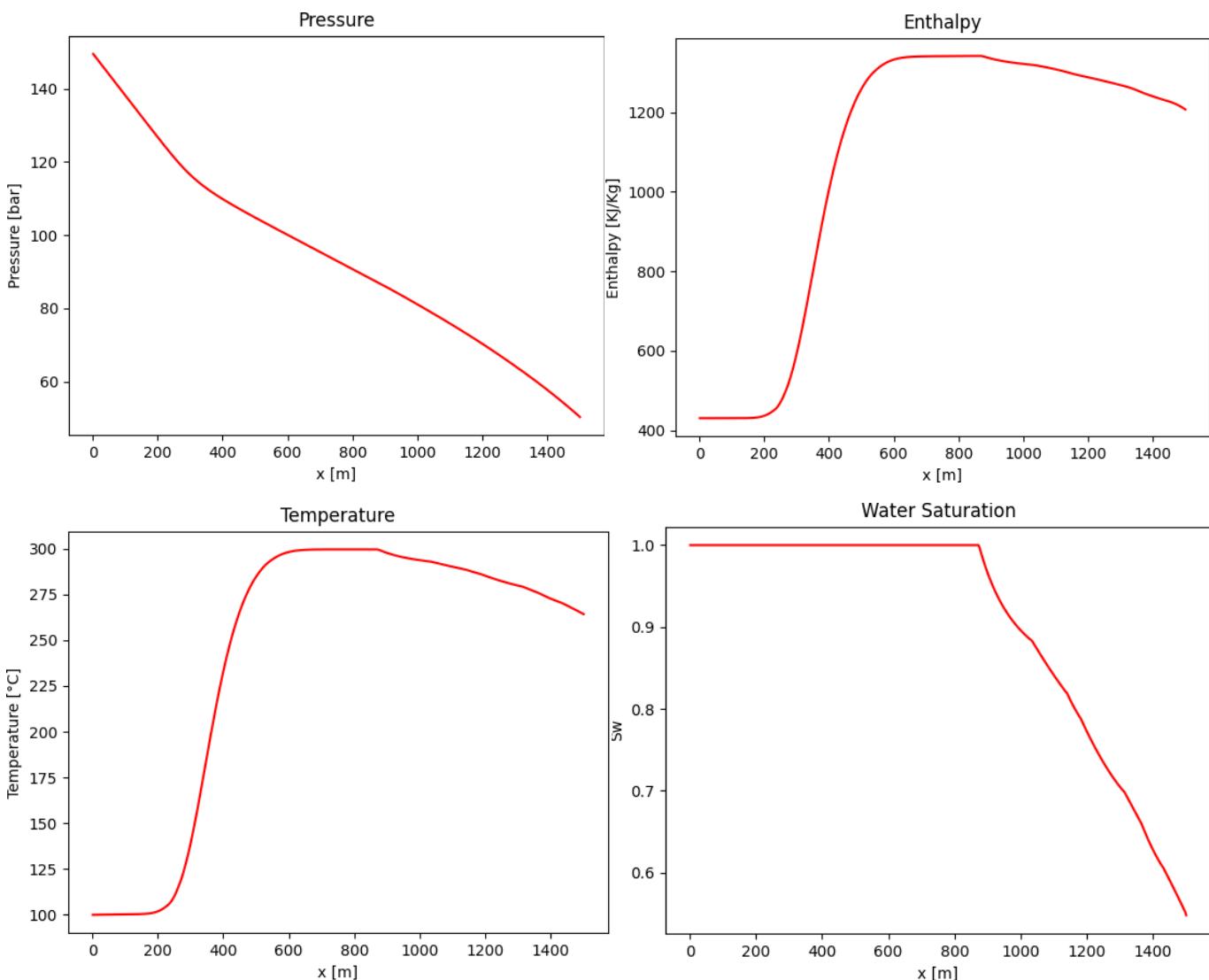
$$E_2 = 214,652.7 [kg] * 2794.2 \left[\frac{kJ}{kg} \right] + 29,700 [m^3] * 200 \left[\frac{kJ}{m^3 k} \right] * (537.6[k])$$

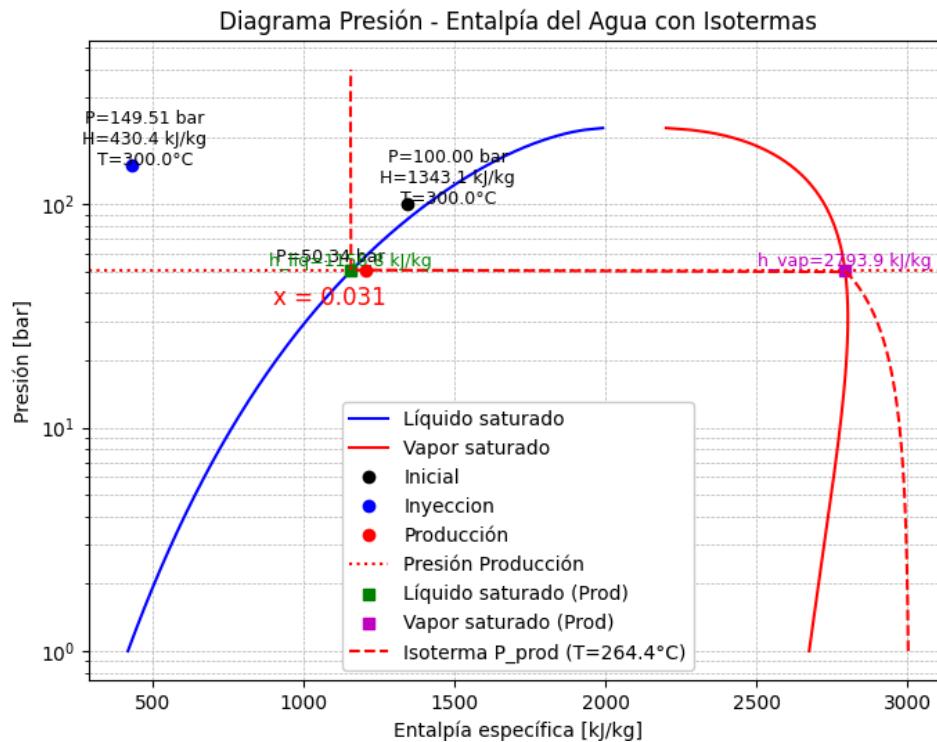
$$E_2 = 599,782,574.3[kJ] + .3193x10^{10}[kJ]$$

$$E_2 = .3793x10^{10}[kJ]$$

$$E_1 < E_2$$

Resultados para 30 años:

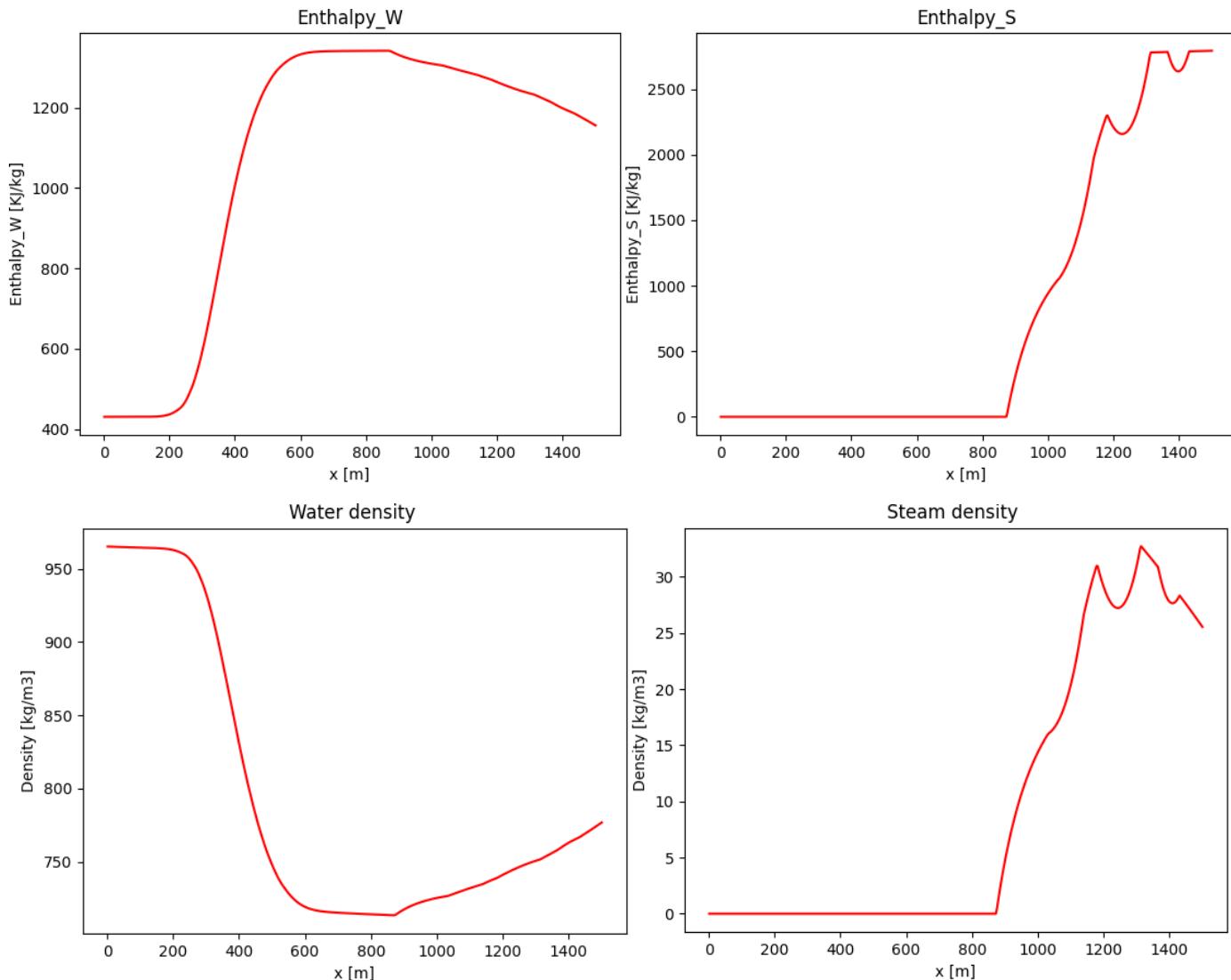




Se tiene que, para las condiciones de presión y entalpia obtenidas, la fracción de vapor en las cercanías del pozo productor calculada mediante la librería **Coolprop** es de $x=0.031$

También podemos obtener los valores de entalpia y la densidad, tanto para el agua y como para el vapor

```
property_container.output_props = {'temp [K]': lambda: property_container.temperature,
                                    #'temp [°C]': lambda: property_container.temperature-273.15,
                                    #'sat_W': lambda: property_container.saturation[0],
                                    #'sat_S': lambda: property_container.saturation[1],
                                    'Enthalpy_W [KJ/kmol]': lambda: property_container.enthalpy[0],
                                    'Enthalpy_S [KJ/kmol]': lambda: property_container.enthalpy[1],
                                    'Enthalpy_W [KJ/kg]': lambda: property_container.enthalpy[0]/18.01528,
                                    'Enthalpy_S [KJ/kg]': lambda: property_container.enthalpy[1]/18.01528,
                                    'Dens_W': lambda: property_container.dens[0],
                                    'Dens_S': lambda: property_container.dens[1]
                                   }
```



De las gráficas, se tiene que los datos obtenidos en las cercanías del pozo productor:

$$\text{Presion} = P = 50.33 \left[\frac{\text{kJ}}{\text{kg}} \right]$$

$$\text{Entalpia de la mezcla} = h_m = 1207.2 \left[\frac{\text{kJ}}{\text{kg}} \right]$$

$$\text{Temperatura} = T = 264.23 [\text{°C}]$$

$$\text{Saturacion de agua} = S_w = .548$$

$$\text{Densidad de agua} = \rho_w = 776.79 \left[\frac{\text{kg}}{\text{m}^3} \right]$$

$$\text{Densidad del vapor} = \rho_s = 25.54 \left[\frac{\text{kg}}{\text{m}^3} \right]$$

$$\text{Entalpia del agua} = h_w = 1156.07 \left[\frac{\text{kJ}}{\text{kg}} \right]$$

$$\text{Entalpia del vapor} = h_s = 2793.69 \left[\frac{\text{kJ}}{\text{kg}} \right]$$

Comprobación de la fracción másica:

$$h_m = [(1 - x) * h_f(P_2) + x * h_g(P_2)]$$

$$x = \frac{h_m - h_f(P_2)}{h_g(P_2) - h_f(P_2)}$$

$$x = \frac{1207.2 - 1156.07}{2793.69 - 1156.07}$$

$$x = 0.031$$

Comprobación0

$$s_w = \frac{\rho_s(h_s - h)}{h(\rho_w - \rho_s) - (h_w\rho_w - h_s\rho_s)}$$

$$s_w = \frac{25.54(2793.69 - 1207.2)}{1207.2 * (776.79 - 25.54) - (1156.07 * 776.79 - 2793.69 * 25.54)}$$

$$s_w = \frac{40,518.9546}{906,909 - 826,672.77} = \frac{40,518.9546}{80,236.23} = .505$$

Comprobación de la fracción másica usando las densidades calculadas:

$$V_{celda} = 100 * 100 * 3 = 30000 [\text{m}^3]$$

$$V_{poroso} = V_{celda} * \emptyset = 30000 [\text{m}^3] * .01 = 300 [\text{m}^3]$$

$$V_w = V_{poroso} * S_w = 300 [\text{m}^3] * .548 = 164.4 [\text{m}^3]$$

$$V_s = V_{poroso} * S_s = 300 [\text{m}^3] * .452 = 135.6 [\text{m}^3]$$

$$m_w = 164.4 [\text{m}^3] * 776.79 \left[\frac{\text{kg}}{\text{m}^3} \right] = 127,704.27 [\text{kg}]$$

$$m_s = 135.6 [\text{m}^3] * 25.54 \left[\frac{\text{kg}}{\text{m}^3} \right] = 3,463.22 [\text{kg}]$$

$$m_T = m_w + m_s = 127,704.27 [\text{kg}] + 3,463.22 [\text{kg}] = 131,167.49 [\text{kg}]$$

$$m_w = 97.36 \% = .9736$$

$$m_s = 2.64 \% = .0264$$

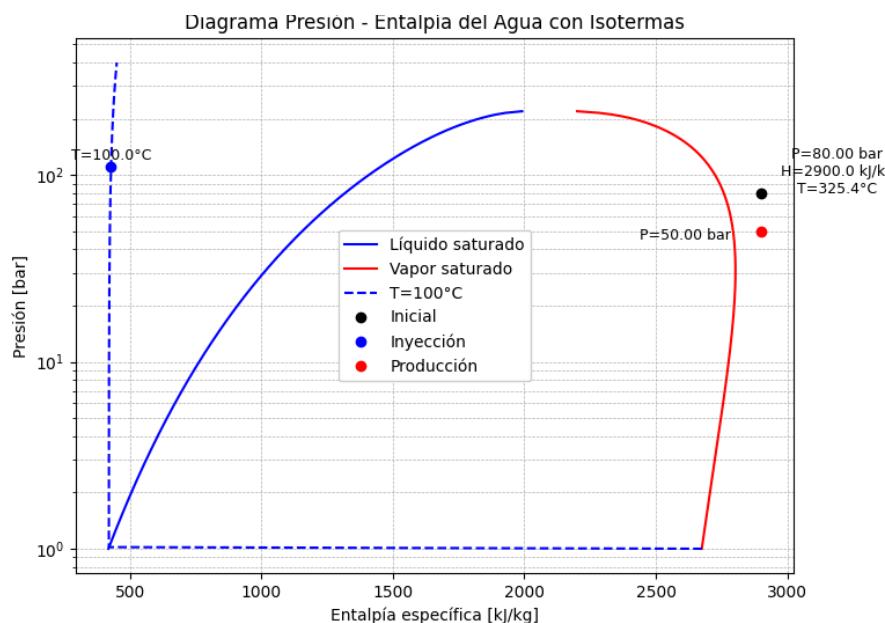
Ejemplo 5

Para este ejemplo, utilizaremos presión y entalpia para establecer una saturación de vapor del 100% en el yacimiento como condición inicial:

```
def set_initial_conditions(self):
    # initialization with constant pressure and temperature
    self.input_distribution = {"pressure": 80.,                      # [bar]
                               "enthalpy": 2900 * 18.01528   # [kJ/kg]----[kJ/kmol]
```

Y una presión de fondo de:

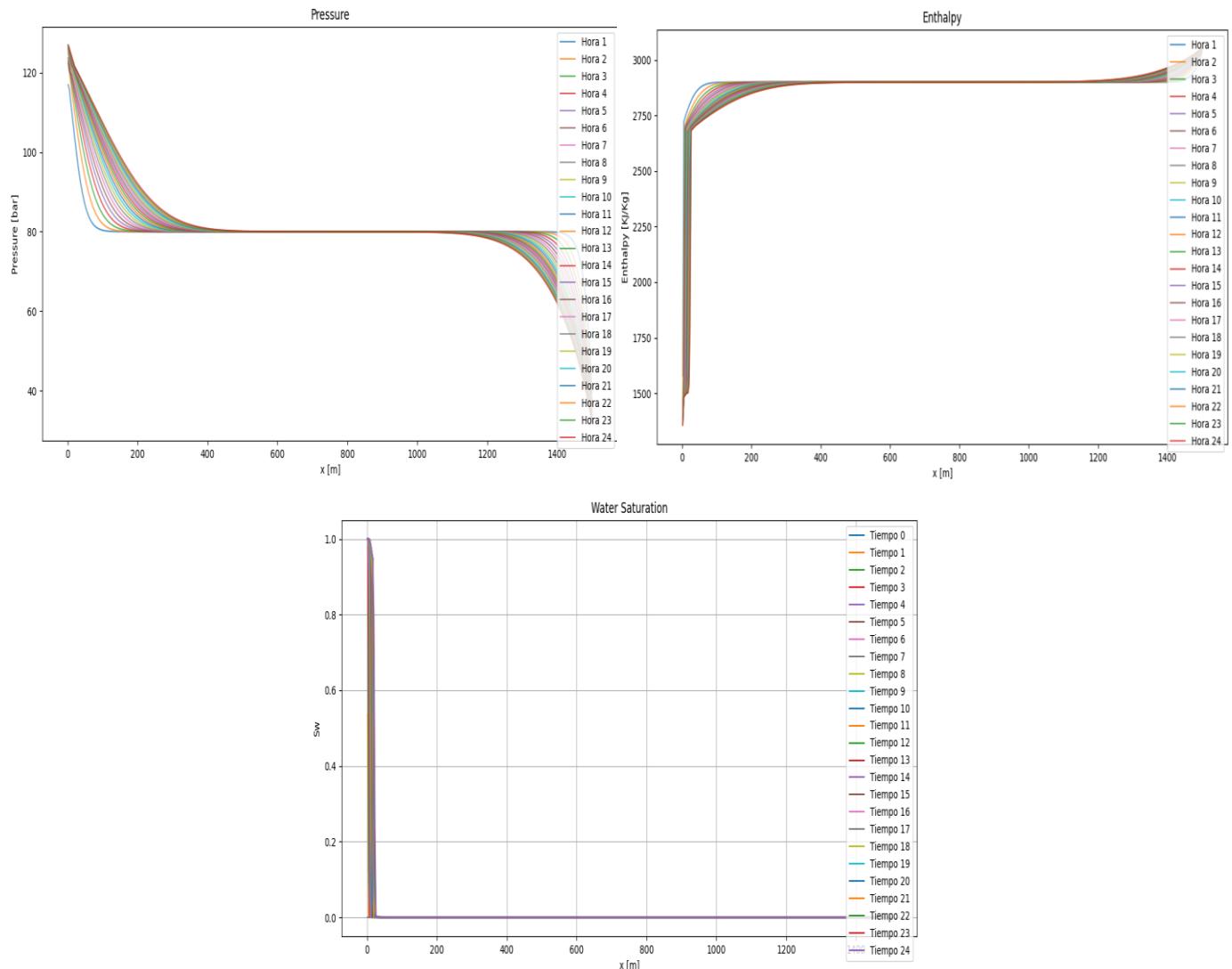
```
self.P_iny=130
self.P_prod=30
```



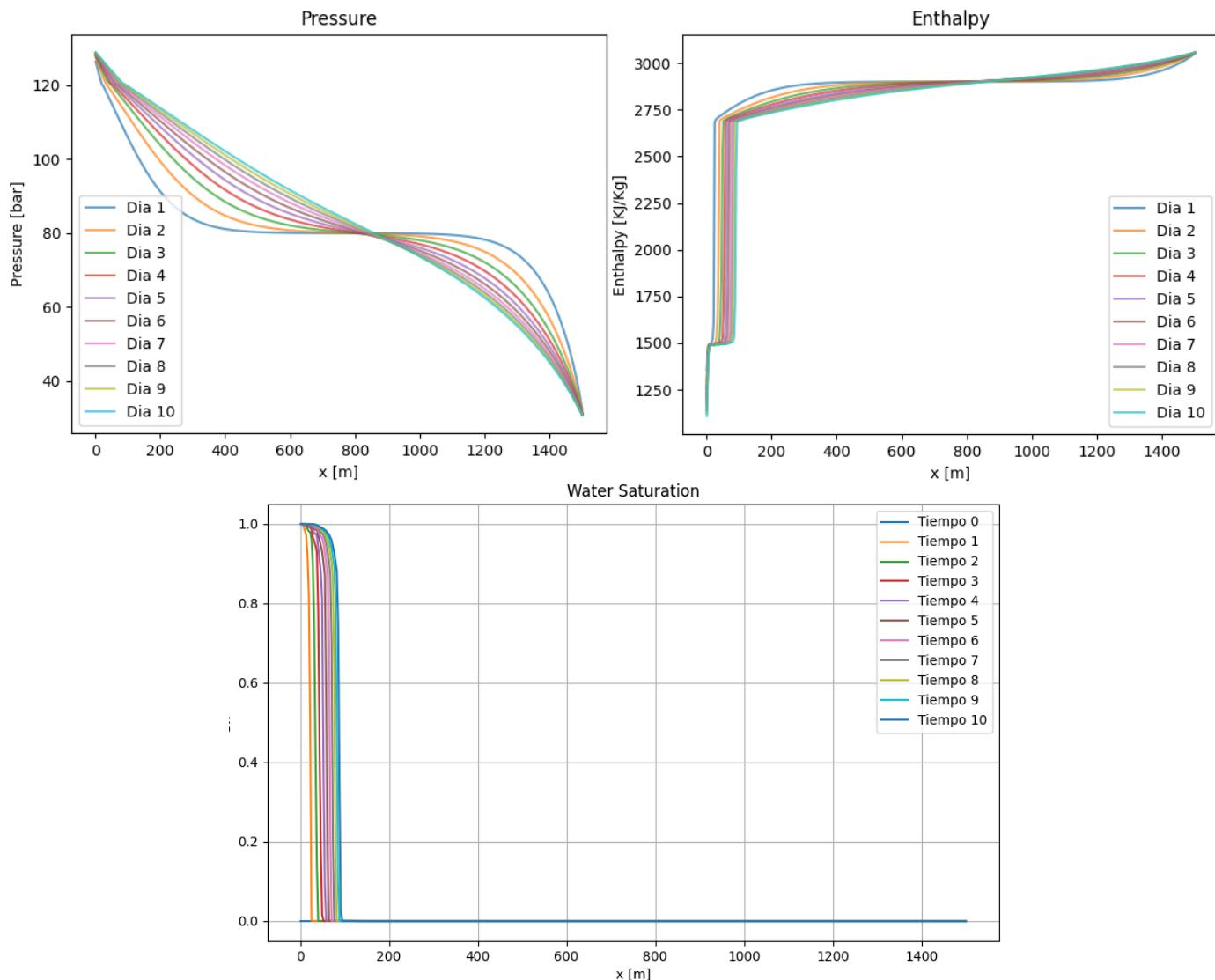
Se muestran los perfiles de presión y entalpia. Para graficar las propiedades secundarias para todos los tiempos:

```
# Crear el gráfico para cada tiempo
plt.figure(figsize=(10, 6))
for t in range(len(tiempo_final)-1): # Iterar sobre todos los tiempos
    # (menos el 1º, que corresponde al estado inicial)
    plt.plot(x, estado_final['temp [°C]'][t+1], label=f'Tiempo {t+1}')
# Añadir título, etiquetas y leyenda
plt.title("Temperature")
plt.xlabel("x [m]")
plt.ylabel("Temperature [°C]")
plt.legend(loc='best')
plt.grid(True)
plt.savefig('Temperature.png', format='png') # Guarda la figura en formato PNG
```

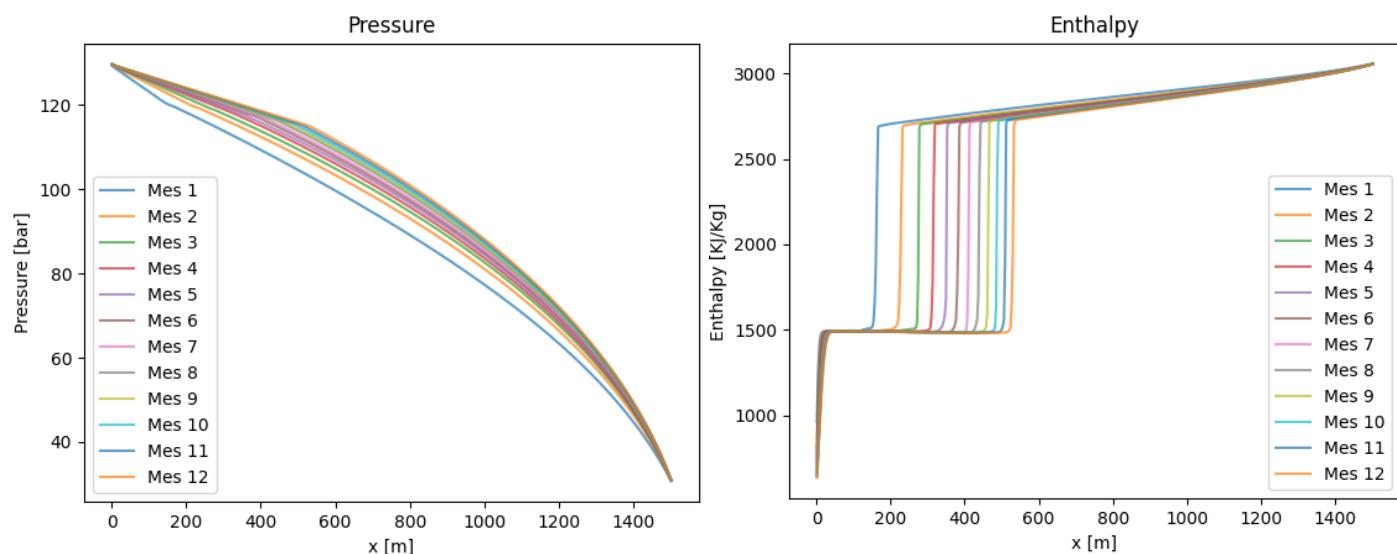
Para 1 día:

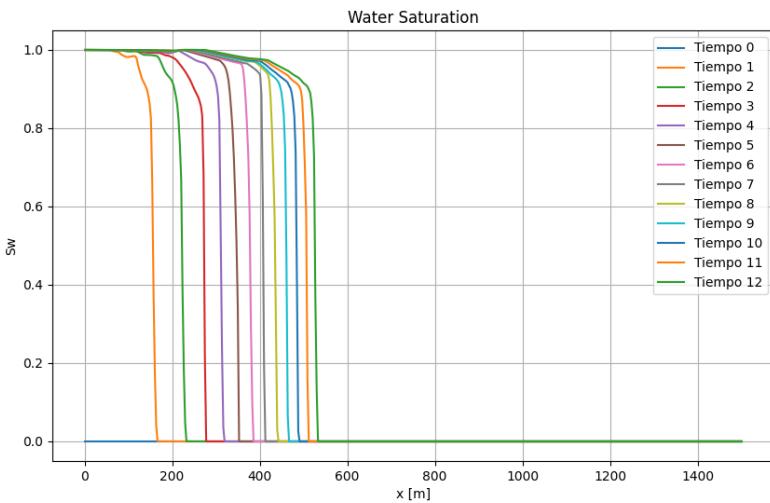


Para 10 días:

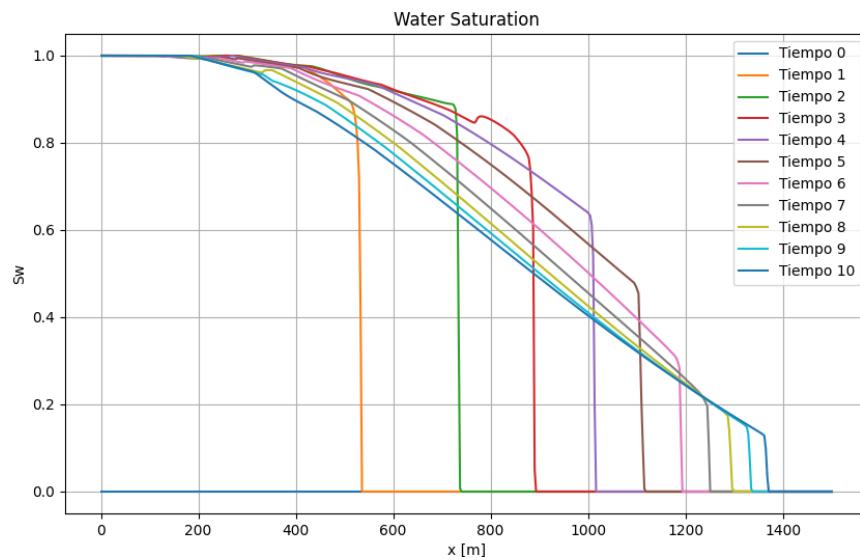
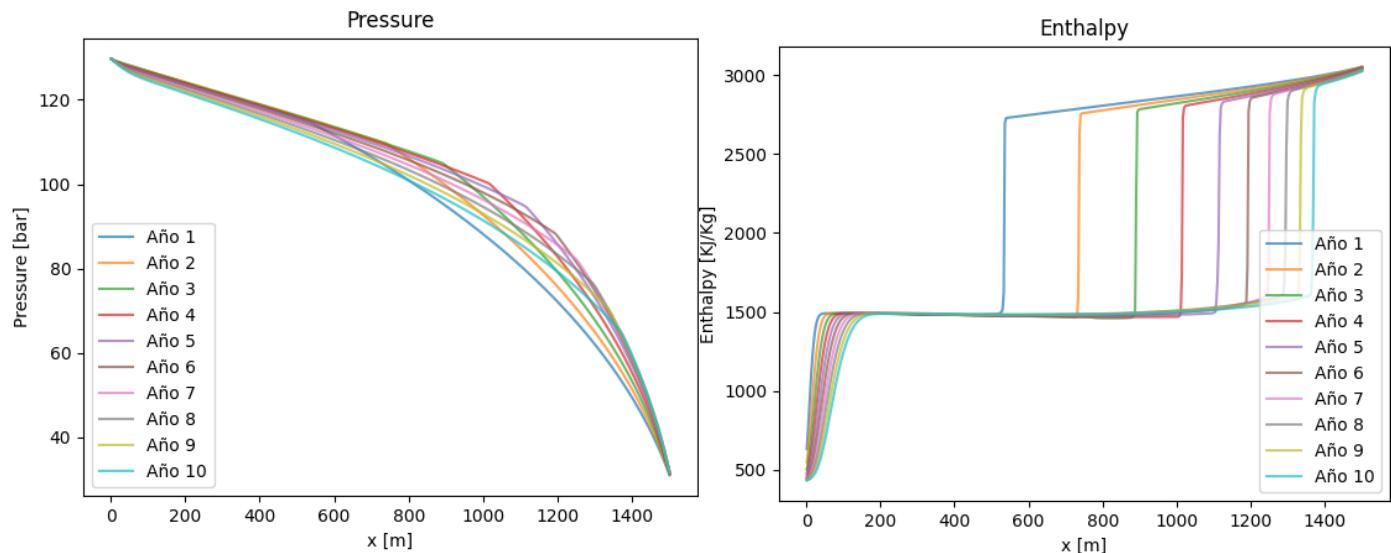


Para 1 año:

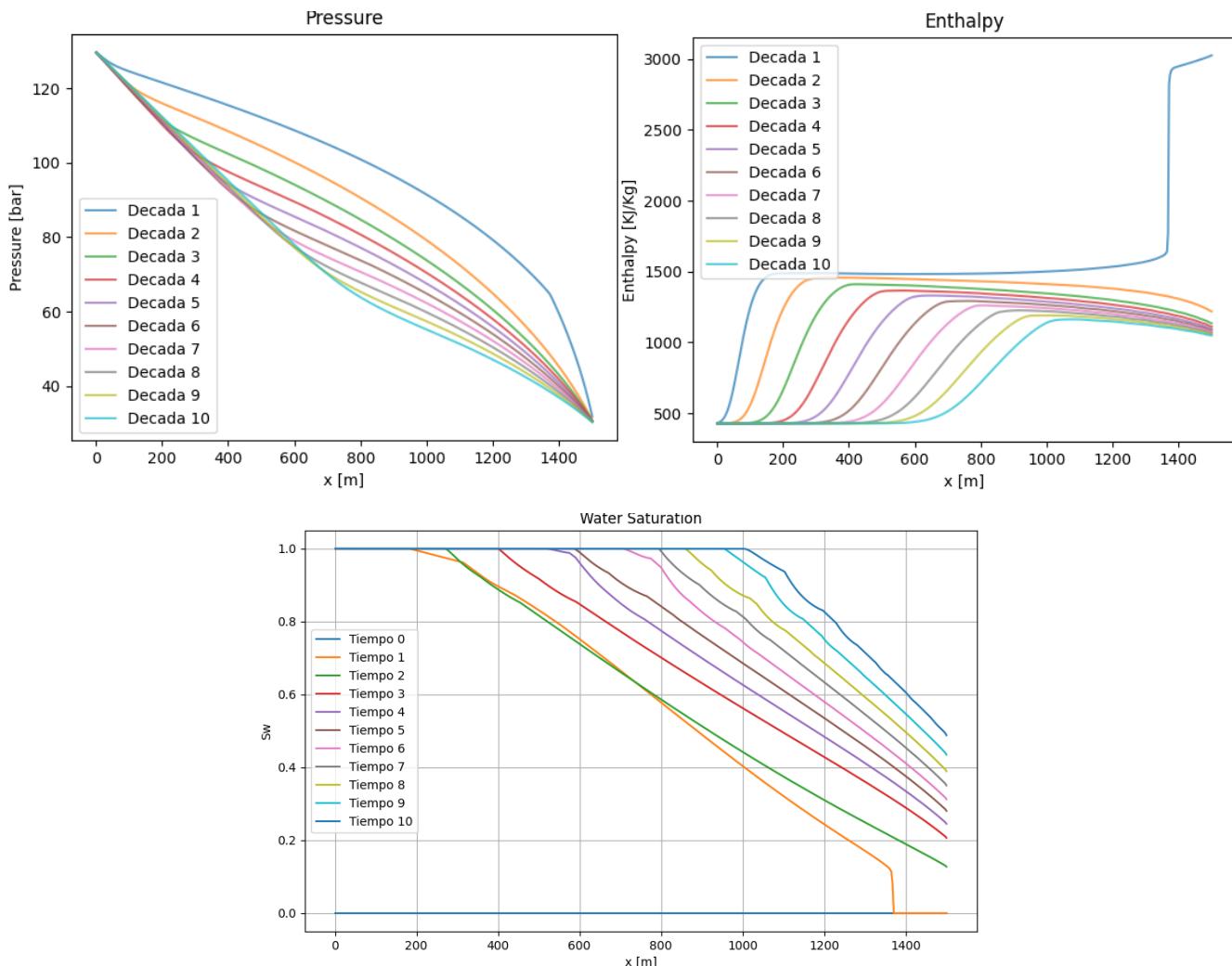




Para 10 años



Para 100 años:



Ejemplo 6

En este ejemplo se muestra cómo podemos modificar los valores de permeabilidad relativa y su efecto en la simulación. Se utilizan dos valores de “n” en el modelo de permeabilidad relativa ($n=1$ y $n=3$)

```
class iapws_water_relperm_evaluator(property_evaluator_iface):
    def __init__(self):
        super().__init__()

    def evaluate(self, state):
        water_saturation = iapws_water_saturation_evaluator()
        water_rp = water_saturation.evaluate(state) ** 1
        return water_rp

class iapws_steam_relperm_evaluator(property_evaluator_iface):
    def __init__(self):
        super().__init__()

    def evaluate(self, state):
        steam_saturation = iapws_steam_saturation_evaluator()
        steam_rp = steam_saturation.evaluate(state) ** 1
        return steam_rp
```

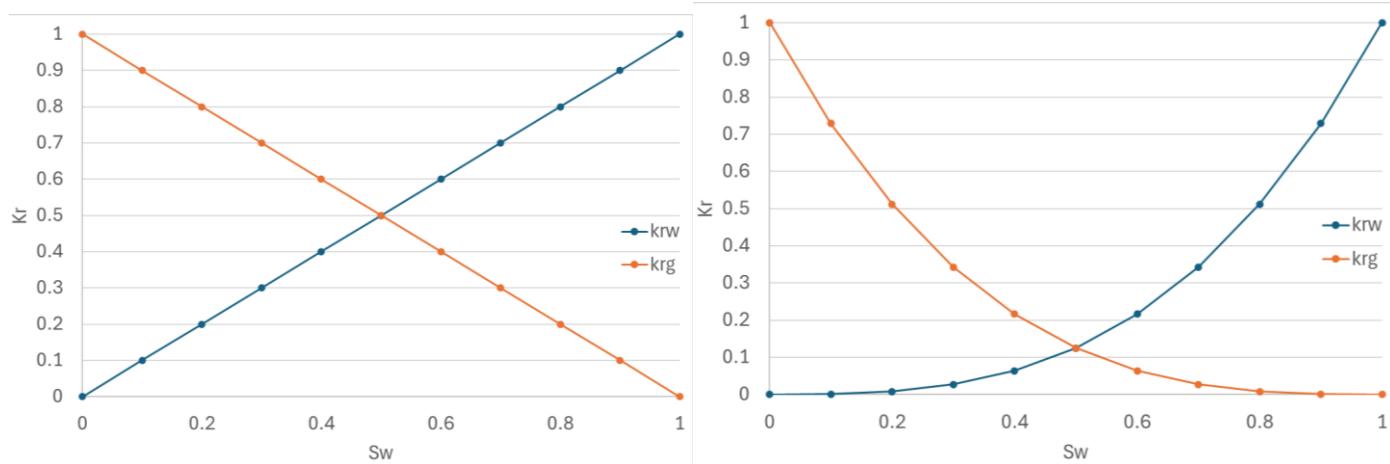
```
class iapws_water_relperm_evaluator(property_evaluator_iface):
    def __init__(self):
        super().__init__()

    def evaluate(self, state):
        water_saturation = iapws_water_saturation_evaluator()
        water_rp = water_saturation.evaluate(state) ** 3
        return water_rp

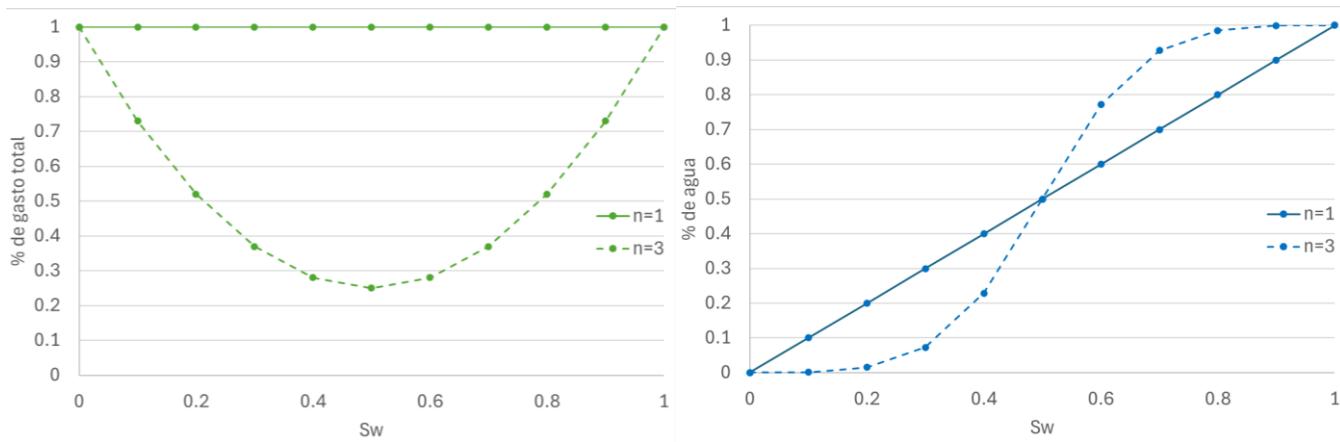
class iapws_steam_relperm_evaluator(property_evaluator_iface):
    def __init__(self):
        super().__init__()

    def evaluate(self, state):
        steam_saturation = iapws_steam_saturation_evaluator()
        steam_rp = steam_saturation.evaluate(state) ** 3
        return steam_rp
```

Las curvas de permeabilidad relativa de cada modelo se muestran a continuación:



Es importante notar el efecto que tiene cada modelo sobre el gasto esperado: cuando se utiliza un valor de $n=1$, la suma de las permeabilidades relativas siempre es uno. Es decir, la movilidad que pierde una fase lo gana la otra. Sin embargo, cuando se usa un valor de $n=3$, la suma de las permeabilidades relativas no es igual a uno, por lo que el gasto total se reduce. Es decir, ambas fases pierden movilidad. Las gráficas mostradas a continuación muestran este efecto



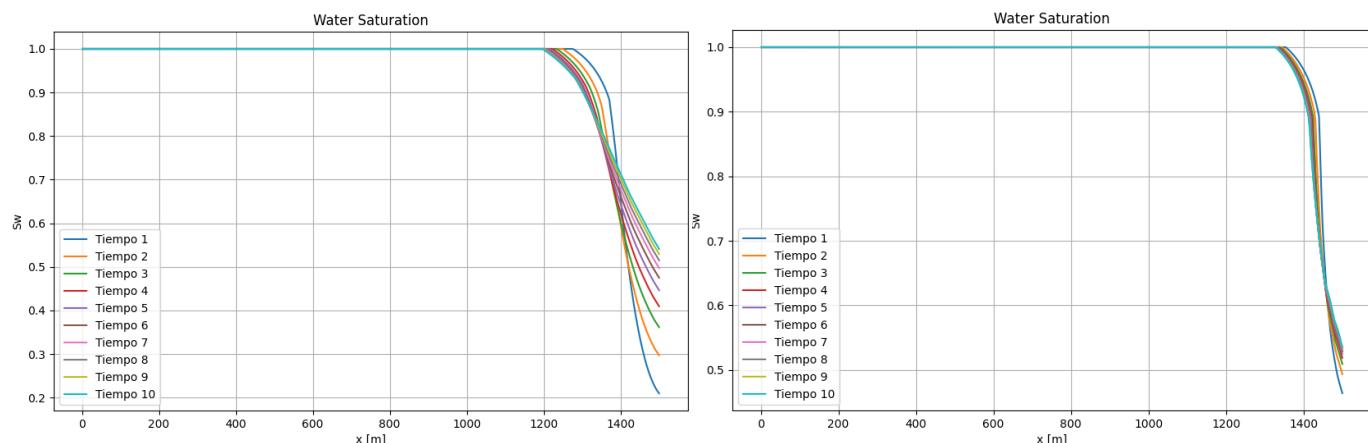
Para la simulación, se utiliza una viscosidad constante para ambas fases, con el fin de que la movilidad de cada fase dependa únicamente de la permeabilidad relativa y poder comparar su efecto en los resultados de la simulación.

```
property_container.viscosity_ev = dict([('water', ConstFunc(0.02)), # [cP]
                                         ('steam', ConstFunc(0.02))]) # [cP]
```

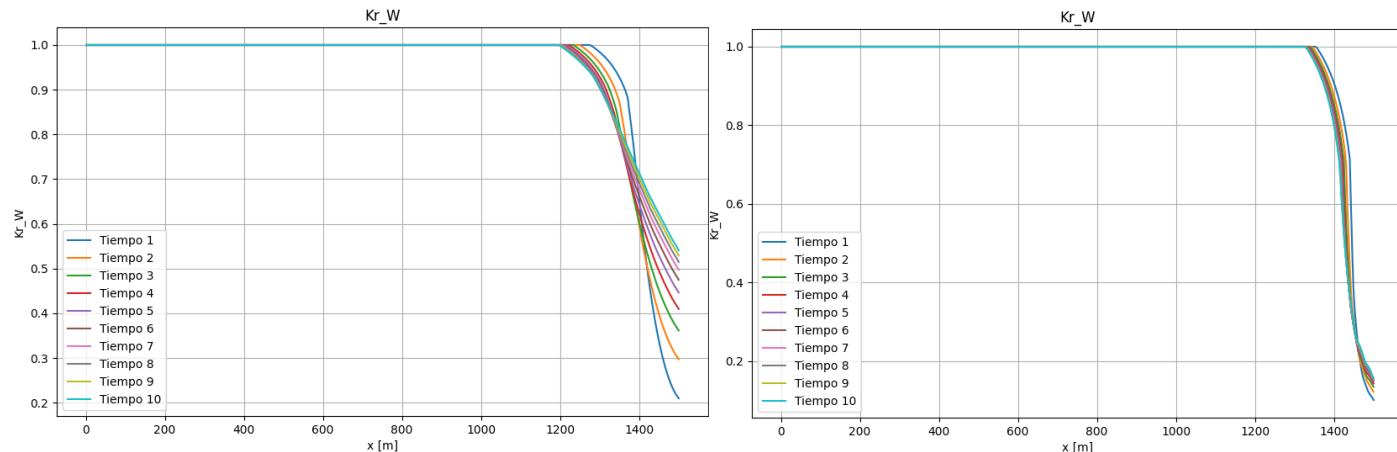
Se corre la simulación por 500 días, mostrando los resultados cada 50 días:

```
time=50
# # Correr simulaciones
for t in range(10): #
    m.run(time) #
    resultados = np.array(m.physics.engine.X, copy=False)
    # Extraer los valores de presión (índices impares)
    pressure = resultados[:,m.nb * 2:2]
    # Extraer los valores de entalpía (índices pares)
    enthalpy = resultados[1:m.nb * 2:2]
    # Agregar curva de presión
    ax1.plot(x, pressure, label=f'Dia {(t+1)*time}', alpha=0.7)
    # Agregar curva de entalpía
    ax2.plot(x, enthalpy / 18.015, label=f'Dia {(t+1)*time}', alpha=0.7)
```

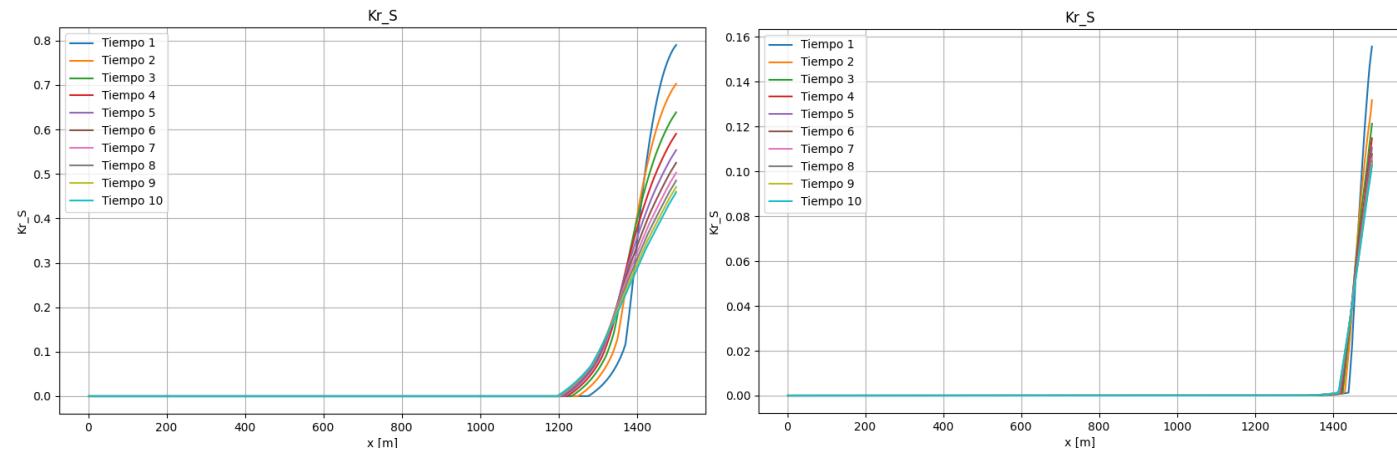
A continuación, se muestran los resultados obtenidos para cada modelo. Las figuras del lado izquierdo y derecho muestran los resultados usando el primer modelo y segundo modelo, respectivamente. Los perfiles de saturación de agua obtenidos son:



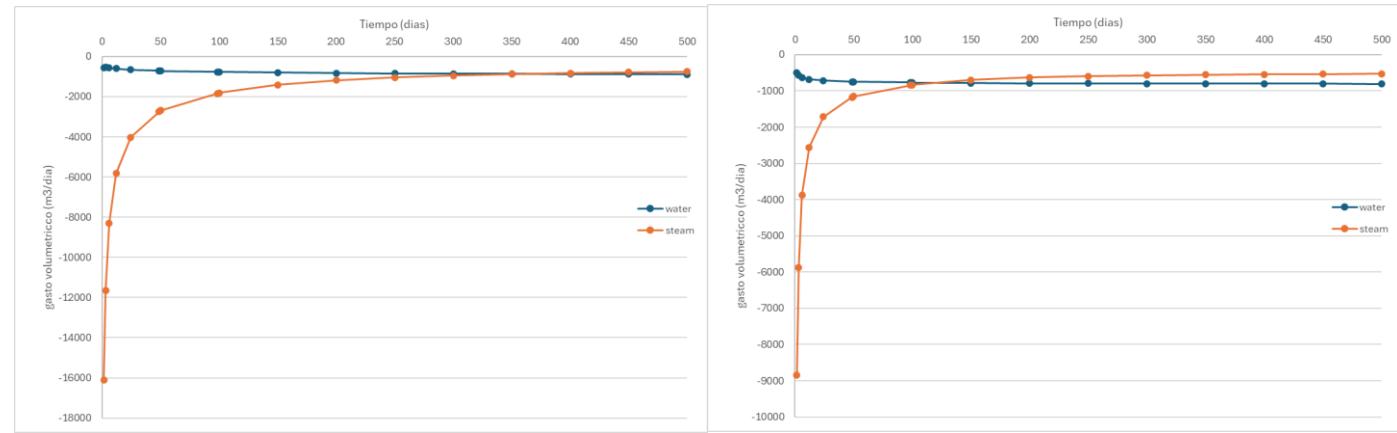
Los perfiles de permeabilidad relativa de la fase agua obtenidos son:



Los perfiles de permeabilidad relativa de la fase vapor obtenidos son:

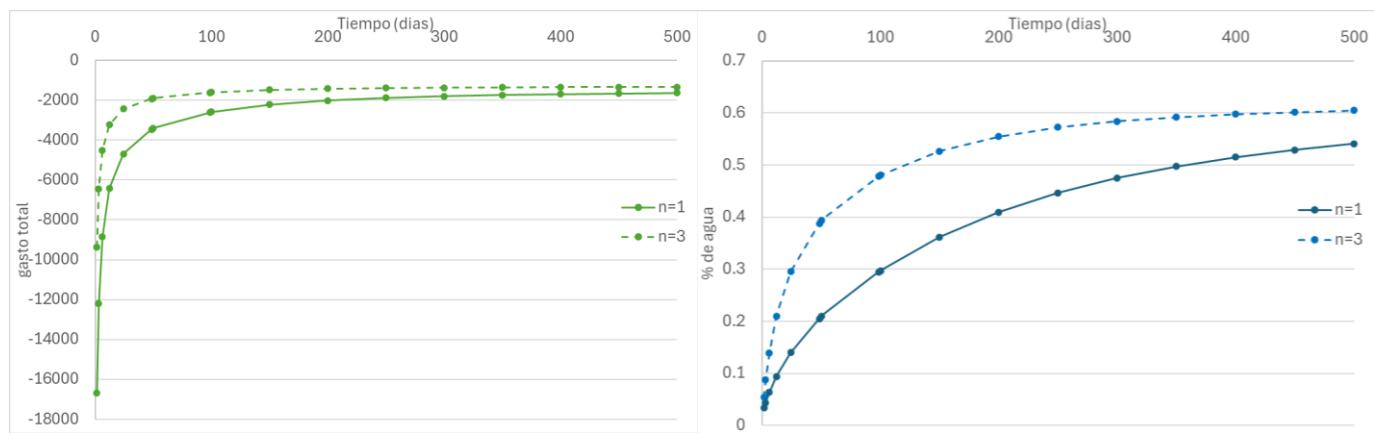


A continuación, se muestra el gasto volumétrico del pozo para de cada fase



En ambos casos la producción volumétrica inicial de la fase vapor es mucho mayor que la del agua. Sin embargo, llega el momento en que la producción de agua es mayor. Este punto concuerda con el tiempo en que la permeabilidad relativa del agua en las cercanías del pozo supera el valor de 0.5, el cual es diferente para cada modelo: en el modelo n=1 esto ocurre a los 350 días, mientras que en el modelo n=3 ocurre alrededor de los 120 días.

La grafica de la izquierda muestra el gasto total para cada uno de los modelos, mientras que la grafica de la derecha muestra el % de agua en la celda del pozo productor.



Podemos ver que, efectivamente, el gasto total (agua + vapor) del pozo productor en el modelo n=3 es siempre menor al del modelo n=1.

Ejemplo 7

Para este ejemplo, correremos una simulación en 2D, la cual tendrá 100x100x1 celdas. Para esto, modificamos la función set_reservoir en el archivo model.py:

```
self.Long_x=1000.    # [m]
self.Long_y=1000.    # [m]

(self.nx, self.ny, self.nz) = (100, 100, 1)
dx=self.Long_x/self.nx
dy=self.Long_y/self.ny

self.nb = self.nx * self.ny * self.nz
perm = np.ones(self.nb) * 1.
perm[:self.nx*self.ny] =5.    # [mD]

self.reservoir = StructReservoir(self.timer, nx=self.nx, ny=self.ny, nz=self.nz, dx=dx, dy=dy, dz=100.,
                                 permx=perm, permxy=perm, permz=perm, poro=.01,
                                 hcap=2470., rcond= 172.8 )
```

Establecemos el tiempo final:

```
end_time = 30                      # End time of the simulation (years)
```

Para visualizar, le pediremos a DARTS que exporte los archivos de salida .vtk o .vts. Primero, damos nombre a la carpeta que contendrá los archivos:

```
# output initial solution to vtk file
output_dir_base = 'vtk_output'
```

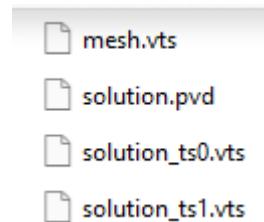
Se agregan las variables secundarias a las principales, para que al momento de imprimir los archivos .vts contengan todas las variables de interés:

```
prop_list = m.physics.vars + m.output.properties
```

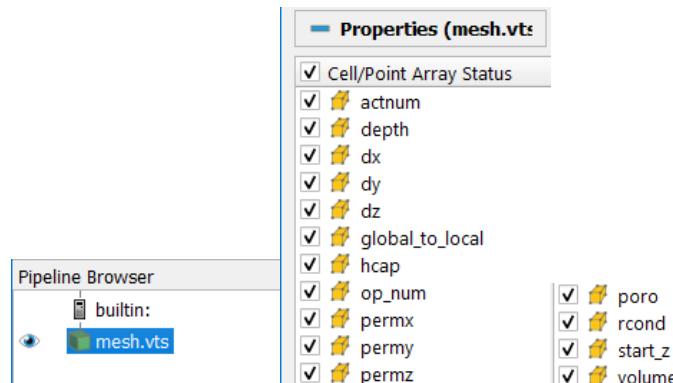
Corremos la simulación e imprimimos:

```
m.run(end_time*365)                  # End time of the simulation (days)
caso= '1'
output_dir = output_dir_base + caso
m.output.output_to_vtk(output_directory=output_dir, output_properties=prop_list)
```

Esto creara 4 archivos (la malla, y los resultados para tiempo inicial y final, además de un archivo .pvd que contiene la información de todos los pasos de tiempo):



El archivo mesh.vts contiene las propiedades físicas de cada celda de la malla. Al abrirlos en Paraview:

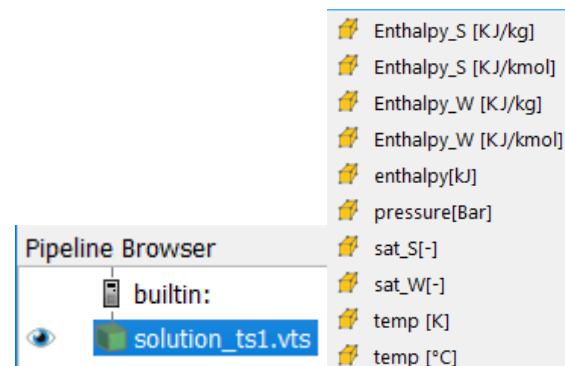


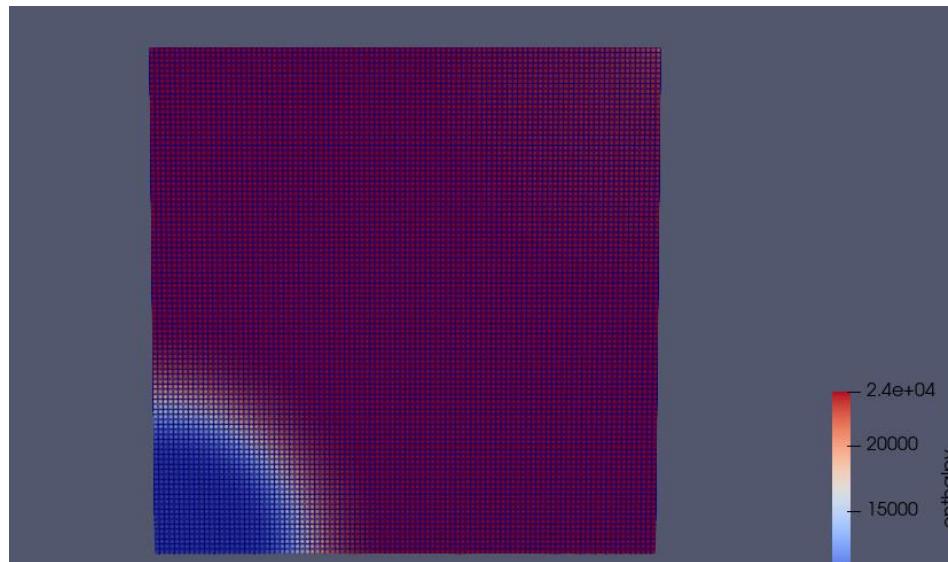
Si abrimos el archivo “solution.vts” vemos que contienen la solución de las variables principales, así como de las variables secundarias. Es importante recordar que la entalpia usada como variable principal en la simulación es la entalpia de la mezcla. Aquí se muestra como “Ethalpy[KJ]” (sus unidades en realidad son [KJ/mol]):

$$h_{\text{mezcla}} = (1 - x) h_f(T) + x h_g(T) = h_f(T) + x h_{fg}(T)$$

- $h_f(T)$: entalpía específica del líquido saturado (agua).
- $h_g(T)$: entalpía específica del vapor saturado.
- (Opcional) $h_{fg}(T) = h_g - h_f$.

X= fracción molar del vapor (no confundir con “sat_S”, que representa la fracción volumétrica del vapor)





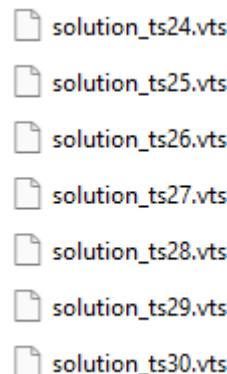
Si queremos imprimir los resultados a diferentes tiempos (por ejemplo, cada año), corremos la simulación 30 veces, con un tiempo de 1 año para cada corrida:

```
end_time = 30 # End time of the simulation (years)

caso= '2'
output_dir = output_dir_base + caso

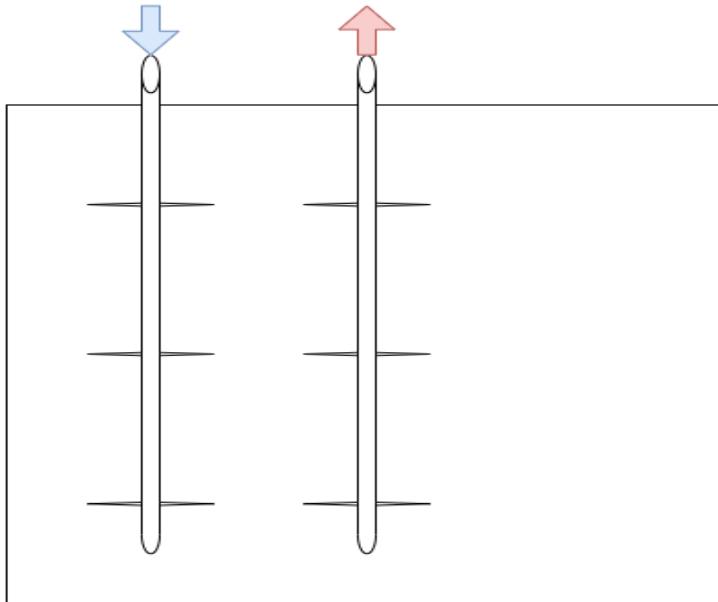
# Correr simulaciones
for t in range(end_time):
    m.run(365)
    m.output.output_to_vtk(output_directory=output_dir, output_properties=prop_list)
```

Esto creara 30 archivos:



Ejemplo 8

En este ejemplo se establecerán 2 pozos, uno inyector y otro productor, cada uno con 3 zonas de disparos o perforaciones, las cuales a su vez tendrán sus propios valores de daño.



Para esto, al momento de definir el pozo utilizamos el método `add_perforation` tres veces (la separación de los disparos será de 5 bloques. Para el pozo inyector, todos los disparos tendrán el mismo valor (cero, sin daño ni estimulación):

```
# add well ----# 1
self.reservoir.add_well("INJ", wellbore_diameter=0.3048)
self.reservoir.add_perforation("INJ", cell_index=(10, 10, 1), skin=0.0,
                                well_radius=0.1524, multi_segment=False,well_indexD=0)
self.reservoir.add_perforation("INJ", cell_index=(10, 25, 1), skin=0.0,
                                well_radius=0.1524, multi_segment=False,well_indexD=0)
self.reservoir.add_perforation("INJ", cell_index=(10, 40, 1), skin=0.0,
                                well_radius=0.1524, multi_segment=False,well_indexD=0)
```

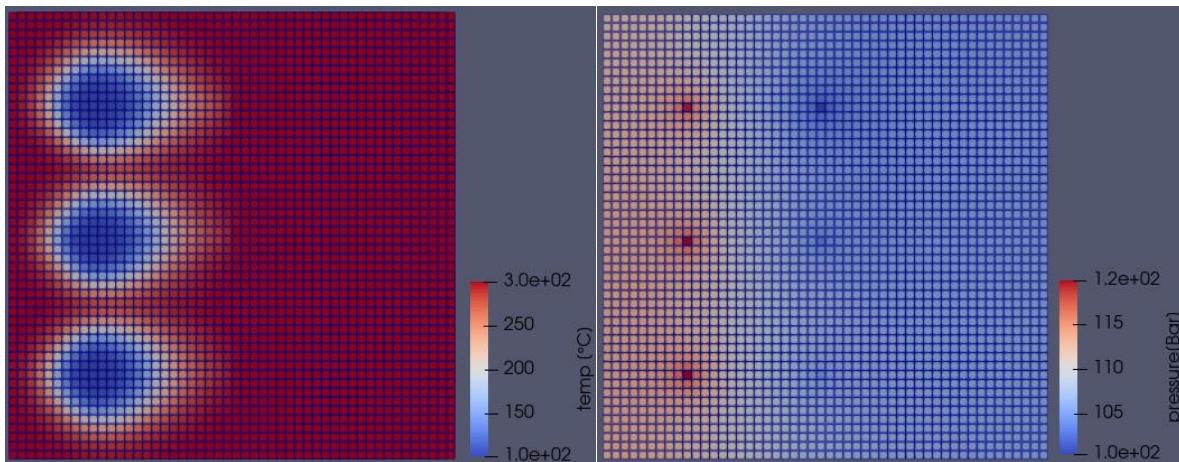
El 1º disparo del pozo productor tendrá un valor positivo (pozo dañado) mientras que el 3º disparo tendrá un valor negativo (pozo estimulado)

```
# add well ----# 2
self.reservoir.add_well("PRD", wellbore_diameter=0.3048)
self.reservoir.add_perforation("PRD", cell_index=( 25, 10, 1), skin=3.0,
                                well_radius=0.1524, multi_segment=False,well_indexD=0)
self.reservoir.add_perforation("PRD", cell_index=( 25, 25, 1), skin=0.0,
                                well_radius=0.1524, multi_segment=False,well_indexD=0)
self.reservoir.add_perforation("PRD", cell_index=( 25, 40, 1), skin=-3.0,
                                well_radius=0.1524, multi_segment=False,well_indexD=0)
```

Los valores de daño afectan los valores de índice de productividad calculados::

???????

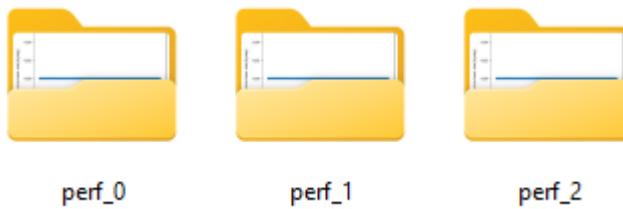
Corremos la simulación por 30 años. Si vemos las gráficas de temperatura y presión, podemos observar que efectivamente existen tres zonas de inyección (y de producción)



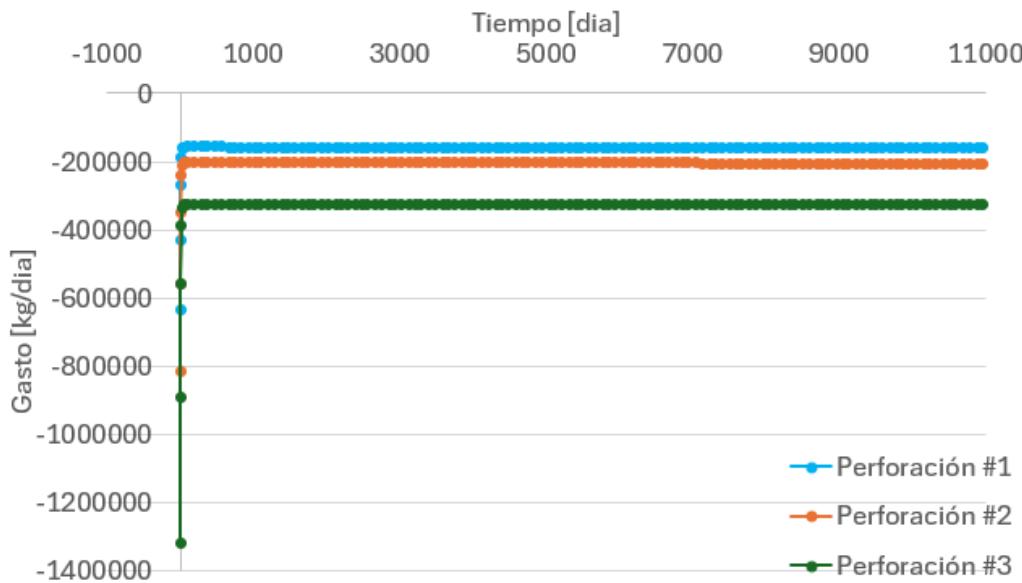
Después de correr la simulación, podemos imprimir la información de los pozos y los disparos mediante:

```
# Escribimos los del pozo y ploteamos
m.output.store_well_time_data( save_output_files=True)
m.output.plot_well_time_data()
```

En la ubicación \output\figures\well_time_plots se ploteará la información de cada pozo. Como en este caso se establecieron tres perforaciones:



Por ejemplo, si comparamos el gasto de producción de una sola de las perforaciones (se plotean los valores mayores a 1 dia):



El daño provocara que, para un mismo gradiente de presión, se produzca un gasto menor, mientras que la estimulacion aumentara dicho gasto.

También podemos activar, desactivar y modificar pozos a diferentes tiempos de la simulación. Por ejemplo, en el caso #2, el 1º pozo se cerrará después de 15 años de producción y se agregará un 3º pozo (también de inyección) en el otro extremo del yacimiento. Es importante mencionar que, aunque un pozo se activa en un tiempo posterior, debemos definirlo desde el inicio (al momento de inicializar el modelo):

```
# add well -----# 3
self.reservoir.add_well("INJ_2", wellbore_diameter=0.3048)
self.reservoir.add_perforation("INJ_2", cell_index=( 40, 10, 1), skin=0.0,
                                well_radius=0.1524, multi_segment=False,well_indexD=0)
self.reservoir.add_perforation("INJ_2", cell_index=( 40, 25, 1), skin=0.0,
                                well_radius=0.1524, multi_segment=False,well_indexD=0)
self.reservoir.add_perforation("INJ_2", cell_index=( 40, 40, 1), skin=0.0,
                                well_radius=0.1524, multi_segment=False,well_indexD=0)
```

Como queremos que el 3º pozo se active hasta después de 15 años, simplemente se le asigna un gasto molar de cero al momento de inicializar el modelo.:

```
#----- pozo 3
self.physics.set_well_controls(wctrl=self.reservoir.wells[2].control,
                                control_type=well_control_iface.MOLAR_RATE,
                                is_inj=True,
                                target=0.0,
                                phase_name='water',
                                inj_temp=inj_temp + 273.15 # [c]-----[k]
                                )
```

Se corre la simulación, utilizando un condicional “if” para cerrar y activar los pozos deseados:

```
# se modifica el 1º pozo para que inyecte agua a una temperatura diferente
from darts.engines import well_control_iface
Paso_de_tiempo= 365
tiempo_inicial=0
tiempo_total=0

# Correr simulaciones
for t in range(end_time):

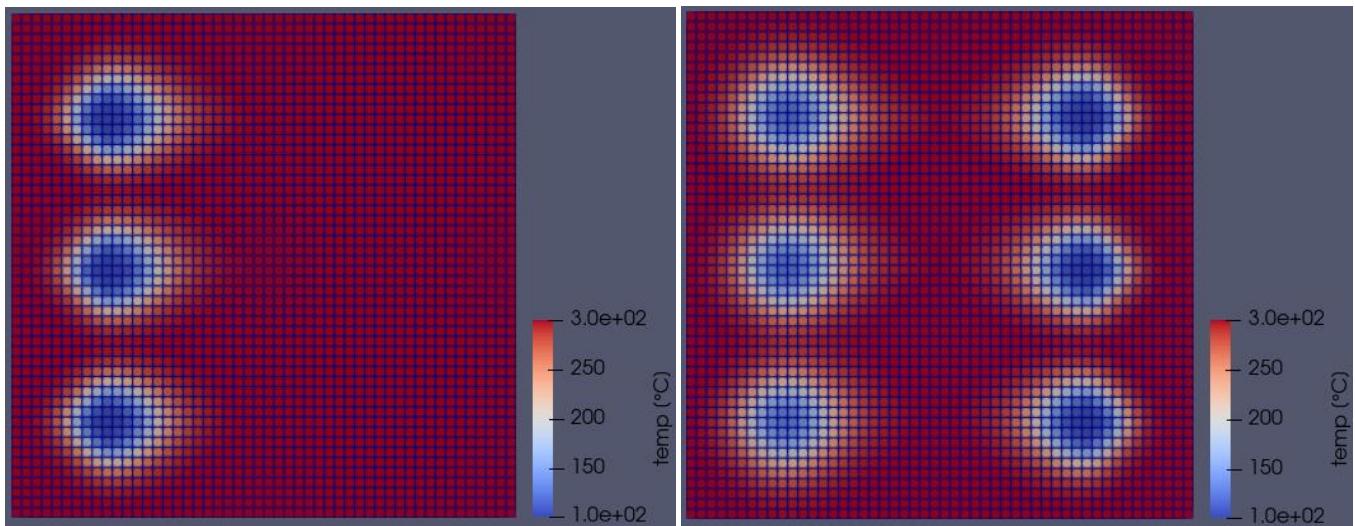
    m.run(Paso_de_tiempo)
    m.output.output_to_vtk(output_directory=output_dir, output_properties=prop_list)
    tiempo_total=tiempo_total + Paso_de_tiempo

    # A los 15 años
    if tiempo_total==365*15:

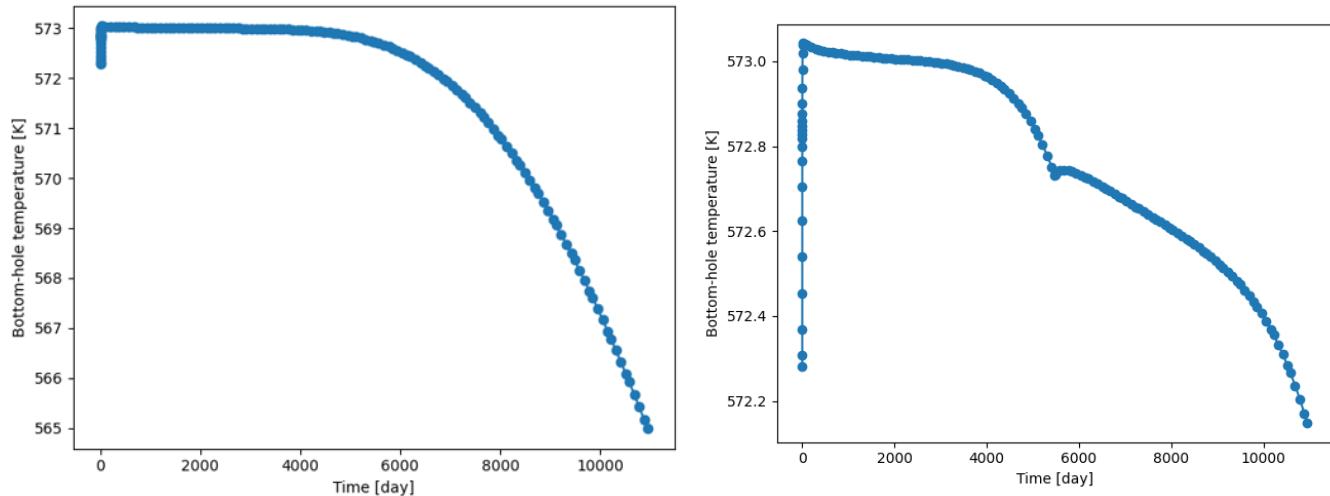
        # se cierra el 1º pozo
        mass_rate=0.0                      # [Kg/s]
        mol_rate=mass_rate*4795.928784     # [Kg/s] -----> [kmol/day]
        inj_temp= 100                      # [c]----[k]
        m.physics.set_well_controls(wctrl=m.reservoir.wells[0].control,
                                      control_type=well_control_iface.MOLAR_RATE,
                                      is_inj=True,
                                      target=mol_rate,
                                      phase_name='water',
                                      inj_temp=inj_temp + 273.15  # [c]----[k]
                                    )

# Se activa el 3º pozo para que inyecte agua
mass_rate=8.0                          # [Kg/s]
mol_rate=mass_rate*4795.928784         # [Kg/s] -----> [kmol/day]
inj_temp= 100                          # [c]----[k]
m.physics.set_well_controls(wctrl=m.reservoir.wells[2].control,
                            control_type=well_control_iface.MOLAR_RATE,
                            is_inj=True,
                            target=mol_rate,
                            phase_name='water',
                            inj_temp=inj_temp + 273.15  # [c]----[k]
                          )
```

Resultados para 15 y 30 años:



Si comparamos la temperatura de fondo de pozo productor para ambos casos:



Vemos que, para el segundo caso, la caída en la temperatura del fluido producido no es tan pronunciada como en el primer caso.

Ejemplo 9

En este ejemplo se utiliza y analiza el efecto de fronteras abiertas en el yacimiento. Para facilitar los datos de entrada, se crea un pequeño archivo .py que contiene el caso que vamos a simular:

```
Select_case.py X
Ejemplo_9 > Select_case.py > ...
1 # case = '1'      # ----- frontera cerrada
2 # case = '2'      # ----- frontera abierta
3
4 case = '1'
```

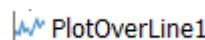
Se establece un solo pozo inyector, el cual se ubica en el centro del dominio (sin pozo productor):

```
def set_wells(self):
    # add well
    self.reservoir.add_well("INJ", wellbore_diameter=0.3048)
    self.reservoir.add_perforation("INJ", cell_index=(50, 50, 1), skin=0.0,
                                   well_radius=0.1524, multi_segment=False,well_indexD=0)
```

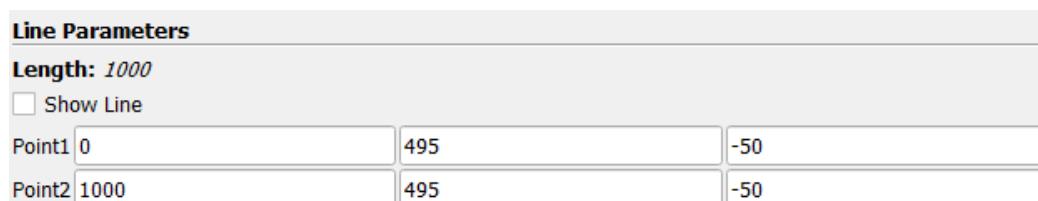
Agregamos una frontera abierta del lado izquierdo del modelo (se establece un volumen excesivamente grande para las celdas izquierdas). Para esto, se utiliza el método “boundary volumes”

```
self.reservoir.boundary_volumes['yz_minus'] = 1e8
#self.reservoir.boundary_volumes['yz_plus'] = 1e8
#self.reservoir.boundary_volumes['xz_minus'] = 1e8
#self.reservoir.boundary_volumes['xz_plus'] = 1e8
#self.reservoir.boundary_volumes['xy_minus'] = 1e8
#self.reservoir.boundary_volumes['xy_plus'] = 1e8
```

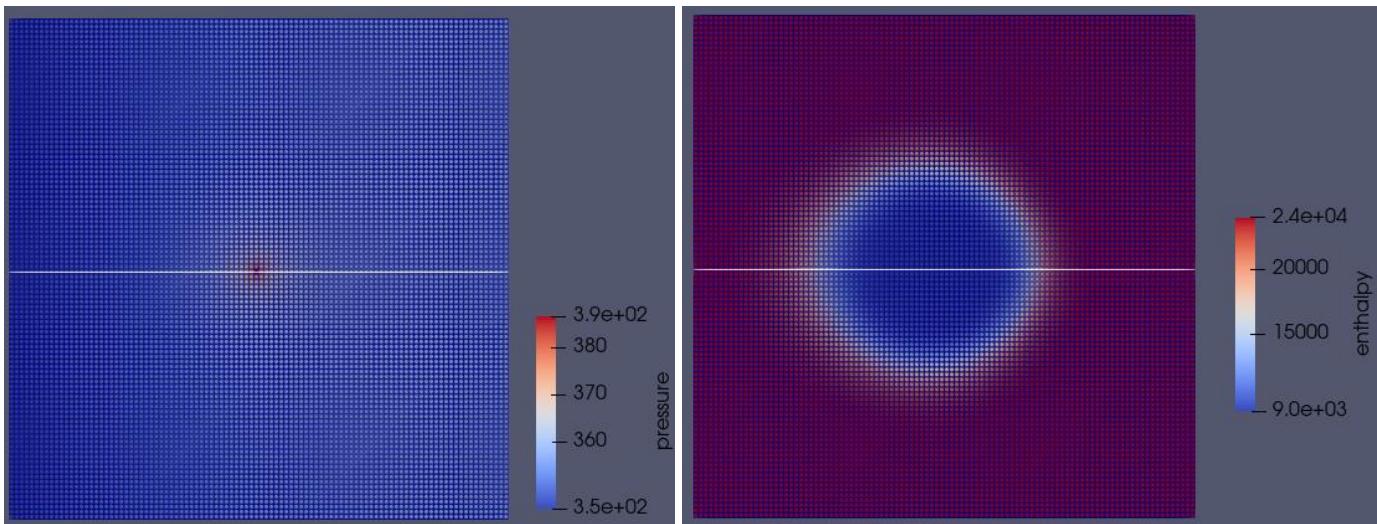
Corremos la simulación y abrimos el archivo en paraview. Agregamos una línea en el centro del modelo usando el filtro “plotoverline”:



Podemos plotear cualquiera de las variables sobre la línea creada. Las coordenadas de los dos puntos que forman la línea son:



Podemos visualizar la línea creada junto con los mapas de presión o entalpia:



A continuación, ploteamos la distribución de presión sobre esta línea, seleccionando la variable deseada dentro del submenu “series parameteres ” del menu “Display”

Display (XYChartRepresentation)

Attribute Type Point Data

X Axis Parameters

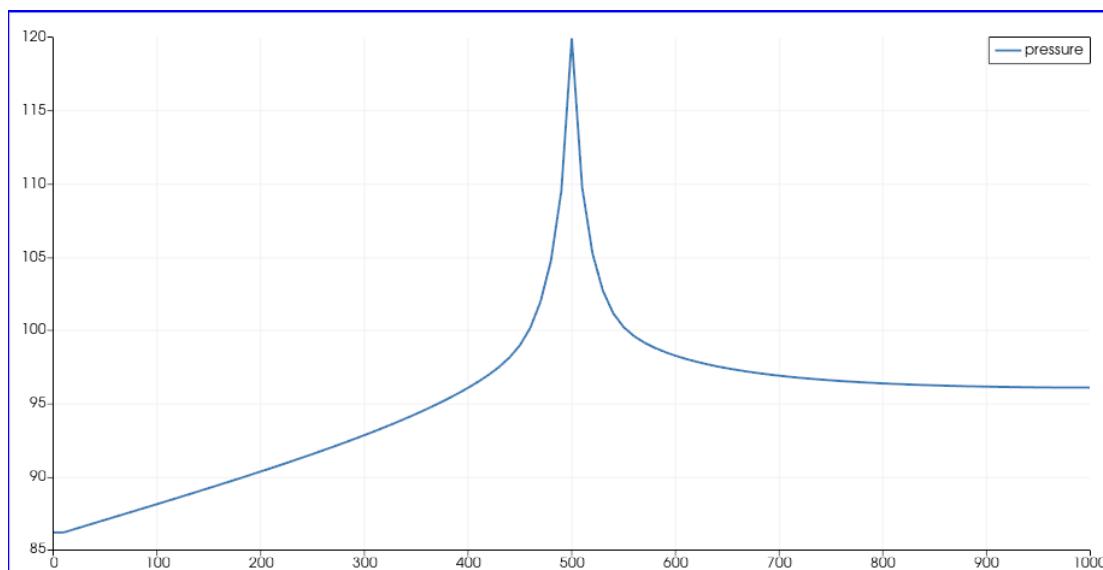
Use Index For XAxis

X Array Name arc_length

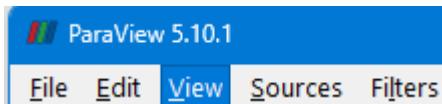
Series Parameters

	Variable	Legend Name
<input checked="" type="checkbox"/>	pressure	
<input type="checkbox"/>		

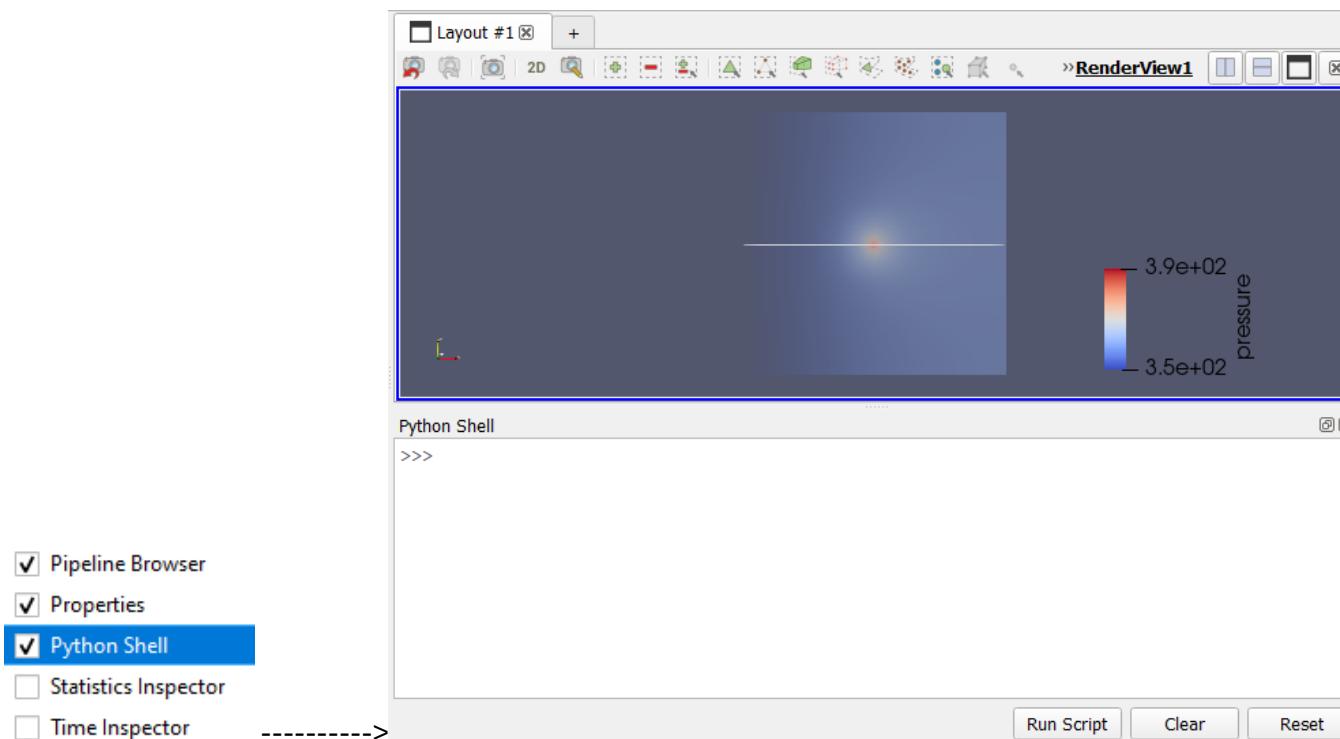
Al graficar, observamos un pico de presión en el centro, debido al pozo inyector. Del lado izquierdo, la presión es ligeramente menor en comparación al lado derecho, debido a la frontera abierta:



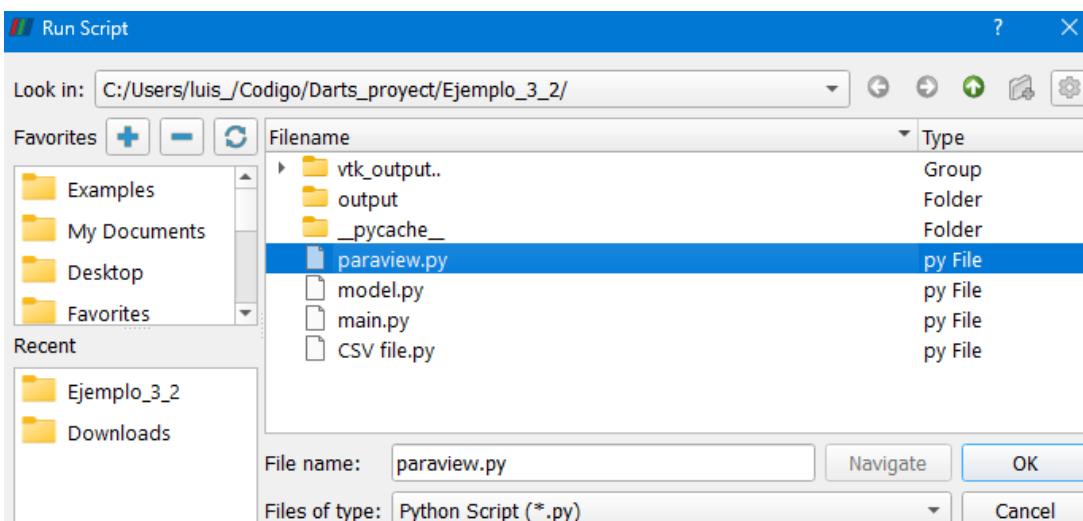
Se puede repetir el proceso para cada paso de tiempo de la simulación, es decir, para cada uno de los archivos .vts. Podemos automatizar este proceso utilizando un script de python. Para correr scripts de python en paraview, vamos a la opción de “view”:



Y activamos la opción de python shell, lo cual abrirá una terminal:



Damos click en el botón “Run script” y seleccionamos el archivo “paraview.py”:



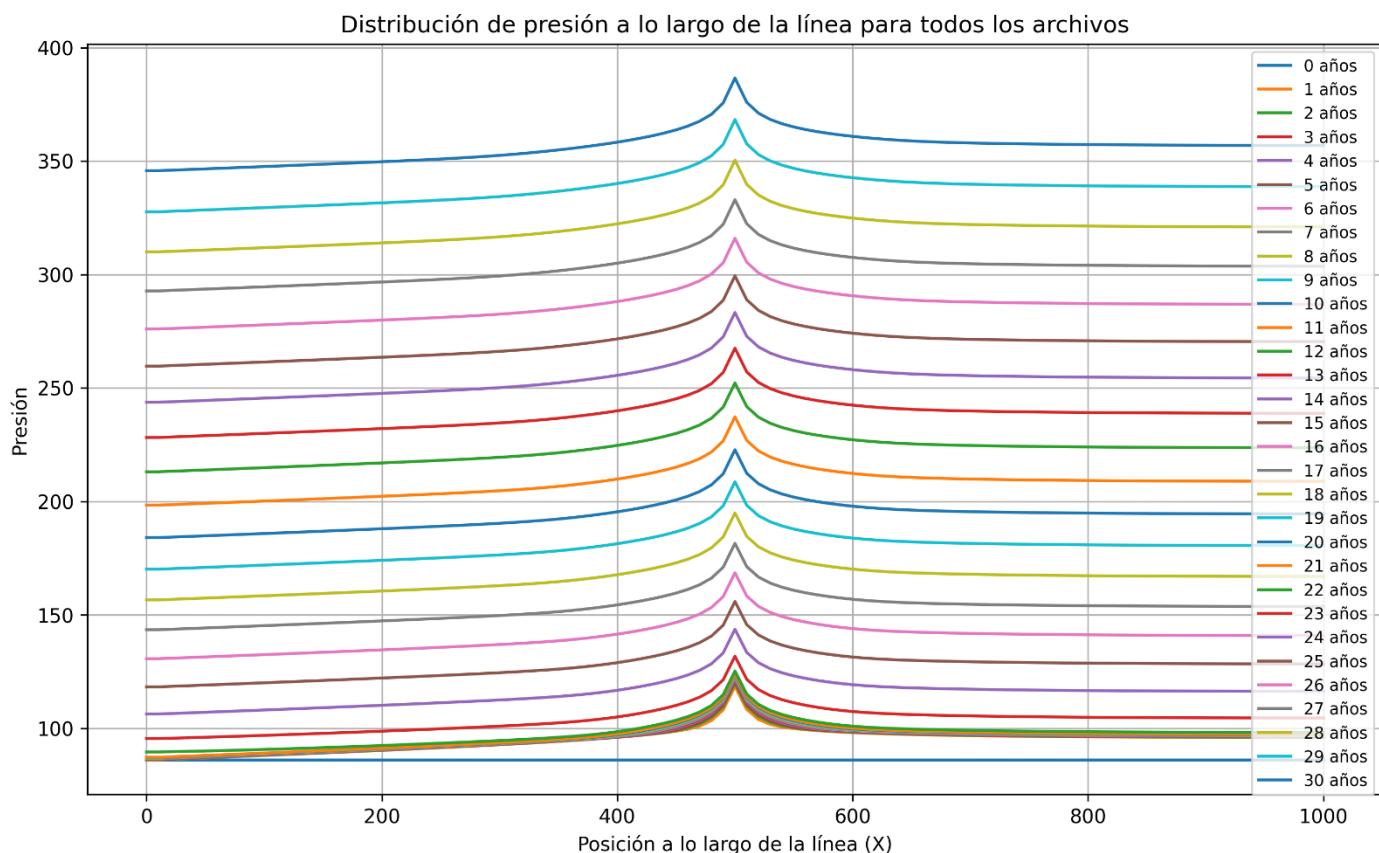
Esto creará un archivo .csv que contiene los datos de presión sobre la línea creada, para cada paso de tiempo:

 presion_sobre_linea_1.csv

Podemos correr un segundo script (fuera de paraview) para leer este archivo y plotear la información:

 CSV file.py

Al correrlo, se obtendrá el perfil de presión para todos los tiempos. En teoría, la presión global del yacimiento debería aumentar rápidamente conforme el fluido es inyectado, debido a que no existe un pozo productor (se trata de agua, un fluido incompresible). Sin embargo, la presión global del yacimiento tarda aproximadamente 13 años en aumentar. Esto se debe a que las celdas del lado izquierdo son tan grandes que pueden “absorber” un gran volumen del agua inyectada antes de empezar a mostrar un aumento en la presión (frontera abierta).



En contraste, si dejamos todas las fronteras como cerradas y repetimos la simulación:

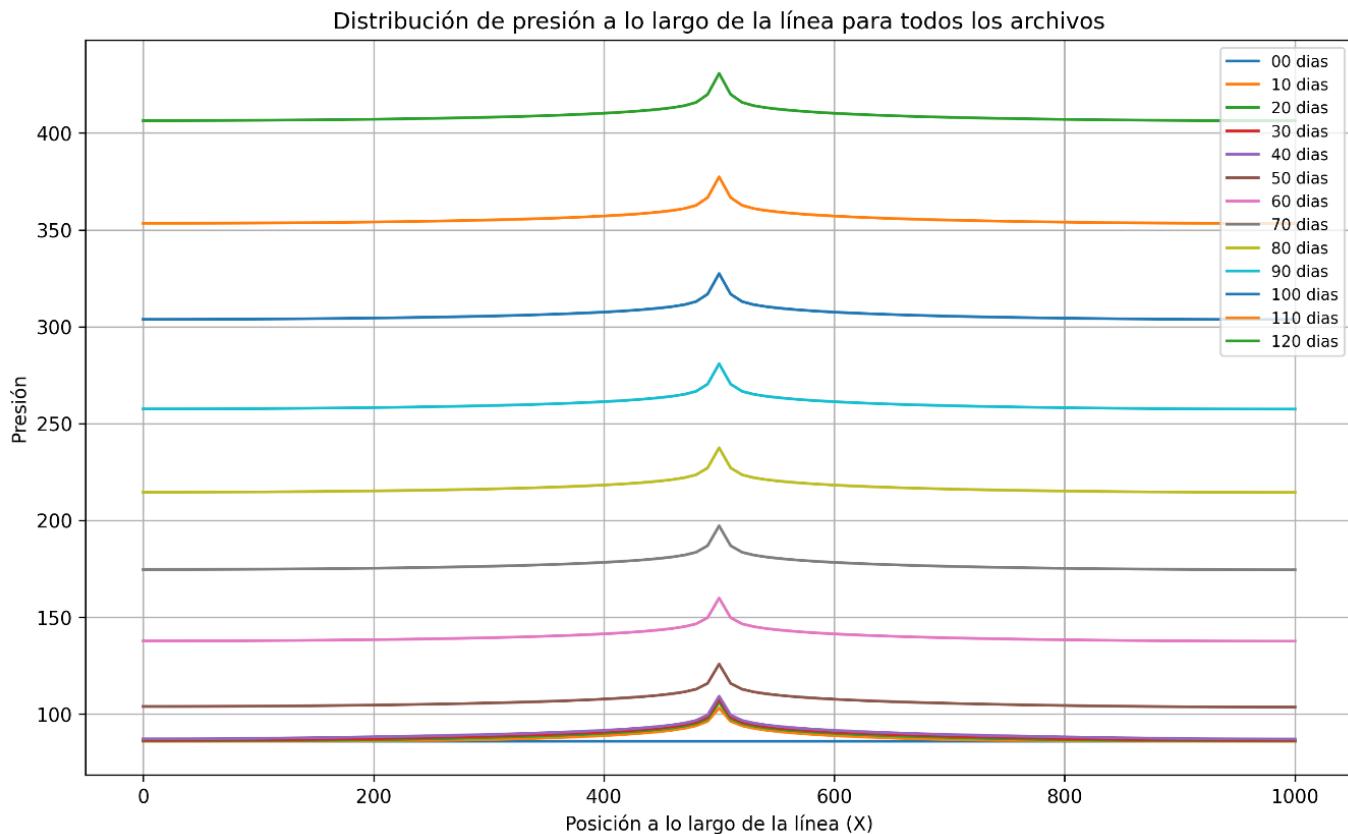
```

Select_case.py
main.py
model.py

Ejemplo_5 > Select_case.py > ...
1 # case = '1'      # ----- frontera cerrada
2 # case = '2'      # ----- frontera abierta
3
4 case = '2' |

```

Veremos que la presión global del yacimiento comienza a aumentar a los 40 días y sube mucho más rápido, tanto que la simulación se detiene a los 120 días debido a que se llega al límite superior de presión (considerando los límites establecidos para este ejemplo en específico). Además, existe una simetría en el perfil de presión obtenido, debido a que las fronteras en los extremos son iguales (fronteras cerradas)



Ejemplo 10

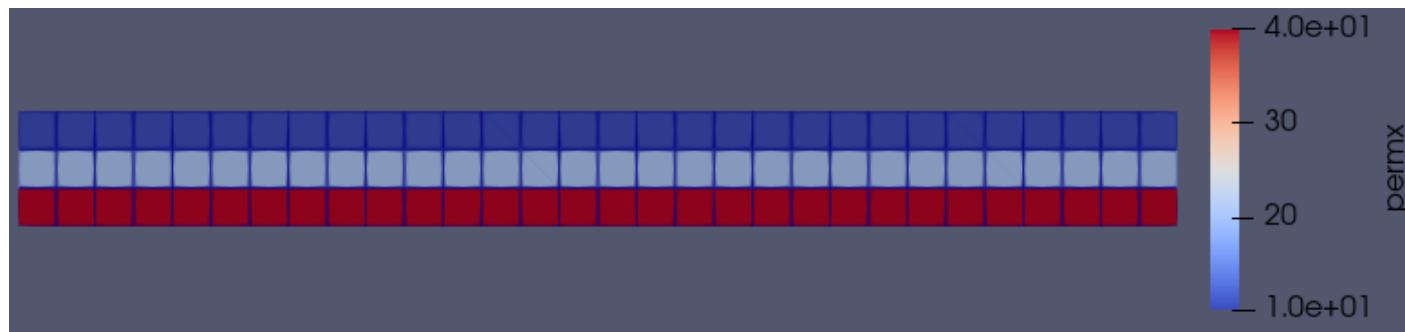
En este ejemplo, vamos a usar un modelo 3D y mostrar como asignar valores de permeabilidad específicos para cada capa, así como condiciones iniciales para cada capa. Se utilizarán 2 pozos, un inyector y un productor, ubicados en las esquinas opuestas del yacimiento. Vamos a crear 3 capas, cada una con un valor de permeabilidad diferente (la capa más profunda es la más permeable):

```
# Crear un arreglo vacío
perm = np.zeros(self.nb)

# Asignar los valores según las capas
perm[0:self.nx*self.ny] = 10          # 5 [mD]
perm[self.nx*self.ny:2*self.nx*self.ny] = 20    # 10 [mD]
perm[2*self.nx*self.ny:3*self.nx*self.ny] = 40    # 15 [mD]

self.reservoir = StructReservoir(self.timer, nx=self.nx, ny=self.ny, nz=self.nz, dx=self.dx, dy=self.dy, dz=self.dz,
                                 permx=perm, permy=perm, permz=perm, poro=.01,
                                 hcap=2470., rcond= 172.8 )
```

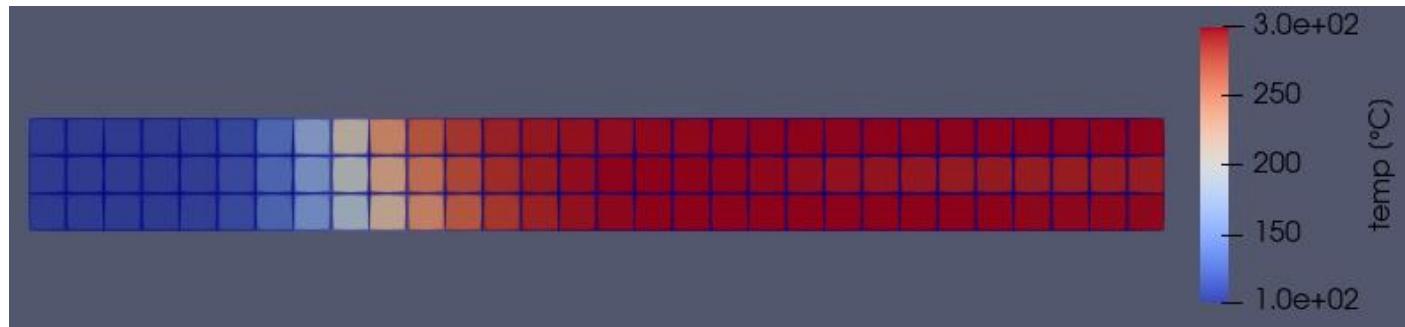
Si abrimos el archivo mesh.vts después de correr la simulación, veremos que efectivamente las permeabilidades se han establecido:



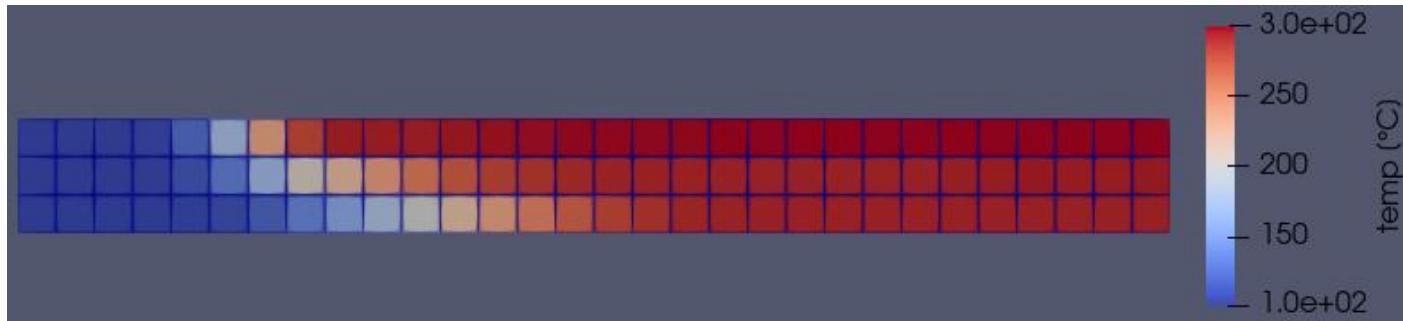
vista del plan X-Z

Si comparamos los resultados de temperatura con el caso con permeabilidades iguales:

Caso con permeabilidad igual:



Caso con permeabilidad variable:



También se pueden modificar los valores de permeabilidad una vez creado el objeto “reservoir”. Esto solo funciona si se modifican antes de inicializar el modelo

```
# LA PERMEABILIDAD SE DEBE CAMBIAR ANTES DE INICIALIZAR PARA QUE SURTA EFECTO !!!  
# new_perm = np.zeros((m.nx , m.ny))  
# new_perm[:] = 20.0  
  
# # Asignar solo la última capa  
# m.reservoir.global_data['permx'][ :, :, 0] = new_perm  
# m.reservoir.global_data['permy'][ :, :, 0] = new_perm  
# m.reservoir.global_data['permz'][ :, :, 0] = new_perm  
  
print('-----MODEL INITIALIZATION-----')  
print('\n')  
  
m.init()
```

Para establecer condiciones iniciales por capa se utiliza el siguiente método, el cual ocupa una tabla de profundidades, y las condiciones iniciales a esa profundidad:

```
self.physics.set_initial_conditions_from_depth_table(mesh=self.reservoir.mesh,  
                                                    input_distribution=input_distribution, input_depth=input_depth)
```

Sin embargo, se encontró que esta función tiene un pequeño error. Para implementarla correctamente, hacemos una copia de la función original y la modificamos (para evitar modificar el código base de DARTS). Para utilizar la versión corregida y no la original, hacemos uso de las siguientes librerías:

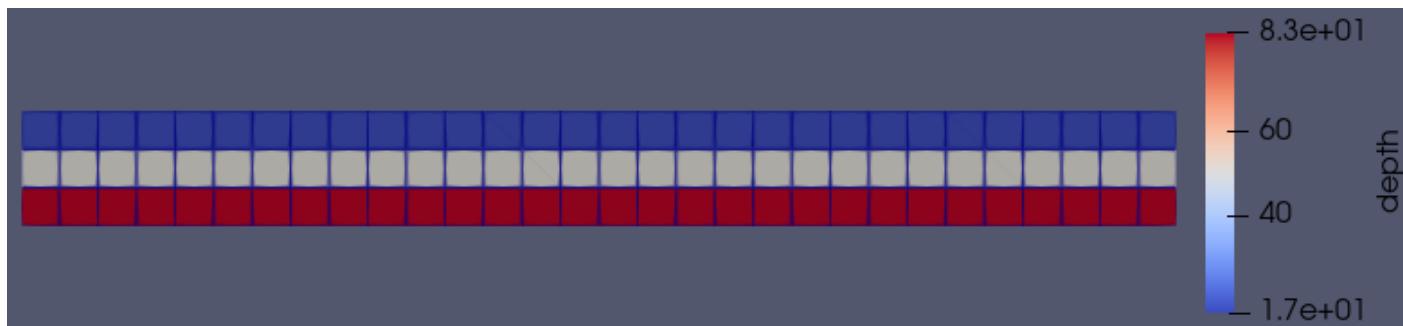
```
from scipy.interpolate import interp1d
```

Con esto, definimos la nueva función al final del archivo model.py:

```
def patched_set_initial_conditions_from_depth_table(...)

# 🔪 Parcheamos el método de Geothermal
Geothermal.set_initial_conditions_from_depth_table = (
    patched_set_initial_conditions_from_depth_table
)
```

La información de las condiciones iniciales (presión, por ejemplo) suele estar referenciadas a la profundidad del yacimiento. Sin embargo, la profundidad del yacimiento no se ha establecido. Si observamos la variable de profundidad (Depth) en el archivo mesh.vts vemos que corresponde a la profundidad relativa:



Para poder establecer una profundidad absoluta, agregamos el dato de entrada “start_z”

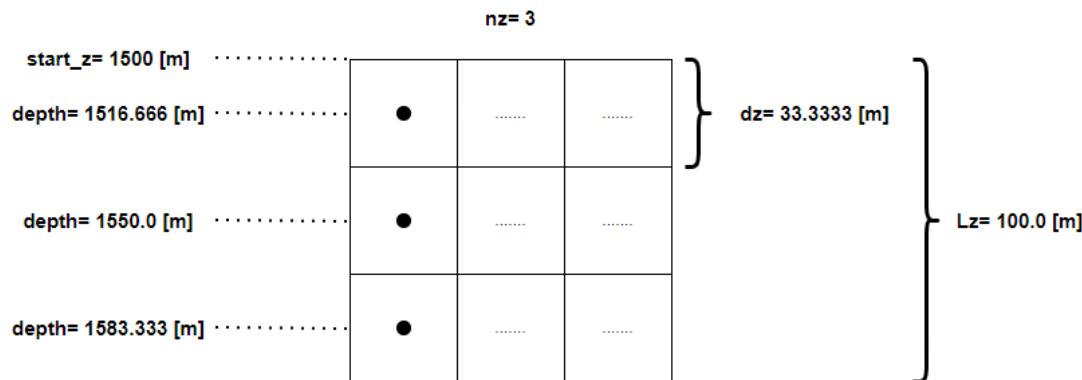
```

first_depth=1500.           # [m]  profundidad de la capa mas somera

self.reservoir = StructReservoir(self.timer, nx=self.nx, ny=self.ny, nz=self.nz, dx=self.dx, dy=self.dy, dz=self.dz,
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
permx=perm, permxy=perm, permz=perm, poro=.01, start_z=1500.,
hcap=2470., rcond= 172.8 )
#                                         start_z: top reservoir depth (a single float value or nx*ny values)

```

Este dato corresponde a la profundidad de la primera capa. Para nuestro caso, esto significa que:



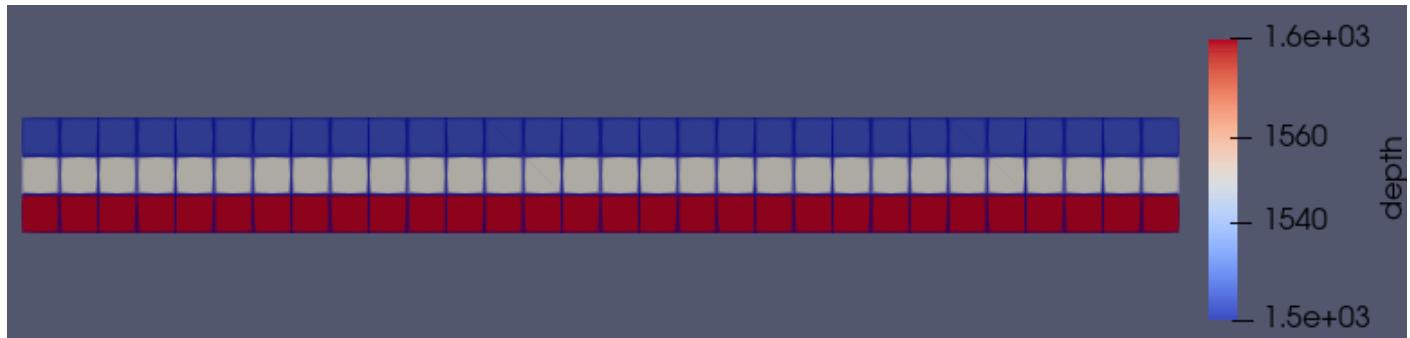
Una vez establecida la profundidad, podemos definir los valores iniciales de profundidad y temperatura (DARTS hará una interpolación para calcular los valores iniciales):

```
input_depth = [1516.666, 1550, 1583.333]

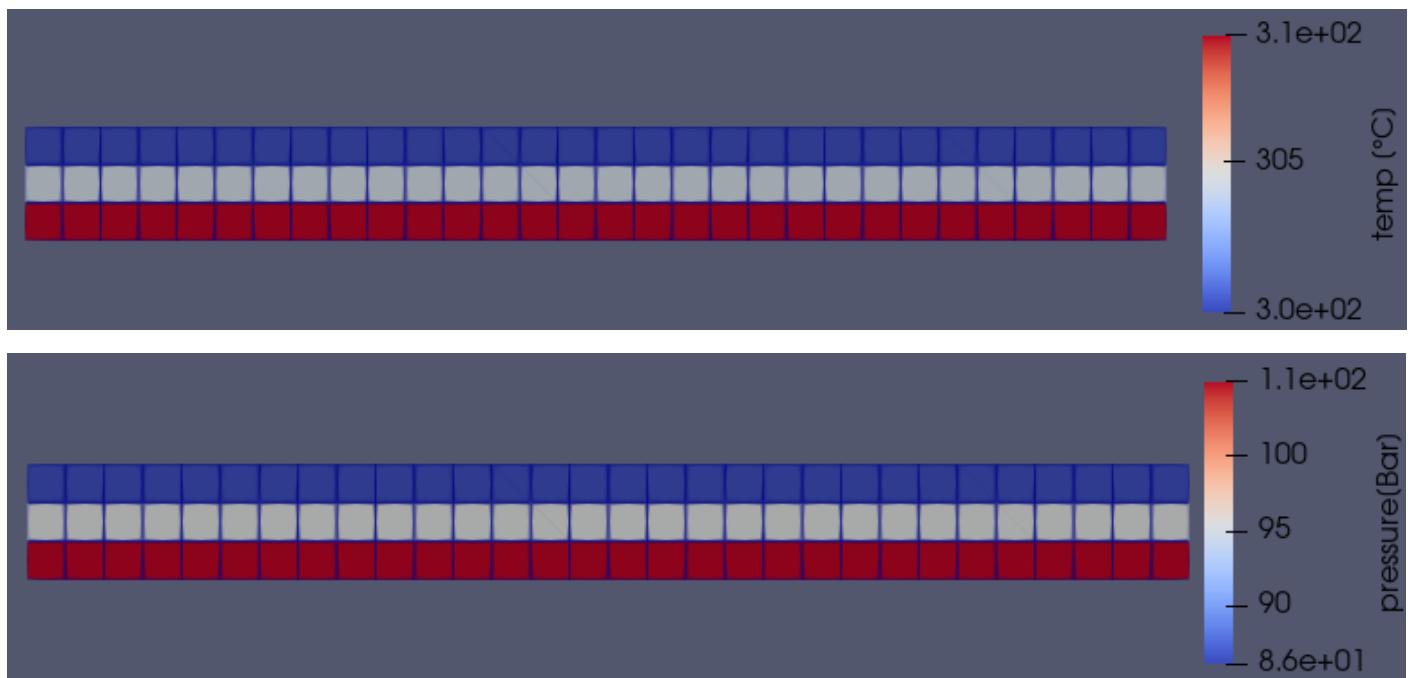
input_distribution = {
    "pressure": [86., 95, 105],      #
    "temperature": [300.0 + 273.15, 305.0 + 273.15, 310.0 + 273.15]  #
}

self.physics.set_initial_conditions_from_depth_table(mesh=self.reservoir.mesh,
                                                       input_distribution=input_distribution, input_depth=input_depth)
```

Después de correr la simulación, podemos verificar las profundidades:



Si vemos el archivo solution_ts0.vts para ver las condiciones iniciales, vemos que la presión y temperatura ya no son homogéneas:



Ejemplo 11

En este ejemplo haremos una simulación en estado natural, es decir, haremos una simulación sin efectos externos como fronteras abiertas o fuentes/sumideros (pozos). Esto ayudara a conocer el gradiente de presión natural que existe en el yacimiento. Para esto, se crea un yacimiento en 2D:

```
def set_reservoir(self):
    print('\n-----DEFINE RESERVOIR :')
    print('\n')
    #####
    self.nx, self.ny, self.nz = 50, 1, 100
    self.nb = self.nx * self.ny * self.nz # número total de bloques

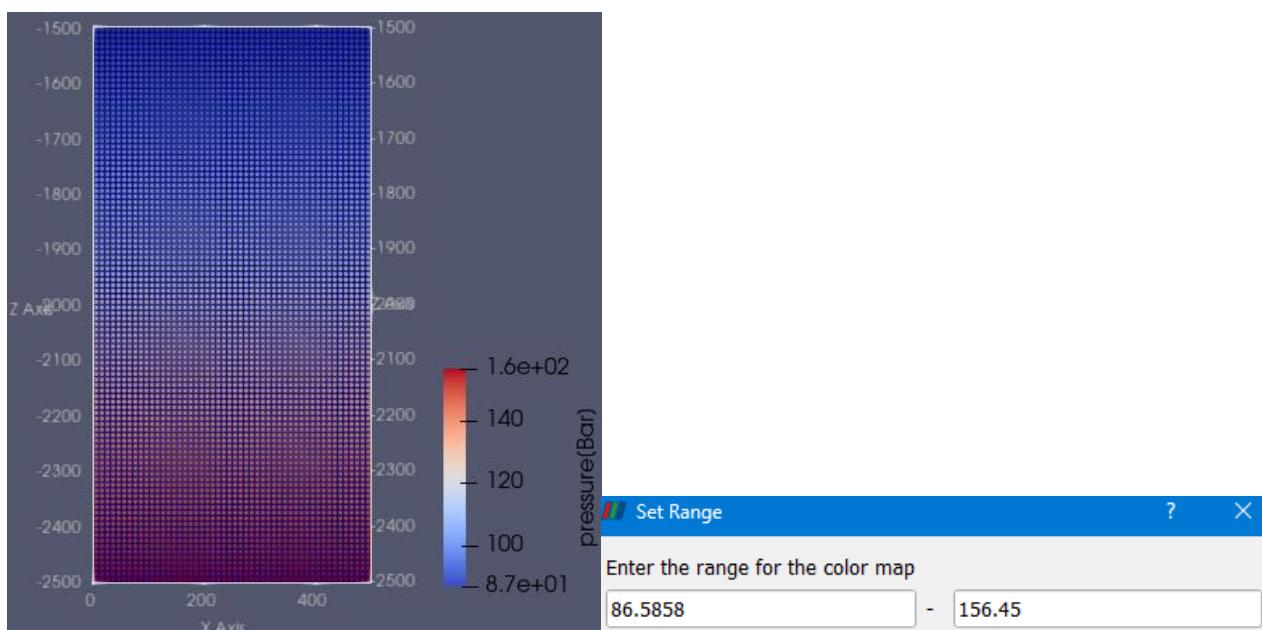
    Long_x=500. # [m]
    Long_y=20. # [m]
    Lony_z= 1000. # [m]

    self.dx=Long_x/self.nx
    self.dy=Long_y/self.ny
    self.dz=Lony_z/self.nz
```

Suponemos que solo existe agua líquida en el yacimiento, por lo que establecemos una presión uniforme de 100 [bar] y 300°C en todo el yacimiento. Esto equivale a tener solo agua líquida (sin vapor) dentro del medio:

```
# initialization with constant pressure and temperature
self.input_distribution = {"pressure": 100., # [bar]
                           "temperature": 300. + 273.15 # [c]----[k]
                           }
self.physics.set_initial_conditions_from_array(mesh=self.reservoir.mesh,
                                                input_distribution=self.input_distribution)
```

Corremos la simulación por 500 días (en realidad, el gradiente de presión se establece a poco más de 60 días):



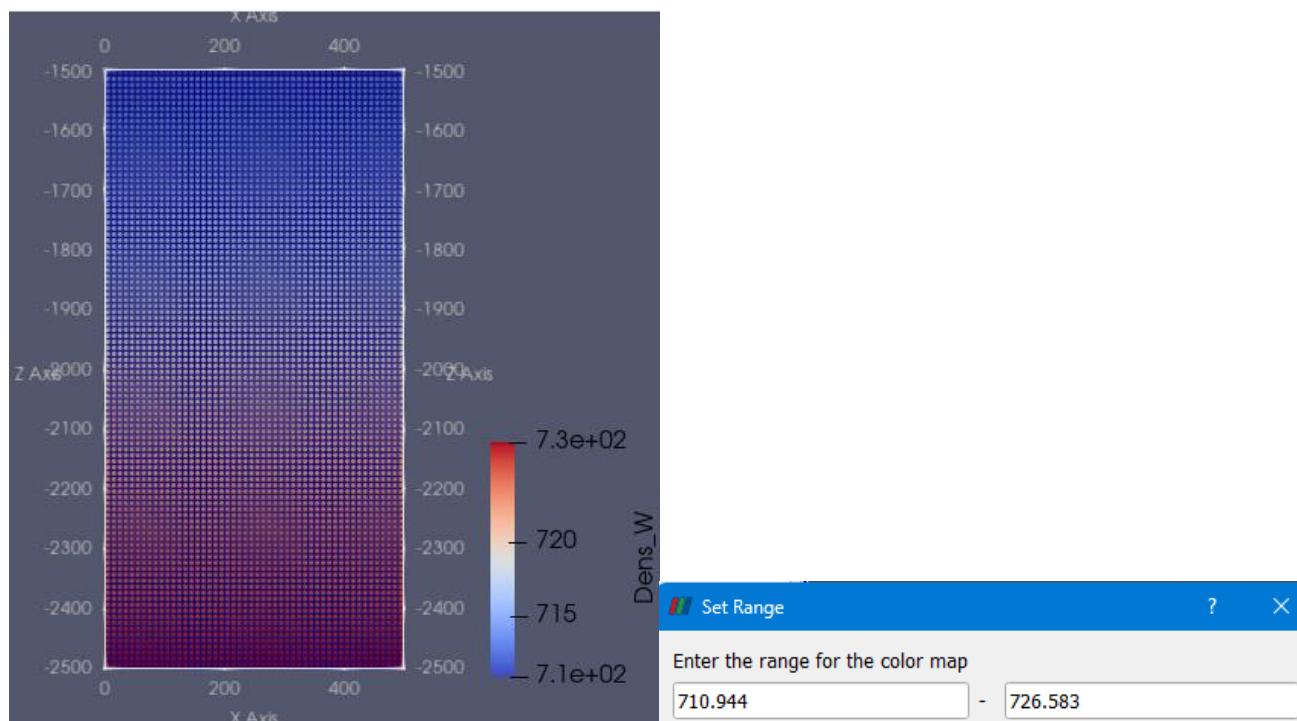
La diferencia de presión entre el tope y el fondo del yacimiento es de aproximadamente 70 [bar]. Esto nos da un gradiente de:

$$\frac{\Delta P}{\Delta z} = \frac{70 \times 10^5}{1000} = 7000 \left[\frac{Pa}{m} \right]$$

La densidad de la roca saturada con agua que ejercería un gradiente de presión equivalente sería de:

$$\rho = \frac{\Delta P}{gH} = \frac{7,000,000}{9810} = 713.55 \left[\frac{kg}{m^3} \right]$$

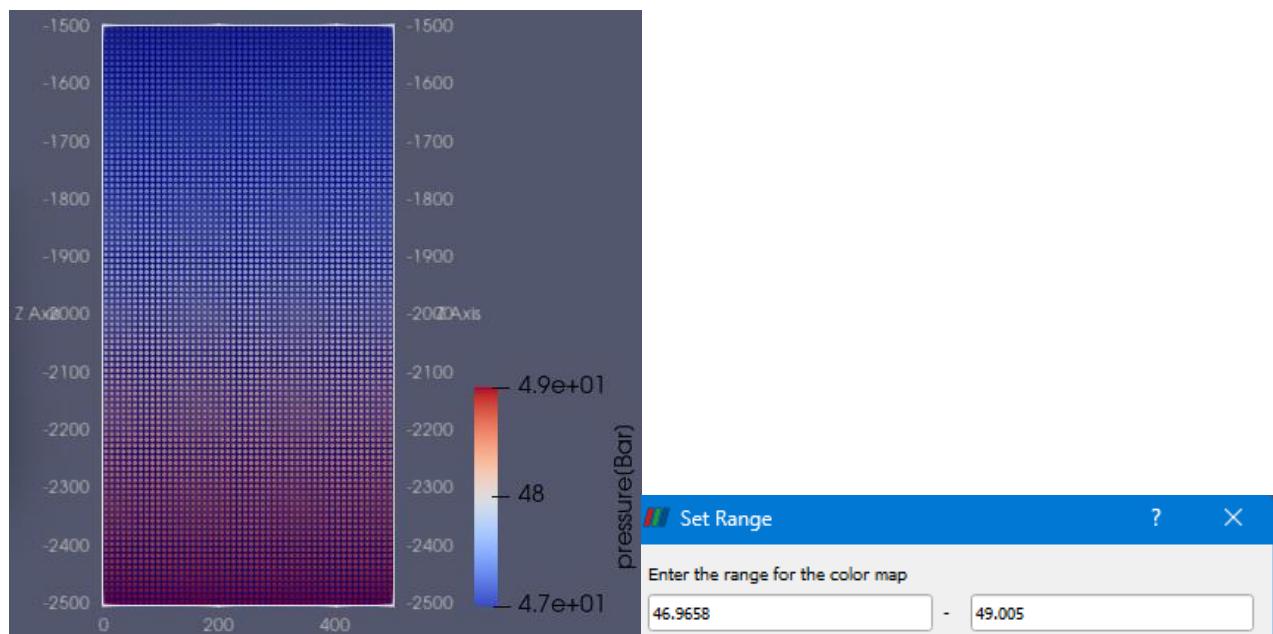
Esta densidad es prácticamente la del agua. Con 1 % de porosidad eso implicaría que la matriz rocosa tendría una densidad parecida a la del agua, lo cual no corresponde a una roca geológica real (por ejemplo: granito, basalto, lutitas suelen estar en ~2400–3000 kg/m³). Esto se debe a que, aunque DARTS si toma en cuenta el término gravitacional en la simulación, lo que en verdad resuelve para obtener el gradiente de presión es la ecuación de Darcy. Si vemos el mapa de densidad del agua resultante de la simulación, vemos que la densidad calculada previamente está dentro del rango obtenido en la simulación:



Ahora corremos una simulación similar para conocer el gradiente de presión que ejercería una columna de 1000 [m] de vapor. Establecemos una presión de 50 [bar] para asegurarnos que solo existe vapor en el yacimiento:

```
# initialization with constant pressure and temperature
self.input_distribution = {"pressure": 50., # [bar]
                           "temperature": 300. + 273.15 # [c]----[k]}
```

Si vemos el mapa de presión resultante



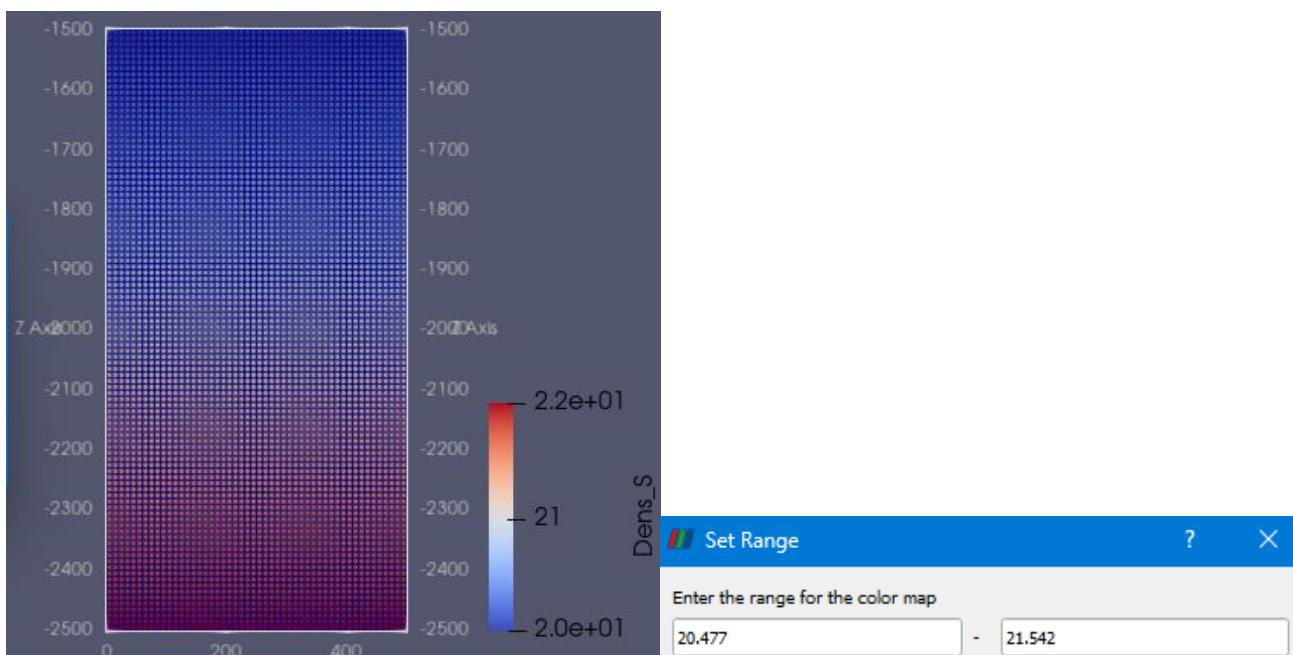
La diferencia de presión entre el tope y el fondo del yacimiento es de aproximadamente 2 [bar]. Esto nos da un gradiente de:

$$\frac{\Delta P}{\Delta z} = \frac{2 \times 10^5}{1000} = 200 \left[\frac{Pa}{m} \right]$$

La densidad de la roca saturada con agua que ejercería un gradiente de presión equivalente sería de:

$$\rho = \frac{\Delta P}{gH} = \frac{200,000}{9810} = 20.387 \left[\frac{kg}{m^3} \right]$$

Si vemos la densidad del vapor resultante de la simulación:



Podemos usar esta información para establecer las condiciones iniciales para el caso específico en el que se tenga un casquete de vapor. Si queremos modelar un yacimiento con un espesor 300 [m] de vapor y 700 [m] de agua, entonces dicha columna de agua resultara en un gradiente de presión equivalente a:

$$\Delta P = \frac{\Delta P}{\Delta z} * H = (.070)(700) = 49 [bar]$$

Y la columna de vapor tendrá:

$$\Delta P = \frac{\Delta P}{\Delta z} * H = (.002)(300) = 0.6 [bar]$$

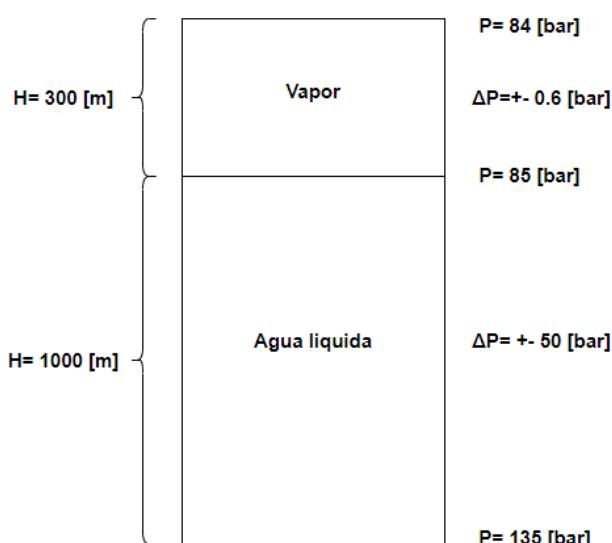
Se utilizará la profundidad para determinar las condiciones iniciales de presión y temperatura, por lo que debemos definir la profundidad a la que se encuentra el tope del yacimiento:

Sabemos que a 300 [°C], la presión de saturación es de alrededor de 86 [bar], por lo que la presión en el límite de ambas fases debería ser cercana a este valor. Se establecen las siguientes condiciones iniciales:

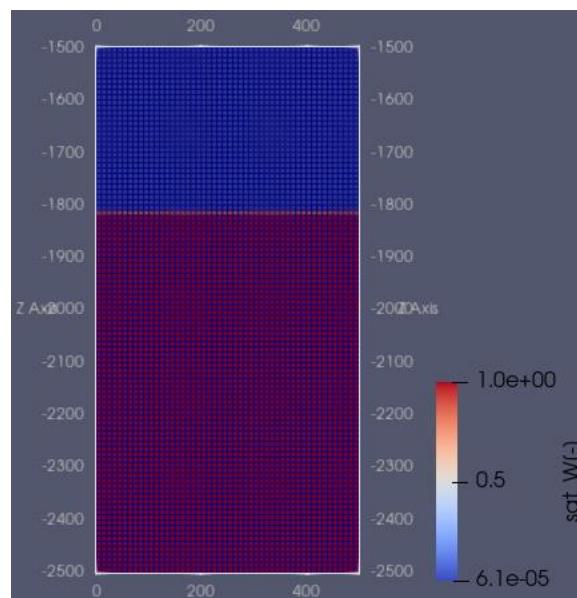
```
##### Caso #2 #####
# CASQUETE DE VAPOR

input_depth = [1500, 1800, 2500]
temp= 300. + 273.15
input_distribution = {
    "pressure": [84, 85, 135],      #
    "temperature": [temp, temp, temp] #
}

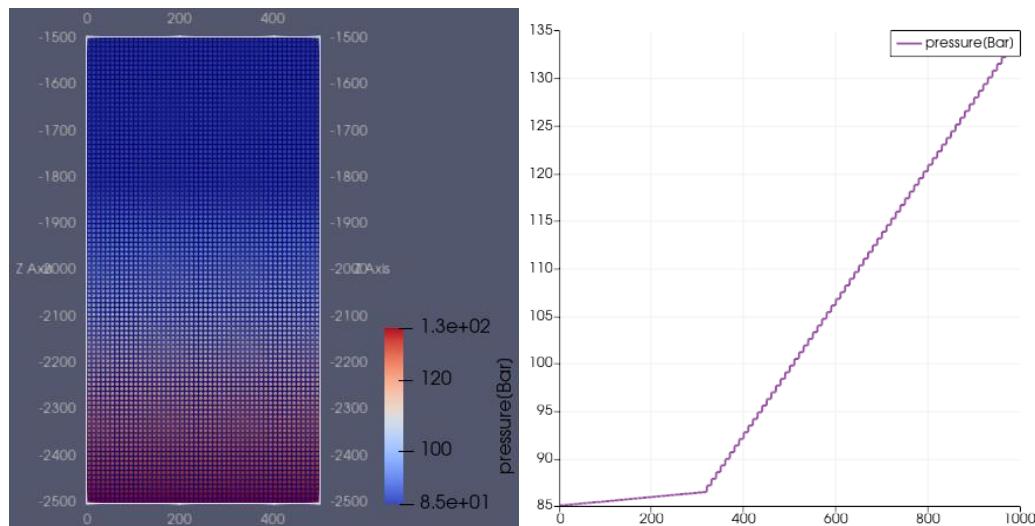
self.physics.set_initial_conditions_from_depth_table(mesh=self.reservoir.mesh,
                                                     input_distribution=input_distribution, input_depth=input_depth)
```



Se establece un valor de presión ligeramente menor a la presión de saturación en el límite de fases, solo para estar seguros de que existe vapor en por lo menos los primeros 300 [m]. Volvemos a correr la simulación en estado natural. Si vemos el mapa de saturación de agua y vapor, vemos que se tiene 100% vapor cerca de los primeros 300 [m] y que, además, el perfil prácticamente no cambia en todo el tiempo de simulación, lo que indica que el estado natural del yacimiento está bien definido:



El mapa de presión resultante y la presión vs profundidad se muestran a continuación:



Una vez definido el estado natural del yacimiento, podemos simular un caso de explotación. Se define un solo pozo productor en la parte más profunda, el cual operara a presión de fondo constante:

```
def set_wells(self):
    # add well
    self.reservoir.add_well("PRD", wellbore_diameter=0.3048)
    self.reservoir.add_perforation("PRD", cell_index=(25, 1, 100), skin=0.0,
                                    well_radius=0.1524, multi_segment=True, well_index=100, well_indexD=0)
```

```

def set_well_controls(self):
    P_prod=70 # [bar]

    print('\n-----Producer :')
    print('PRD pressure [bar] =',P_prod)
    print('-----')

    for i, w in enumerate(self.reservoir.wells):
        self.physics.set_well_controls(wctrl=w.control,
                                       control_type=well_control_iface.BHP,
                                       is_inj=False,
                                       target=P_prod
                                       )

```

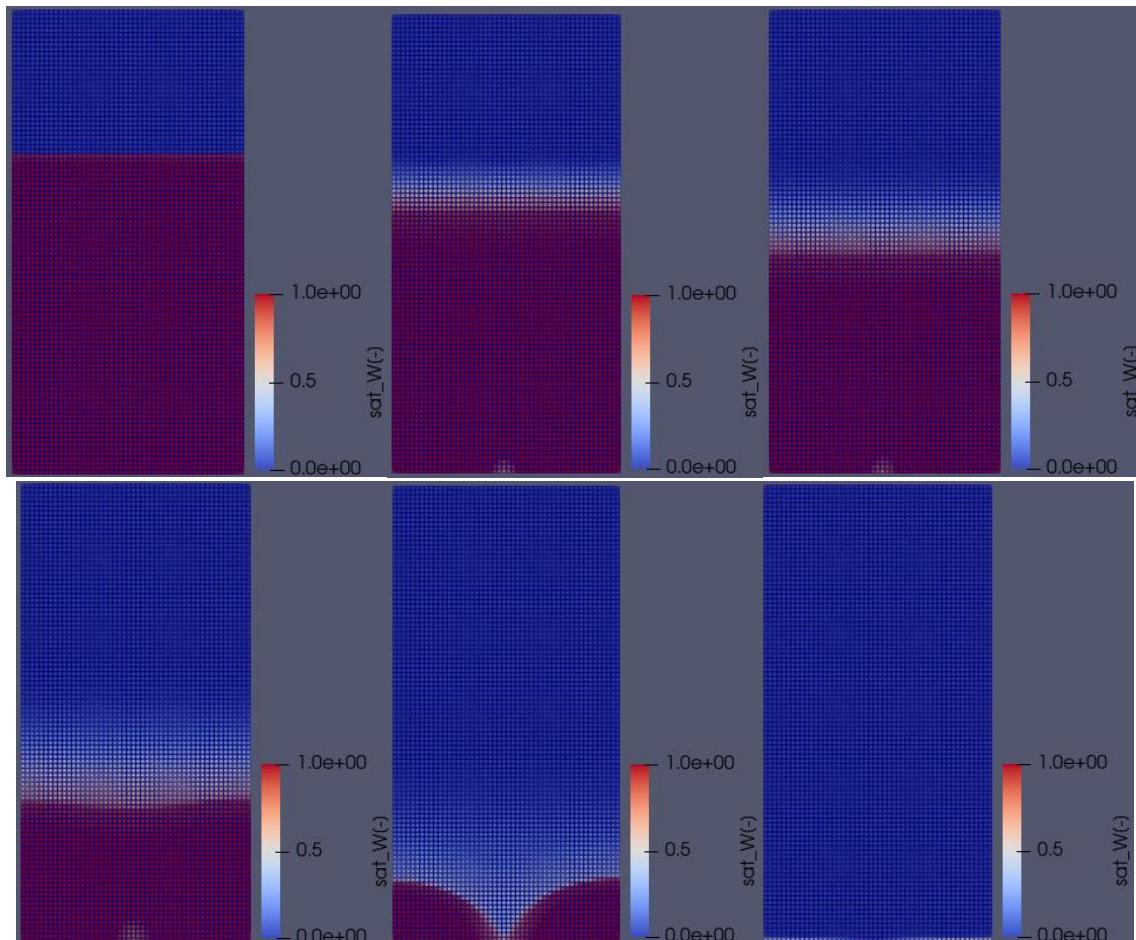
Corremos la simulación usando un paso de tiempo exponencial:

```

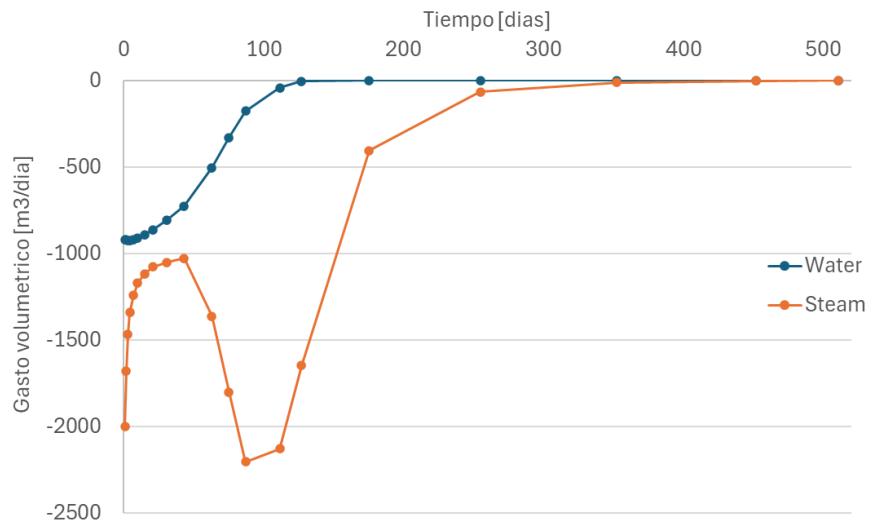
Paso_de_tiempo= 1
tiempo_inicial=0
tiempo_total=0
# # Correr simulaciones
for t in range(9): #
    m.run(Paso_de_tiempo)
    m.output.output_to_vtk(output_directory=output_dir, output_properties=prop_list)
    tiempo_total=tiempo_total + Paso_de_tiempo
    Paso_de_tiempo=Paso_de_tiempo*2

```

Si observamos la saturación de agua para los tiempos 0, 7, 15, 31, 63 y 127 días:



Si ploteamos los gastos volumétricos del pozo productor para cada fase (se plotean únicamente los valores mayores a 1 dia):



Al principio, se tiene el máximo gasto ya que existe el máximo gradiente de presión. Conforme el gradiente de presión viaja, y debido a que no existe un pozo inyector que mantenga la presión, la saturación de vapor aumenta ya que el yacimiento se depresiona lentamente. Conforme el gradiente de presión cae, el gasto también cae. Cerca de los 50 días, el nivel del casquete de vapor alcanza el pozo productor por lo que el gasto de vapor aumenta drásticamente. Cerca de los 100 días, el yacimiento se ha depresionado tanto que no existe agua líquida en el yacimiento. A partir de este punto, el gasto de agua es prácticamente cero y el gradiente de presión cae rápidamente por el que el gasto de vapor también caerá rápidamente (aunque aún se produce vapor durante poco más de 200 días adicionales).

Ejemplo 12

En este ejemplo, simularemos una fuente de calor en la parte inferior del yacimiento. Se establece un yacimiento en 3 dimensiones con solo dos capas en la dirección “z”:

```
self.nx, self.ny, self.nz = 30, 30, 2
self.nb = self.nx * self.ny * self.nz # número total de bloques

Long_x=1000. # [m]
Long_y=1000. # [m]
Long_z= 100. # [m]
```

Se establece una permeabilidad de cero para la capa inferior

```
# Crear un arreglo vacío
perm_x = np.zeros(self.nb)
perm_y = np.zeros(self.nb)
perm_z = np.zeros(self.nb)

# Asignar los valores según las capas

perm_x[0:self.nx*self.ny] = 40          # [mD]
perm_x[self.nx*self.ny:2*self.nx*self.ny] = 0 # [mD]

perm_y[0:self.nx*self.ny] = 40          # [mD]
perm_y[self.nx*self.ny:2*self.nx*self.ny] = 0 # [mD]

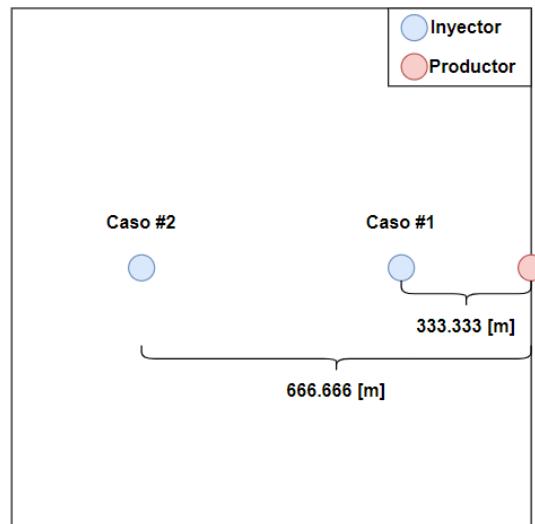
perm_z[0:self.nx*self.ny] = 40          # [mD]
perm_z[self.nx*self.ny:2*self.nx*self.ny] = 0 # [mD]

self.reservoir = StructReservoir(self.timer, nx=self.nx, ny=self.ny, nz=self.nz, dx=self.dx, dy=self.dy, dz=self.dz,
                                 permx=perm_x, perm_y=perm_y, permz=perm_z, poro=.01,
                                 hcap=2470., rcond= 800. )
```

Se aplica una frontera abierta en la parte inferior del yacimiento

```
self.reservoir.boundary_volumes['xy_plus'] = 1e8
```

Se plantean dos casos:



Para el primer caso, la distancia entre el pozo inyector y productor será de 10 celdas (333.333 metros), mientras que en el segundo caso el pozo inyector estará más alejado (doble de distancia). La posición del pozo productor es la misma:

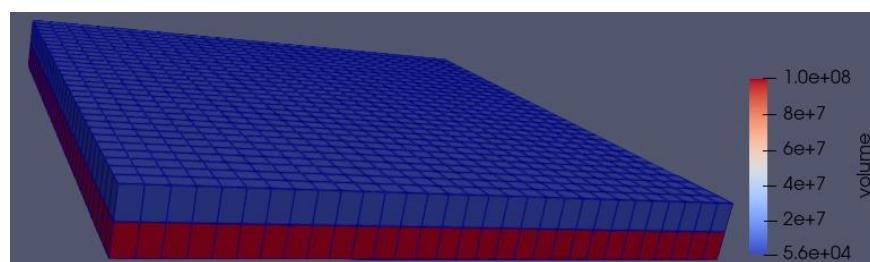
```
# add well
self.reservoir.add_well("INJ", wellbore_diameter=0.3048)

if Select_case.case == '1':
    self.reservoir.add_perforation("INJ", cell_index=(20, 15, 1), skin=0.0,
                                    well_radius=0.1524, multi_segment=True,well_indexD=0)

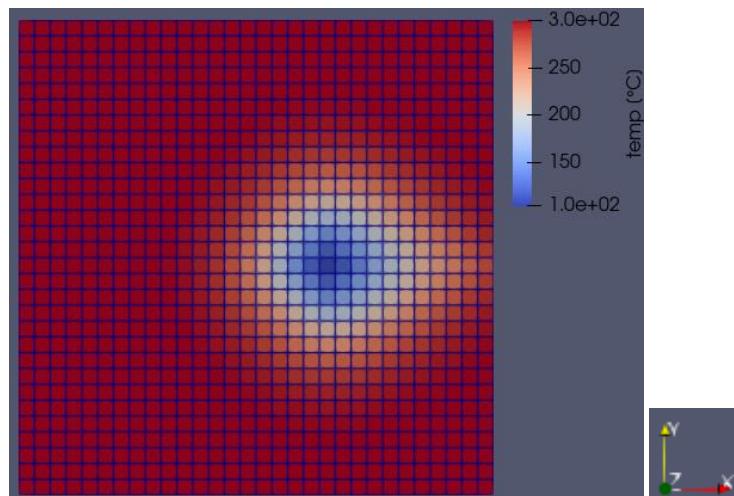
elif Select_case.case == '2':
    self.reservoir.add_perforation("INJ", cell_index=(10, 15, 1), skin=0.0,
                                    well_radius=0.1524, multi_segment=True,well_indexD=0)

# add well
self.reservoir.add_well("PRD", wellbore_diameter=0.3048)
self.reservoir.add_perforation("PRD", cell_index=(30, 15, 1), skin=0.0,
                            well_radius=0.1524, multi_segment=True,well_indexD=0)
```

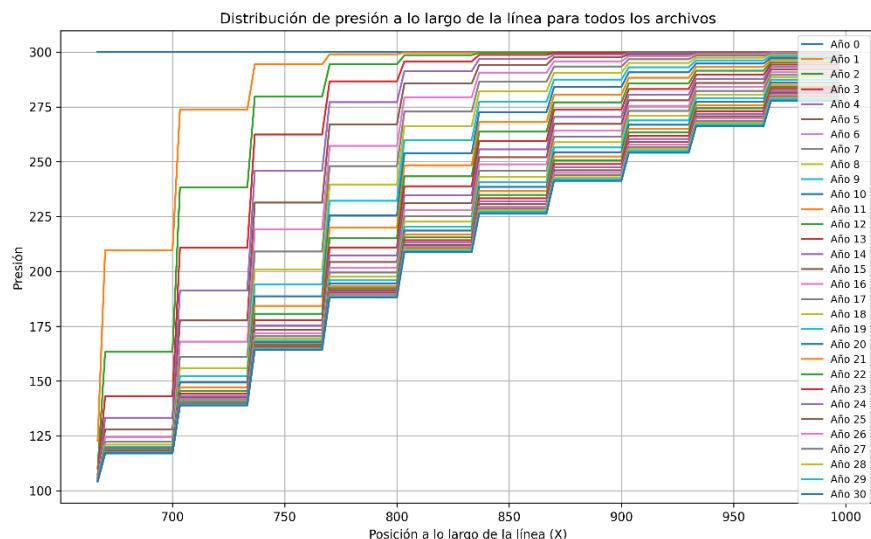
Se muestra los resultados para el primer caso. Si observamos el volumen de las celdas, vemos que la capa inferior funge como una frontera abierta.



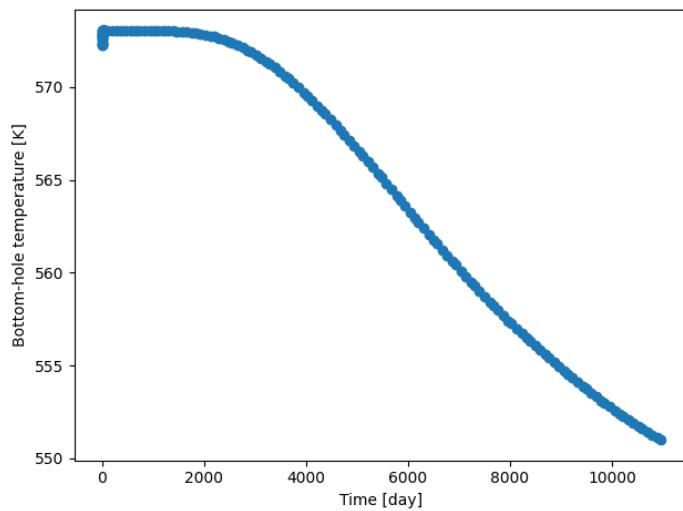
Se muestra el perfil de temperatura después de 30 años:



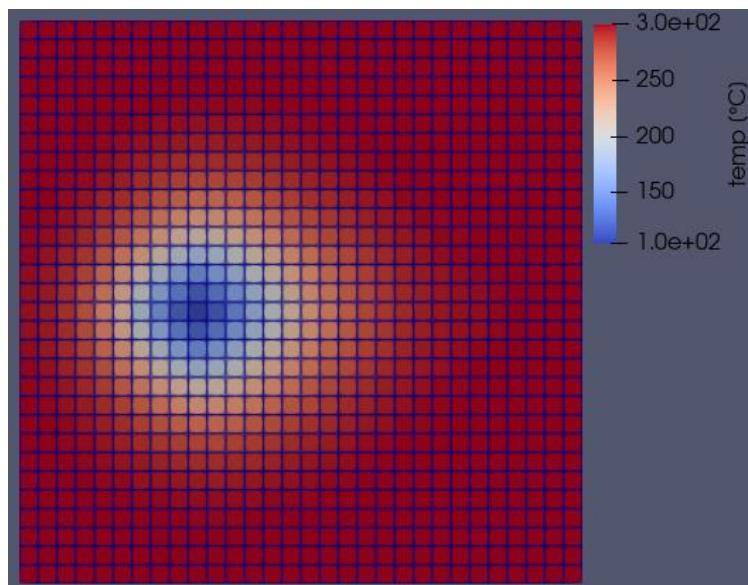
Se obtiene el perfil de temperatura sobre la línea recta que une a los dos pozos:

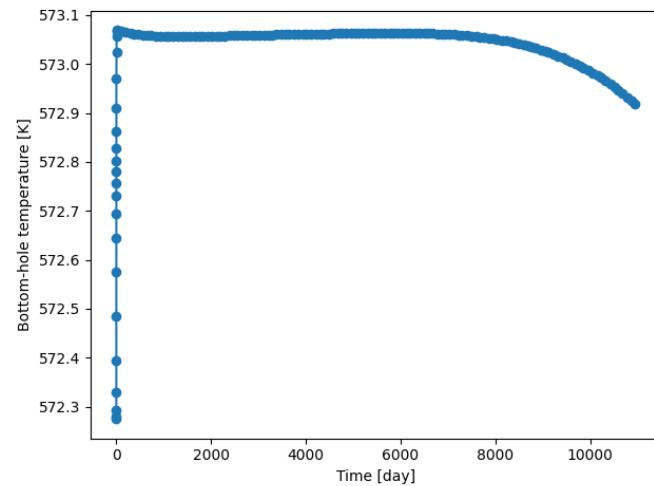
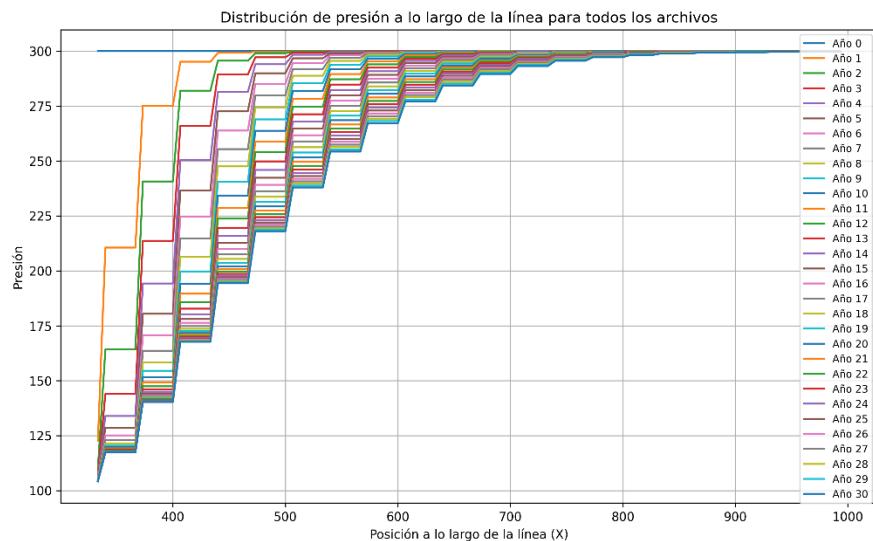


Se puede observar que la distancia entre ambos pozos no es lo suficientemente grande como para permitir que el agua fría inyectada por el pozo inyector (extremo izquierdo de la gráfica) se caliente lo suficiente mientras viaja hacia el pozo productor (extremo derecho). Si observamos la gráfica de temperatura del pozo productor:



Vemos que la temperatura del pozo productor (y por ende, la temperatura del fluido extraído) ha caído alrededor de 22[°C] en los 30 años de producción, algo indeseable en la explotación de un campo geotérmico. Repetimos el ejemplo para el caso 2 (pozo inyector a una distancia mayor):

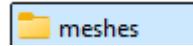




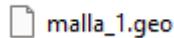
Se recomienda repetir el ejemplo, pero usando un valor diferente de R_cond (por ejemplo, usando 400)

Ejemplo 13

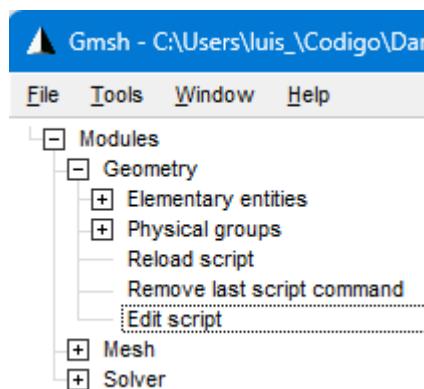
En este ejemplo se utilizará una malla no estructurada. Los archivos de la malla se encuentran en la carpeta:



En ella se ubica el archivo .geo



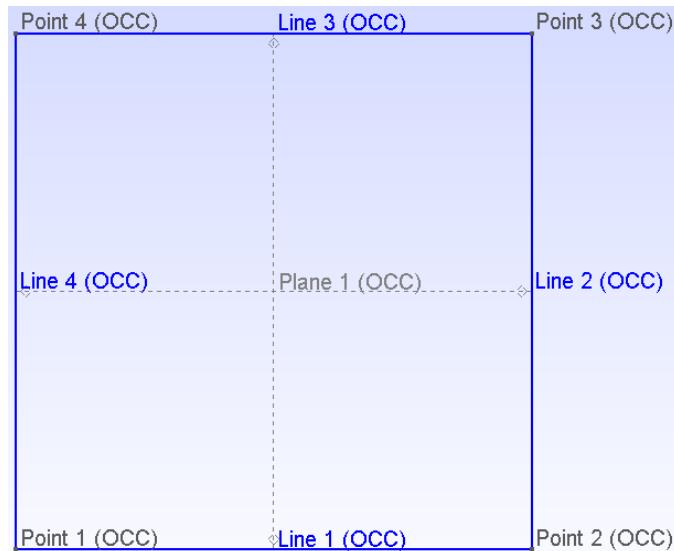
Abrir el archivo en Gmsh. Del lado izquierdo, seleccionar “Edit Script” para ver los comandos con los que se creó la malla:



- height_res: espesor del yacimiento.
- rsv_layers: Indica cuántas capas tendrá el yacimiento.
- lc: Tamaño característico de las celdas alrededor de los puntos creados.

```
height_res= 100;
rsv_layers=3;
lc=50;
```

<pre>// Extra points for boundary of domain: Point(1) = { 0.00000, 0.00000, 0.00000, lc}; Point(2) = {1000.00000, 0.00000, 0.00000, lc}; Point(3) = {1000.00000, 1000.00000, 0.00000, lc}; Point(4) = { 0.00000, 1000.00000, 0.00000, lc}; // Extra lines for boundary of domain: Line(1) = {1, 2}; Line(2) = {2, 3}; Line(3) = {3, 4}; Line(4) = {4, 1}; // Create line loop for boundary surface: Curve Loop(1) = {1, 2, 3, 4}; Plane Surface(1) = {1};</pre>	<p>Se crean los puntos con los cuales se crearán las líneas que definen los límites del dominio. A su vez, estas líneas sirven para formar un loop, y este loop se usa para crear la superficie</p>
---	---



```
// Extrude surface with embedded features

// Reservoir
sr[] = Extrude {0, 0, height_res}{ Surface {1}; Layers{rsv_layers}; Recombine;};
```

- Surface {1}: Es la superficie base que se está extruyendo (el plano base del reservorio).
- Layers {rsv_layers}: Esto divide el volumen extruido en capas horizontales (como estratos del reservorio).
- Recombine: Intenta convertir los elementos tetraédricos (por defecto) en hexaédricos o prismáticos, que son preferidos en muchos métodos numéricos.
- sr[] : Gmsh guarda el resultado de la extrusión en un array sr[]. Este array contiene:

sr[0] La nueva superficie extruida superior

sr[1..n] Las superficies laterales creadas por la extrusión

sr[last] El volumen 3D creado

```
// Horizontal surfaces
Physical Surface(1) = {sr[0]}; // top
Physical Surface(2) = {sr[1]}; // bottom
Physical Surface(3) = {sr[2]}; // Y-
Physical Surface(4) = {sr[3]}; // X+
Physical Surface(5) = {sr[4]}; // Y+
Physical Surface(6) = {sr[5]}; // X-
```

Asignación de superficies físicas:

asignan **nombres (etiquetas físicas)** a las superficies del volumen extruido. Esto es útil para:

- Aplicar **condiciones de frontera** distintas en cada cara (por ejemplo, flujo de entrada, no flujo, presión, etc.),
- Exportar la malla con etiquetas que el simulador pueda entender.

***Nota: Los physical tags solo se verán una vez que se extruye la malla

Se etiqueta el volumen con ID 1 como un volumen físico llamado matrix, con identificador 9991, para poder referenciarlo luego en la simulación.

```
//Reservoir
Physical Volume("matrix", 9991) = {1};
```

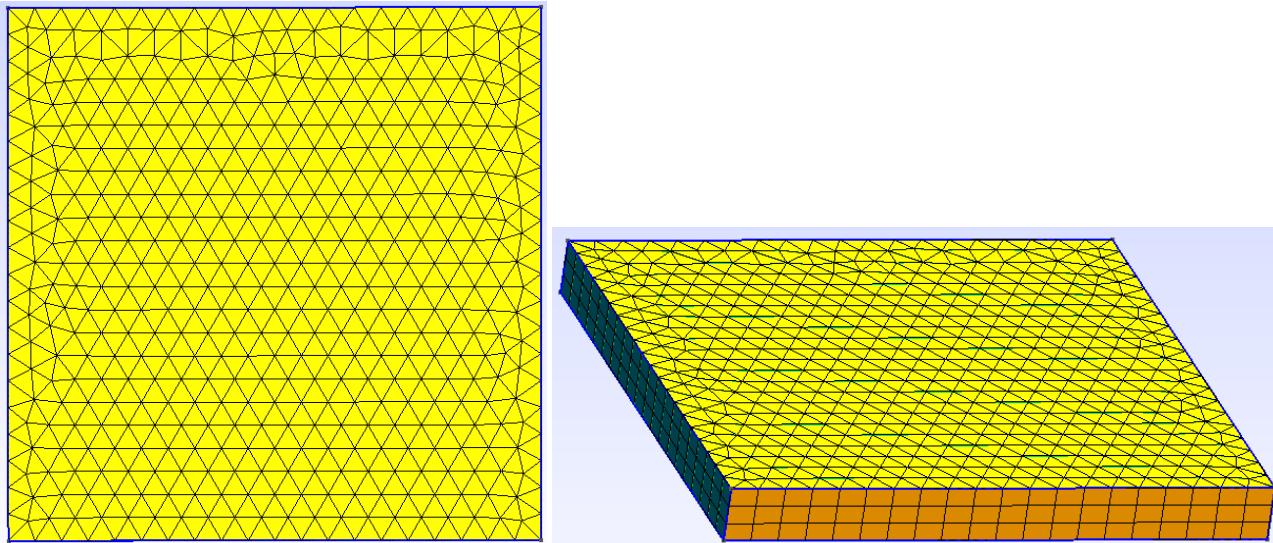
Esta línea **genera la malla 3D** (volúmenes) de la geometría definida.

```
Mesh 3; // Generate 3D mesh
```

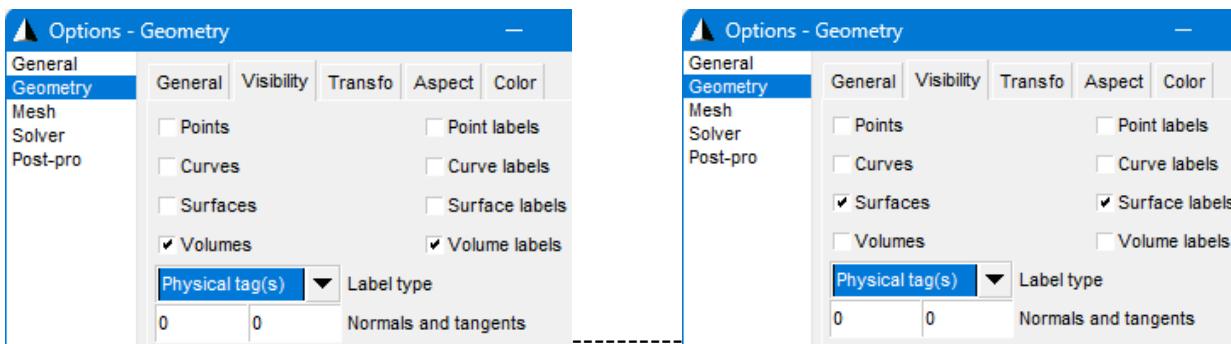
Elimina entidades duplicadas o inconsistentes que se hayan generado durante operaciones geométricas. Solo afecta la **malla**, no la geometría CAD original.

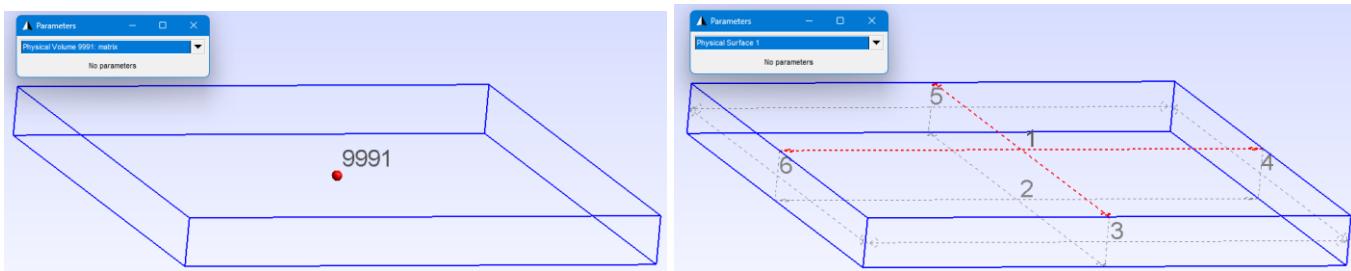
```
Coherence Mesh; // Remove duplicate entities
```

La malla resultante se verá de esta forma:

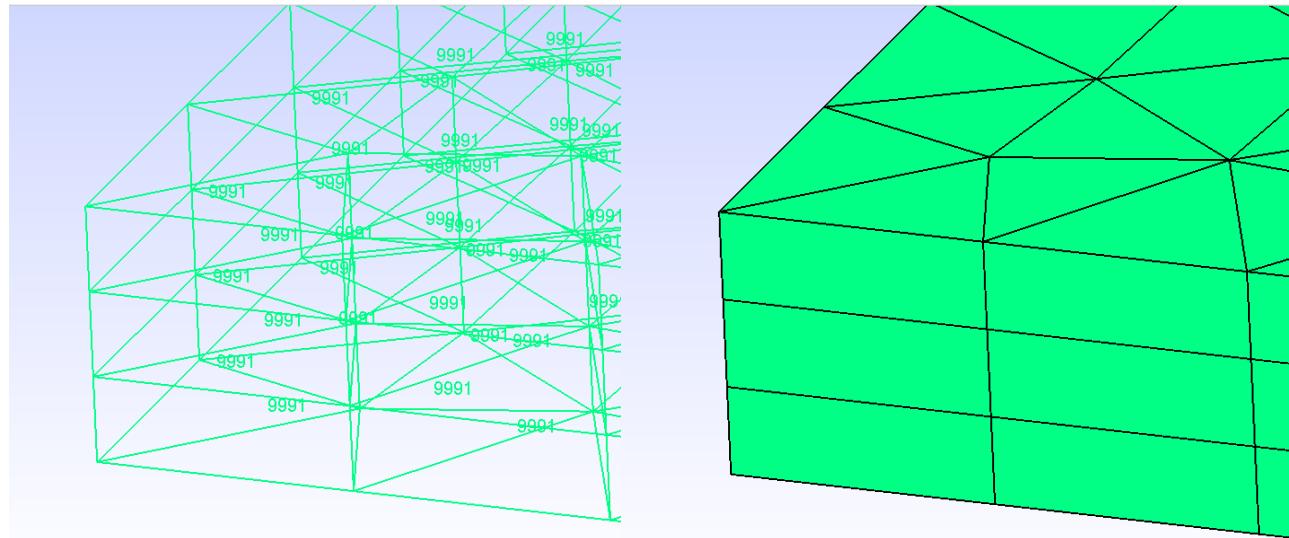
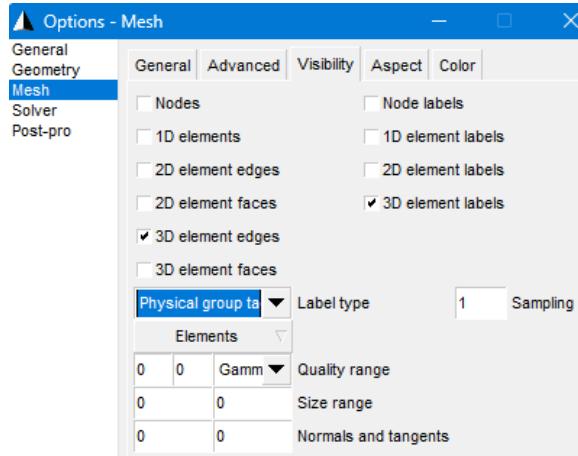


Para ver los physical tags que se asignaron en gmsh:

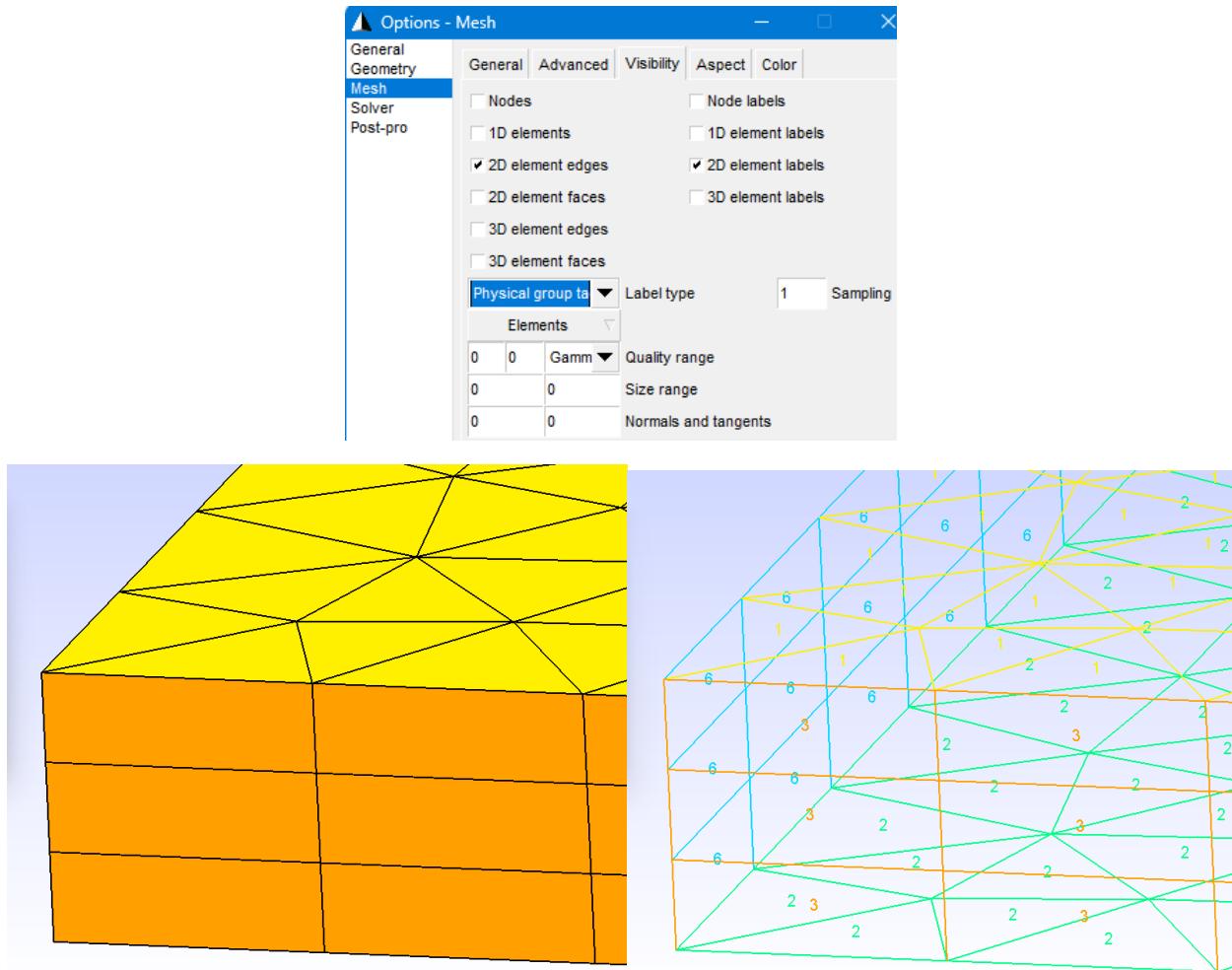




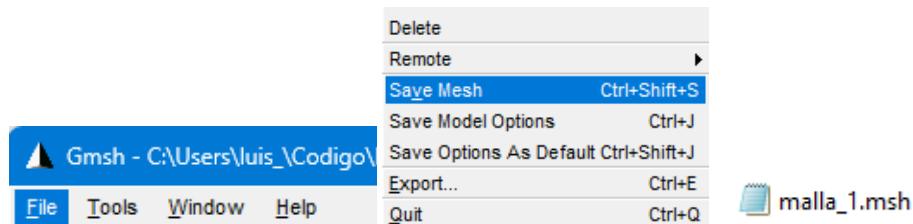
Todas las celdas (elementos 3D) dentro del Volumen tendrán la misma etiqueta que el volumen que las contiene



Lo mismo aplica para las celdas (elementos 2D) que componen las superficies.



Una vez terminada la creación de la malla, se accede a la opción “save mesh” para poder crear el archivo .msh



Una vez creada la malla, el siguiente paso es cargarla al modelo creado. Para esto, se utiliza la clase `UnstructReservoir`:

```
from darts.reservoirs.unstruct_reservoir import UnstructReservoir
```

Se define el nombre del archivo de malla:

```

fname = 'malla_1.msh'
mesh_file = os.path.join('meshes', fname)
print(mesh_file)

```

Y se utiliza la clase UnstructReservoir para cargar la malla (se dan valores cualquiera de permeabilidad, serán sobre escritos más adelante):

```

self.reservoir = UnstructReservoir(timer=self.timer, mesh_file=mesh_file,
                                     permx=permx, permy=permy, permz=permz,
                                     poro=poro,
                                     rcond=rcond,
                                     hcap=hcap)

```

El siguiente paso es indicarle a DARTS el tipo de elemento ('matrix', 'fracture o 'boundary') de cada uno de los elementos de Gmsh. Para esto, se utiliza el numero usado ('physical tag') para identificarlas:

```

self.reservoir.physical_tags['matrix'] = [9991]
self.reservoir.physical_tags['boundary'] = [2, 1, 3, 4, 5, 6]

```

Si DARTS detecta que hay un elemento de la malla que no fue asignado a ninguno de los elementos básicos de DARTS ('matrix', 'fracture o 'boundary'), entonces marcará error. Por ejemplo, si la frontera izquierda (Physical tag=6) no se define como ningún tipo de elemento básico:

```

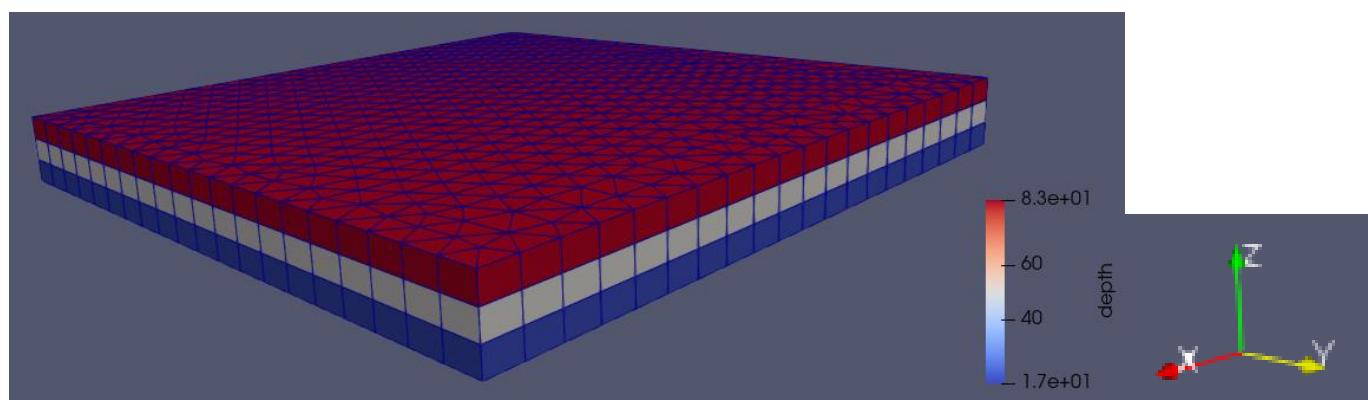
self.reservoir.physical_tags['matrix'] = [9991]
self.reservoir.physical_tags['boundary'] = [2, 1, 3, 4, 5]

```

Marcará error:

```
ValueError: ('Unsupported physical tag found', np.int64(6))
```

Los "physical tag" pueden ser usados para controlar regiones o fronteras específicas del modelo. En este ejemplo solo se muestra como definirlas (los physical tags serán usados en otro ejemplo mas adelante). El siguiente paso es asignar la ubicación de los pozos. El pozo inyector se ubicará en la esquina inferior izquierda, mientras el pozo productor en la esquina opuesta. Algo importante a tomar en cuenta es que, si abrimos el archivo msh.vts después de correr una simulación, vemos que a diferencia de cuando se usa una malla estructurada, la profundidad se asigna al revés. Es decir, la profundidad aumenta en el sentido de "z"



Esto se debe tomar en cuenta al momento de hacer el modelo y asignar la ubicación de los pozos. Una solución sencilla es crear el modelo considerando “Z” al revés, por lo que las coordenadas de los pozos estarán invertidas. Por otro lado, debido a que se trata de una malla no estructurada, asignar los pozos al índice de celda deseado no es tan fácil, por lo que se utiliza una función para calcular dicho índice a partir de las coordenadas de los pozos.

```
def calc_well_loc(self):
    self.inj_well_coords = [[0, 1000, 0]]
    self.prod_well_coords = [[1000, 0, 0]]
```

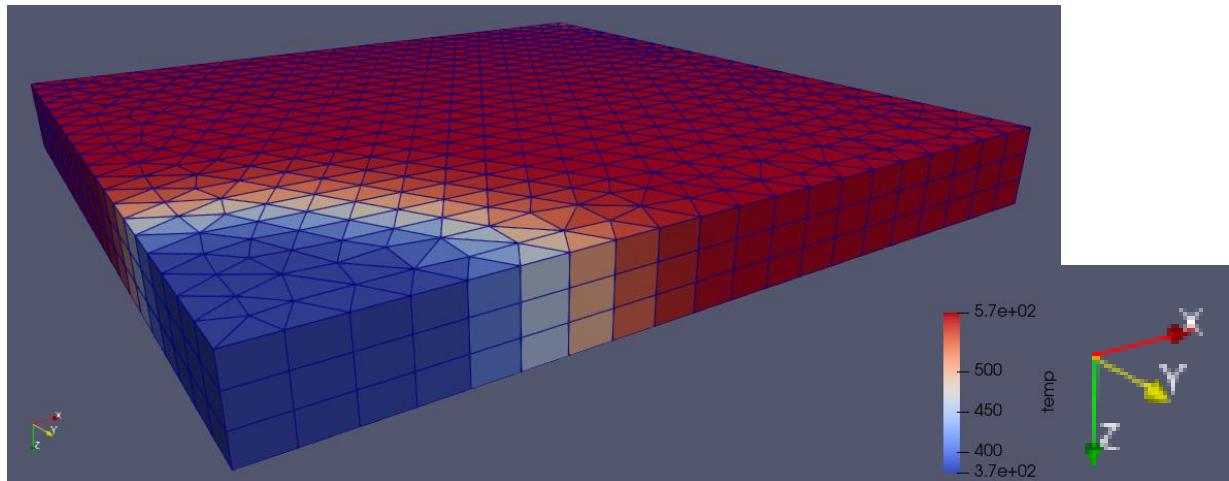
Además, se tiene que definir si el pozo se asigna a la celda de matriz o fractura más cercana. Alternativamente, se puede escoger que se asigne a la celda más cercana, independientemente de si es una celda de matriz o de fractura.

```
self.bound_cond = 'wells_in_nearest_cell'
# self.bound_cond == 'wells_in_frac'
# self.bound_cond == 'wells_in_mat'
```

Esta función devolverá la variable `self.well_perf_loc`, la cual contiene los índices de celda deseados. Esta función se llama al comienzo de la función `set_wells`:

```
def set_wells(self):
    self.calc_well_loc()
```

Si corremos la simulación y visualizamos el campo de temperatura dentro del archivo .vtk resultante:



Notese que se tiene que invertir el eje “z” para poder ver la orientación correcta del modelo, debido al problema de profundidades mencionado anteriormente.

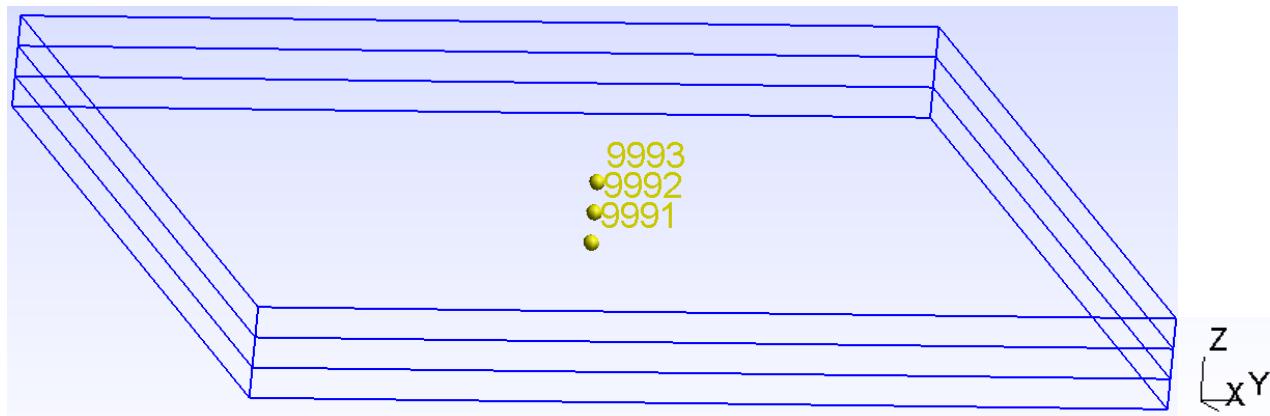
Ejemplo 14

En este ejemplo se mostrará como asignar condiciones iniciales y propiedades específicas de cada capa, usando una malla no estructurada. Para crear 3 capas o volúmenes independientes en Gmsh, se tiene que hacer 3 extrusiones seguidas, utilizando la superficie superior del volumen recién extruido (recordemos que $\text{sr}[0]$ = cara superior del volumen recién creado) :

```
// --- Extrusión capa 1 ---
sr1[] = Extrude {0, 0, h_layer}{ Surface {1}; Layers{1}; Recombine;};
Physical Volume("matrix_layer1", 9991) = {1};

// --- Extrusión capa 2 ---
sr2[] = Extrude {0,0,h_layer} { Surface{sr1[0]}; Layers{1}; Recombine; };
Physical Volume("matrix_layer2", 9992) = {2};

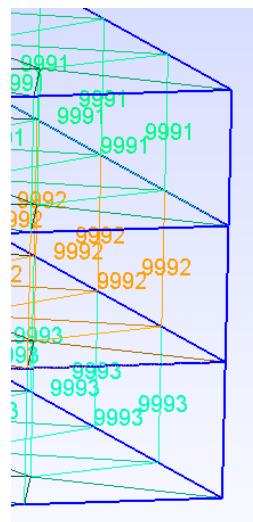
// --- Extrusión capa 3 ---
sr3[] = Extrude {0,0,h_layer} { Surface{sr2[0]}; Layers{1}; Recombine; };
Physical Volume("matrix_layer3", 9993) = {3};
```



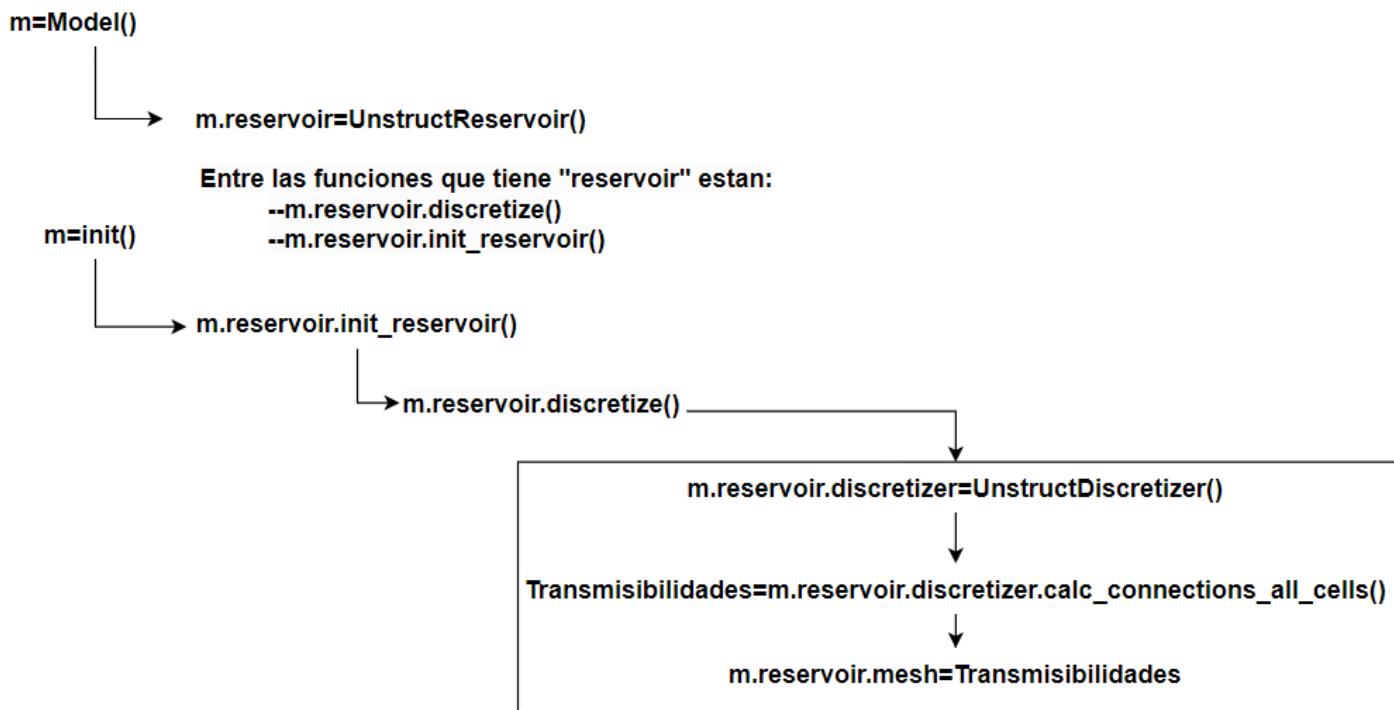
Debemos recordar que, debido al problema de profundidades explicados en el ejemplo anterior, la orientación correcta de la modelo seria “volteando” el eje “z”. De esta forma, la capa clasificada como “9991” seria la capa superior (o más somera):



De nuevo, cada celda tendrá el mismo identificador que el volumen que la contiene:



Para mallas no estructuradas, se debe modificar la función o clase que guarda las permeabilidades. Al igual que el caso anterior, se copiará esta función en lugar de modificar la función original. Se agregará la parte para calcular las profundidades, las condiciones iniciales y las propiedades para cada capa del modelo. El diagrama de flujo resumido para la parte de yacimiento se muestra a continuación:



Las permeabilidades y profundidades de cada celda se ubican dentro de `m.reservoir.discretizer.mat_cell_info_dict`:

```

<code>
  m = <model.Model object at 0x000001C8370A81>
  reservoir = <darts.reservoirs.unstruct_reservoir>
  discretizer = <darts.reservoirs.mesh.unstructured>
  mat_cell_info_dict = {0: <darts.reservoirs.mesh.GeometryModule>
    0 = <darts.reservoirs.mesh.GeometryModule>
      permeability = array([10., 10., 10.])
</code>
  m = <model.Model object at 0x000001C8370A81>
  reservoir = <darts.reservoirs.unstruct_reservoir>
  discretizer = <darts.reservoirs.mesh.unstructured>
  mat_cell_info_dict = {0: <darts.reservoirs.mesh.GeometryModule>
    0 = <darts.reservoirs.mesh.GeometryModule>
      depth = np.float64(1516.666666666667)
</code>
  
```

Estos campos son los que se deben modificar, ya que se utilizan para calcular las transmisibilidades (que a su vez sirven para generar el atributo “mesh”). Como “discretize” es la función que crea “`m.reservoir.discretizer`”, debemos de modificarla. Adicionalmente se modifica la función “`set_layer_properties`”. Ambas son copiadas y utilizadas en lugar de las originales:

```
from darts.reservoirs.unstruct_reservoir import UnstructReservoir
```

```
# ⚡ Parcheamos el método de Geothermal
Geothermal.set_initial_conditions_from_depth_table = (
    patched_set_initial_conditions_from_depth_table
)

# ⚡ Parcheamos el método
UnstructReservoir.discretize = (
    patched_UnstructReservoir_discretize
)

# ⚡ Parcheamos el método
UnstructReservoir.set_layer_properties = (
    patched_set_layer_properties
)
```

Como la función “discretize” hace uso a su vez de la función UnstructDiscretizer, también hay que importarla:

```
from darts.reservoirs.mesh.unstruct_discretizer import UnstructDiscretizer
```

Para agilizar el manejo de los datos de entrada, se crea una pequeña clase para asignar las propiedades:

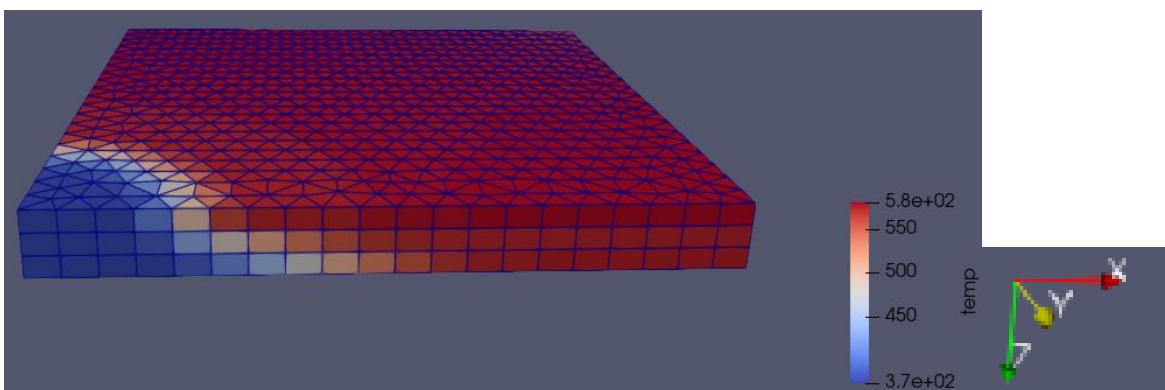
```
from dataclasses import dataclass

@dataclass
class Layer_prop:
    poro: float
    perm: float
    anisotropy: list = None
    hcap: float = 2200
    rcond: float = 181.44
```

Se crea un diccionario que contenga los ID de las capas o regiones que se establecieron en el archivo de Gmsh. En este caso, la única diferencia entre las capas será la permeabilidad:

```
layer_props = {9991: Layer_prop( poro=0.01, perm=10., anisotropy=[1, 1, 1]), # capa de hasta abajo
               9992: Layer_prop(poro=0.01, perm=20., anisotropy=[1, 1, 1]),
               9993: Layer_prop(poro=0.01, perm=40., anisotropy=[1, 1, 1])}
```

Si ploteamos la temperatura, vemos que efectivamente el fluido se canaliza por la capa con mayor permeabilidad (capa “9993”, la más profunda)



Ejemplo 15

En este ejemplo, crearemos un yacimiento con forma irregular

```

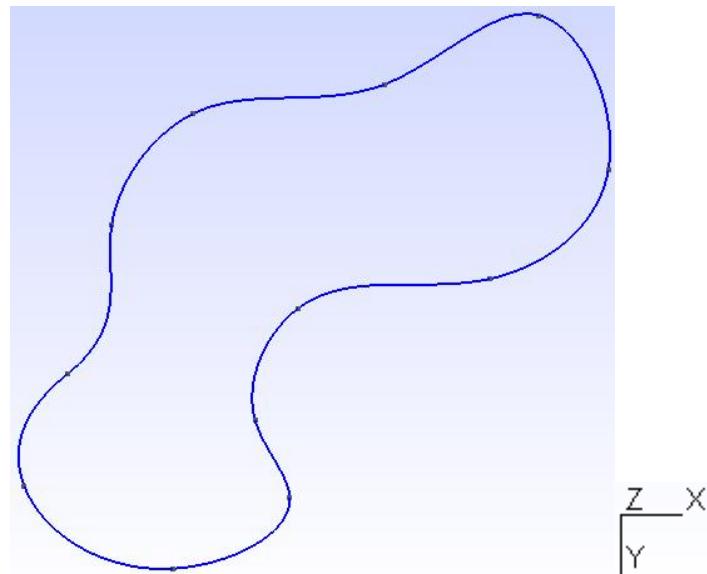
height_res= 100;
rsv_layers=3;
lc=80;
// --- Puntos frontera ---
Point(1) = {200.0, 700.0, 0.0, lc};
Point(2) = {150.0, 850.0, 0.0, lc};
Point(3) = {350.0, 950.0, 0.0, lc};
Point(4) = {500.0, 850.0, 0.0, lc};
Point(5) = {450.0, 750.0, 0.0, lc};
Point(6) = {500.0, 600.0, 0.0, lc};
Point(7) = {750.0, 550.0, 0.0, lc};
Point(8) = {900.0, 400.0, 0.0, lc};
Point(9) = {800.0, 200.0, 0.0, lc};
Point(10) = {600.0, 300.0, 0.0, lc};
Point(11) = {350.0, 350.0, 0.0, lc};
Point(12) = {250.0, 500.0, 0.0, lc};

// --- Curva cerrada (Spline) ---
// IMPORTANTE: incluir el primer punto al final para cerrar
Spline(1) = {1,2,3,4,5,6,7,8,9,10,11,12,1};

// --- Crear un Line Loop con la curva cerrada ---
Line Loop(1) = {1};

// --- Crear superficie a partir del loop ---
Plane Surface(1) = {1};

```



Extruimos para obtener 3 capas

```

height_res= 100;
rsv_layers=3;

```

```

// Altura de cada capa
h_layer = height_res / rsv_layers;

// Extrude surface with embedded features

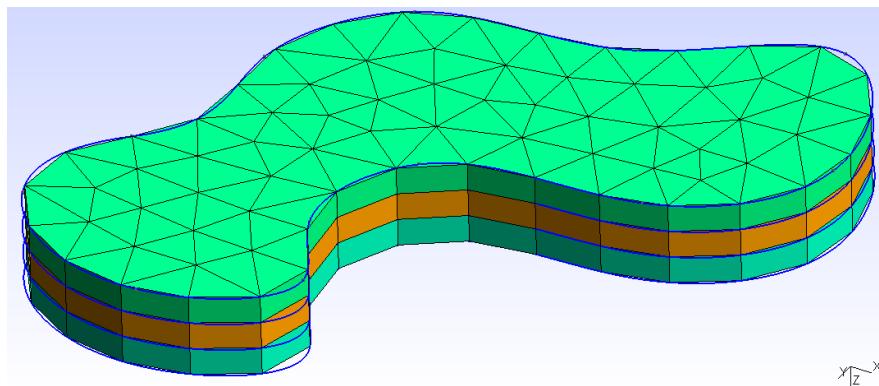
// --- Extrusión capa 1 ---
sr1[] = Extrude {0, 0, h_layer}{ Surface {1}; Layers{1}; Recombine;};
Physical Volume("matrix_layer1", 9991) = {1};

// --- Extrusión capa 2 ---
sr2[] = Extrude {0,0,h_layer} { Surface{sr1[0]}; Layers{1}; Recombine; };
Physical Volume("matrix_layer2", 9992) = {2};

// --- Extrusión capa 3 ---
sr3[] = Extrude {0,0,h_layer} { Surface{sr2[0]}; Layers{1}; Recombine; };
Physical Volume("matrix_layer3", 9993) = {3};

// Mallado
Mesh 3;
Coherence Mesh;

```



Aplicamos refinamiento en las cercanías de los pozos:

```

lc=80;
lc_wells=10;

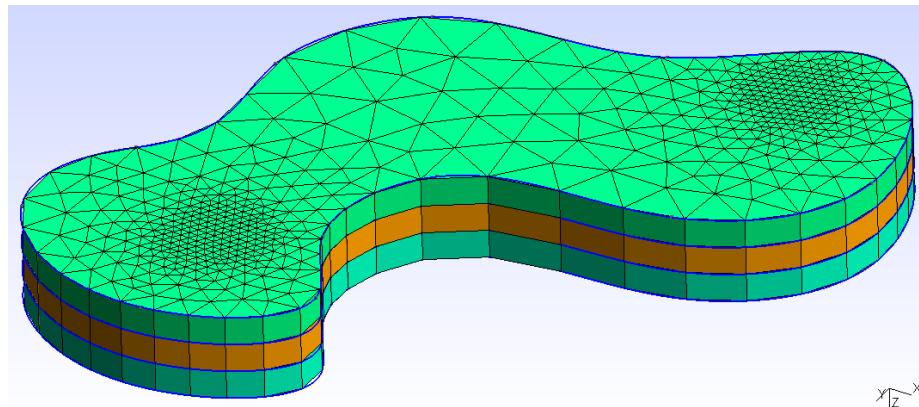
// --- Pozos (puntos de refinamiento) ---
Point(13) = {350, 800, 0, lc}; // inyector
Point(14) = {750, 300, 0, lc}; // productor

// --- Refinamiento de malla usando campos ---
Field[1] = Distance;
Field[1].NodesList = {13,14};

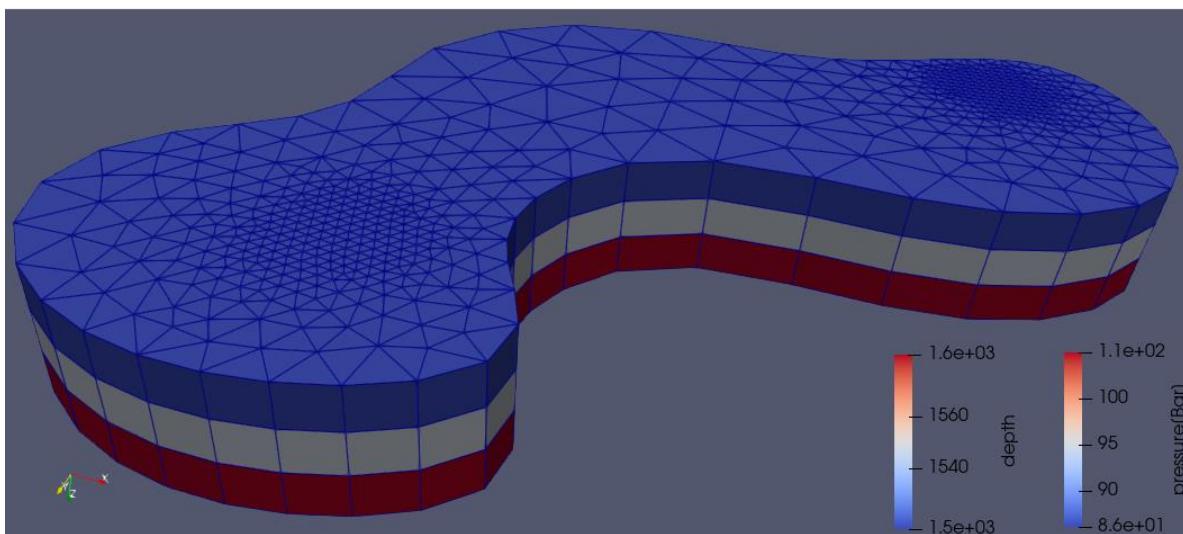
Field[2] = Threshold;
Field[2].InField = 1;
Field[2].SizeMin = lc_wells;
Field[2].SizeMax = lc;
Field[2].DistMin = 50;
Field[2].DistMax = 300;

// --- Aplicar campo ---
Background Field = 2;

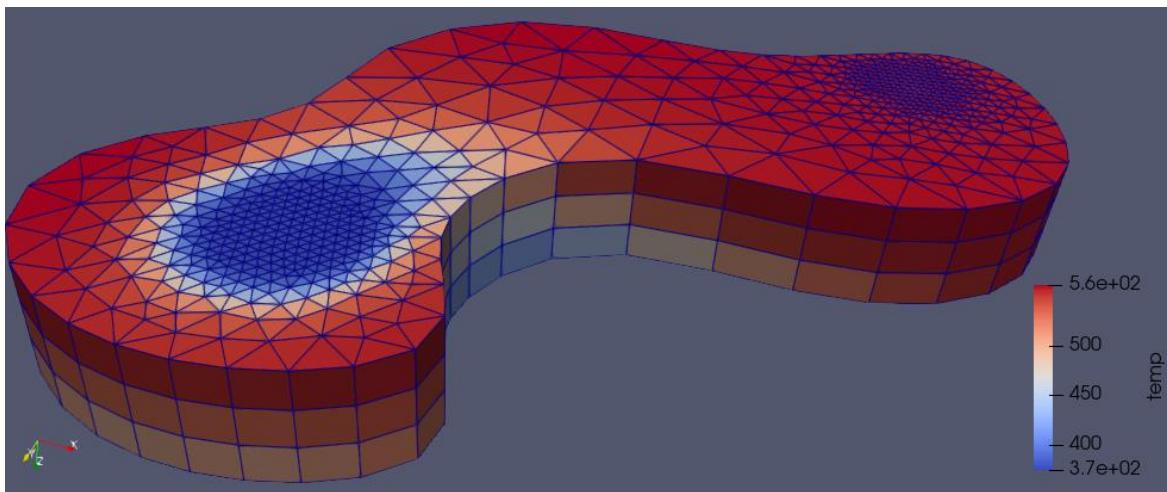
```



Corremos la simulación. Si revisamos el estado inicial, comprobamos que la profundidad y las condiciones iniciales de presión y temperatura se establecieron correctamente:

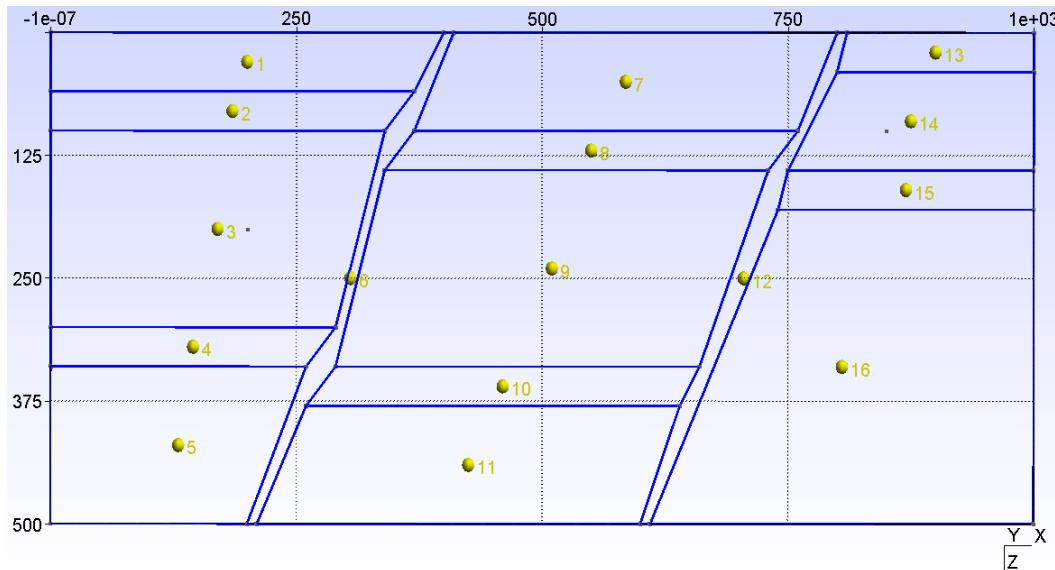


Para el tiempo final vemos que:



Ejemplo 16

Para este ejemplo, simularemos un yacimiento con diferentes estructuras geológicas: 14 estratos y 2 fallas (cada uno con la posibilidad de definir sus propias propiedades) . Para esto, crearemos diferentes volúmenes que representan diferentes estratos, además de las fallas (volúmenes 6 y 12)



Se refina en las ubicaciones de los pozos inyector y productor (en los volúmenes 3 y 14, respectivamente)

```
// Punto donde se desea refinar la malla
Point(50) = {200, 0, 200, lc}; // inyector well coordinate
Point(60) = {850, 0, 100, lc}; // producir well coordinate

// --- Campo Distance ---
Field[1] = Distance;
Field[1].NodesList = {50,60}; // Referencia al punto donde quieras refinar

// --- Campo Threshold ---
Field[2] = Threshold;
Field[2].InField = 1; // Se basa en el campo Distance
Field[2].SizeMin = lc_wells; // Tamaño mínimo de malla cerca del punto
Field[2].SizeMax = lc; // Tamaño máximo lejos del punto
Field[2].DistMin = 50; // Dentro de este radio se aplica SizeMin
Field[2].DistMax = 200; // Despues de este radio se aplica SizeMax

// Aplicar campo como campo de fondo
Background Field = 2;
```

Se extruye en la dirección “y”, estableciendo 3 capas de profundidad

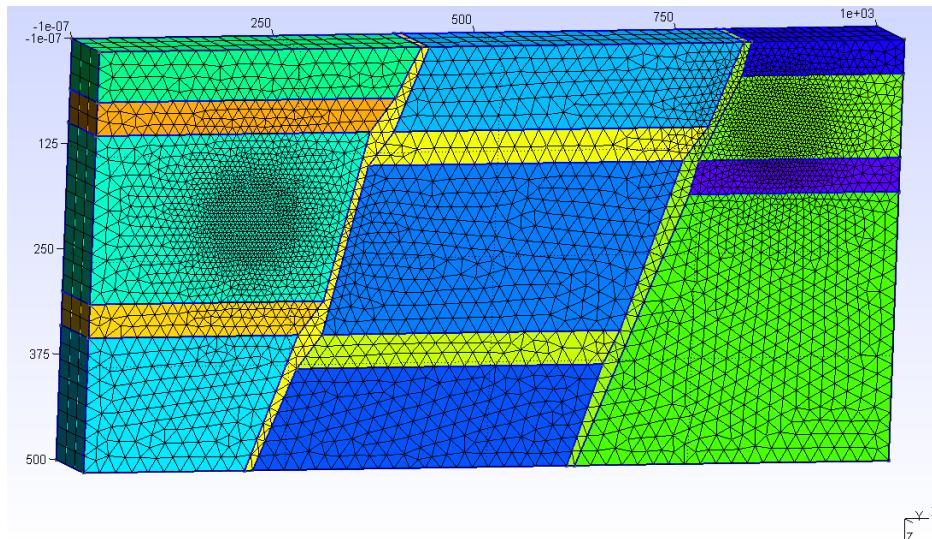
```
height_res= 100;
rsv layers=3;
```

```
// Altura de cada capa
h_layer = height_res;

// Extrude surface with embedded features

// --- Extrusión ---

sr1[] = Extrude {0, h_layer, 0 }{ Surface {1}; Layers{rsy_layers}; Recombine;};
Physical Volume("matrix_layer1", 1) = {1};
```



Se define la ubicación de los pozos:

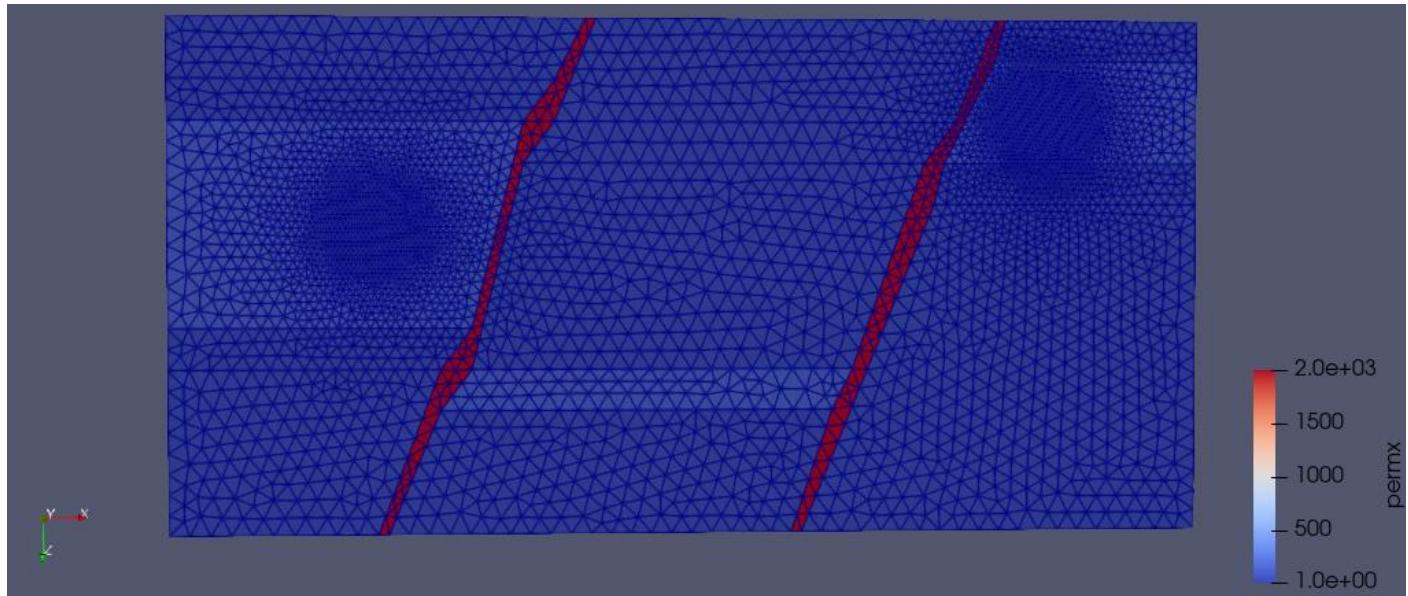
```
self.inj_well_coords = [[200, 0, 200]]
self.prod_well_coords = [[850, 0, 100]]
```

Se establecen valores de permeabilidad alto para las fallas y en los estratos donde se ubican los pozos

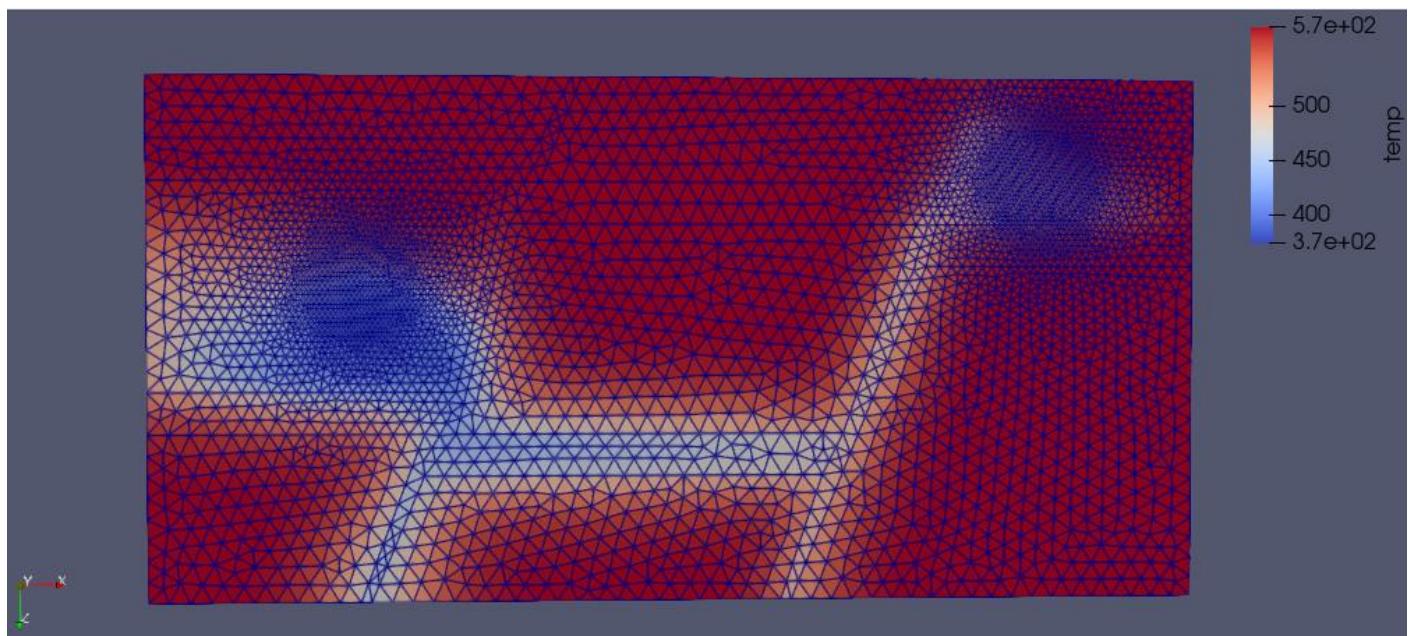
```
def patched_set_layer_properties(self):

    layer_props = {1: Layer_prop( poro=0.01, perm=1., anisotropy=[1, 1, 1]) ,
                   2: Layer_prop(poro=0.01, perm=1., anisotropy=[1, 1, 1]),
                   3: Layer_prop(poro=0.01, perm=100., anisotropy=[1, 1, 1]), # media permeabilidad
                   4: Layer_prop(poro=0.01, perm=1., anisotropy=[1, 1, 1]),
                   5: Layer_prop(poro=0.01, perm=1., anisotropy=[1, 1, 1]),
                   6: Layer_prop(poro=0.01, perm=2000., anisotropy=[1, 1, 1]), # alta permeabilidad
                   7: Layer_prop(poro=0.01, perm=1., anisotropy=[1, 1, 1]),
                   8: Layer_prop(poro=0.01, perm=1., anisotropy=[1, 1, 1]),
                   9: Layer_prop(poro=0.01, perm=1., anisotropy=[1, 1, 1]),
                   10: Layer_prop(poro=0.01, perm=100., anisotropy=[1, 1, 1]), # media permeabilidad
                   11: Layer_prop(poro=0.01, perm=1., anisotropy=[1, 1, 1]), #
                   12: Layer_prop(poro=0.01, perm=2000., anisotropy=[1, 1, 1]), # alta permeabilidad
                   13: Layer_prop(poro=0.01, perm=1., anisotropy=[1, 1, 1]),
                   14: Layer_prop(poro=0.01, perm=100., anisotropy=[1, 1, 1]), # media permeabilidad
                   15: Layer_prop(poro=0.01, perm=1., anisotropy=[1, 1, 1]),
                   16: Layer_prop(poro=0.01, perm=1., anisotropy=[1, 1, 1]) }
```

Corremos la simulación. Si vemos los valores de permeabilidad:

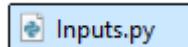


Si vemos el perfil de temperatura:



Ejemplo 17

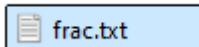
En este ejemplo se mostrará la forma en que se pueden modelar fracturas dentro de DARTS. Para facilitar el manejo de información, primero creamos y cargamos el archivo que contiene los inputs (inputs.py):



Este archivo únicamente contiene una función: input_data_default, la cual, al ser llamada, crea un diccionario con los datos de entrada. Los datos de entrada incluyen, entre otras cosas, las coordenadas de los pozos, el nombre del archivo .txt que contiene las coordenadas de las fracturas, así como los límites del yacimiento:

```
input_data['inj_well_coords'] = [[1000, 1000, 2000]]
input_data['prod_well_coords'] = [[3000, 3000, 2000]]
input_data['x1']=0
input_data['x2']=4000
input_data['y1']=0
input_data['y2']=4000

input_data['frac_file'] = 'frac.txt'
```

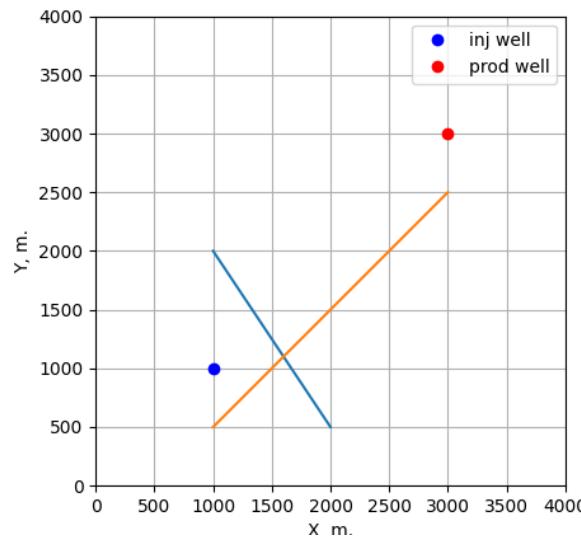


Llamamos la función input_data_default en el archivo main.py

```
from Inputs import input_data_default

input_data = input_data_default()
input_data['frac_file'] = os.path.join(new_directory, input_data['frac_file'])
```

Ploteamos las fracturas y los pozos con la función propia def plot_dfn(input_data):



El siguiente paso es importar las librerías para el pre-procesamiento de las fracturas:

```
from darts.tools.fracture_network.preprocessing_code import frac_preprocessing
```

Esta función requiere los siguientes parámetros:

frac_data_raw	-	input fracture network data in the form <code>[[x1, y1, x2, y2], ..., [...]]</code> [m]
output_dir	-	directory of the output (cleaned) fracture network and potential .geo and .msh results
z_top	0	top surface depth [m]
height_res	50	height of the resulting 1-layer 3D reservoir [m]
box_data	-	coordinates of the bounding box (in order, from bottom left --> bottom right --> top right --> top left) [m]
matrix_perm	1 [mD]	matrix permeability [mD] Al iniciar el modelo, este valor se puede reescribir usando <code>UnstrucReservoir</code> .
margin	25 [m]	margin around the fracture network, used in case no bounding box is given
mesh_clean	False	boolean, if True then will call GMSH to mesh the <code>cleaned</code> fracture network in a triangular mesh. A .msh file will be created.
mesh_raw	False	boolean, if True then will call GMSH to mesh the <code>raw</code> fracture network in a triangular mesh. A .msh file will be created.
decimals	5	number of decimals used to round-off input data in order to remove duplicates we need to have fixed number of decimals

tolerance_zero	1e-10	anything below this threshold is considered absolute zero!
main_algo_iters	2	number of times the main cleaning algorithm is run
apertures_raw	No	list of apertures for each fracture segment [m] Al iniciar el modelo, este valor se puede reescribir usando UnstrucReservoir.
correct_aperture	False	boolean, if True apply aperture correction method
straighten_after_cln	False	boolean, if True then straighten the fractures after cleaning
angle_tol_straighten	7	allowable deviation from a straight line for straightening algorithm
partition_fractures_in_segms	True	boolean, if True partitions the fracture network into smaller subsegments of length char_len
char_len l_f	- necesario	minimum allowable distance between any two vertices in the domain (characteristic length) [m] near fractures gid size large fractures are partitioned into smaller fracture segments with length l_f
merge_threshold h	0.66	scaling parameter chosen on the closed interval from [0.5, 0.86] $l_f \cdot h$ The larger h is, the more simplified the resulting network becomes. The distance between the newly added node and any other node already in the domain must be larger than $l_f \cdot h$, the node is merged into the closest node already in the domain.
char_len_mult	1	multiplier for mesh characteristic length
char_len_boundary	None	characteristic mesh length on the boundary (before multiplier) characteristic length of mesh elements at the boundary grid size near grid boundaries [m]
char_len_well	None	grid size near wells [m]
angle_tol_small_intersect	20	minimum angle in degrees between two intersecting fractures
tolerance_intersect	1e-10	
calc_intersections_before	False	boolean, if True calculates intersections between fractures segments before cleaning
calc_intersections_after	True	boolean, if True calculates intersections between fractures segments after cleaning
num_partition_x	1	number of partitions used in fracture intersection calculation in x-direction (parallel computing required) number of partitions for parallel implementation of intersection finding algorithm
num_partition_y	1	number of partitions used in fracture intersection calculation in y-direction (parallel computing required) number of partitions for parallel implementation of intersection finding algorithm

small_angle_iter	0	number of iterations with small-angle correction algorithm
wells	None	----
input_data	None	El resto de variable de entrada

Además, se deben de dar los siguientes argumentos como parte de la variable ‘input_data’:

input_data['rsv_layers']	1	extrusion - number of layers by Z axis
input_data['overburden_thickness']	0	# # no overburden layers (fractured) by default
input_data['overburden_layers']	0	# # no overburden layers (fractured) by default
input_data['underburden_thickness']	0	# # no overburden layers (fractured) by default
input_data['underburden_layers']	0	# # no overburden layers (fractured) by default
input_data['overburden_2_thickness']	0	# # no second overburden layers (without fractures) by default
input_data['overburden_2_layers']	0	# # no second overburden layers (without fractures) by default
input_data['overburden_2_layers']	0	# # no second overburden layers (without fractures) by default
input_data['underburden_2_layers']	0	# # no second overburden layers (without fractures) by default

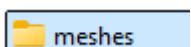
Por ejemplo, la información mínima requerida para llamar la función ‘frac_preprocessing’ es:

```
frac_preprocessing(output_dir=output_dir, frac_data_raw=np.genfromtxt(input_data['frac_file']),
                   char_len=input_data['char_len'], # partition_fractures_in_segms=False,
                   input_data=input_data,
)
```

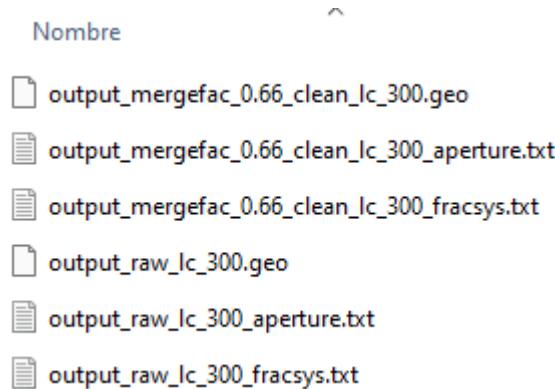
En la línea de arriba, np.genfromtxt() lee el archivo de texto (generalmente CSV, TXT, etc.) y lo convierte en un arreglo (array) de NumPy. Los demás argumentos son:

char_len	300
merge_threshold	0.66 (valor default)

Mientras que input_data contiene los valores default de ‘overburden’ mostrados más arriba (aunque el modelo no utilice datos de ‘overburden’, estos deben de ser pasados a la función). Al llamar la función se creará la siguiente carpeta:



La cual contiene la información tanto de la malla limpia como de la malla bruta ('clean' y 'raw'). Los valores dados para char_len y merge_threshold se incluyen en el nombre del archivo (para limpiar la malla, se hace uso del valor dado en merge_threshold, por lo que este valor aparece únicamente en el nombre de la malla 'limpia'):



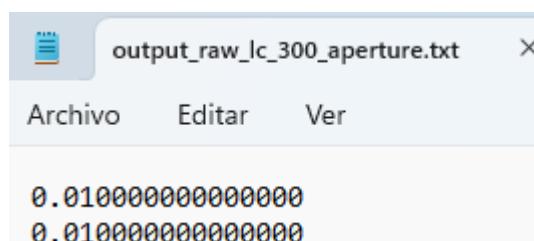
Si abrimos el archivo _aperture.txt, vemos que contiene la apertura en metros de cada una de las fracturas. Como en este caso no especificamos este valor, se dará el valor "default" (0.0001)



Si al momento de llamar frac_preprocessing incluimos la lista de aperturas para cada fractura:

```
frac_preprocessing(output_dir=output_dir, frac_data_raw=np.genfromtxt(input_data['frac_file']),
                   char_len=input_data['char_len'], # partition_fractures_in_segs=False,
                   input_data=input_data,
                   apertures_raw=np.array([.01, .01]))
```

Vemos que el valor en el archivo _aperture.txt cambia:

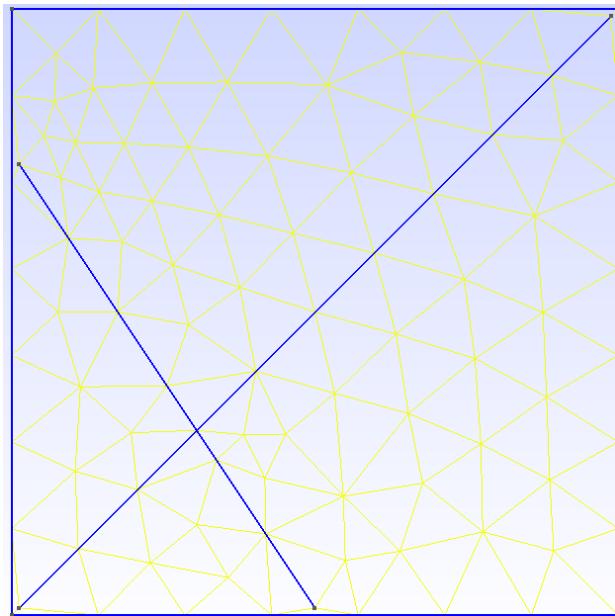


El valor de aperturas usadas en la simulación se puede modificar al momento de llamar Unstructreservoir (el archivo _aperture.txt no reflejara los cambios ya que se crea al momento de generar la malla). Aquí, la variable

frac_aper puede ser un escalar o un vector. Reescribe los valores de frac_aper que se pudieron haber establecido al momento de usar frac_preprocessing (lo mismo aplica para los valores de permeabilidad). Por ejemplo, si quisieramos establecer una apertura de 0.01 [m] para todas las fracturas:

```
frac_aper= 0.01 # (initial) fracture aperture [m] = 1e-2 [m] = 0.01 [m] = 10 [mm]
# initialize reservoir
self.reservoir = UnstructReservoir(timer=self.timer, mesh_file=mesh_file,
                                     permx=permx, permy=permy, permz=permz,
                                     poro=poro,
                                     rcond=rcond,
                                     hcap=hcap,
                                     frac_aper=frac_aper)
```

Al abrir el archivo .geo de la malla sin limpiar ('raw') en Gmsh:



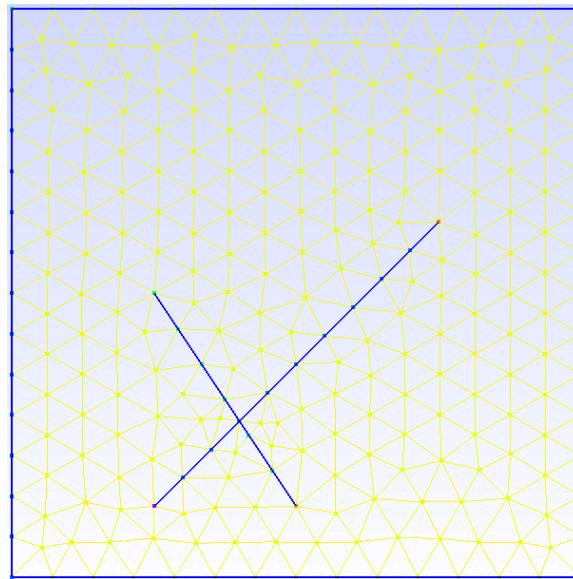
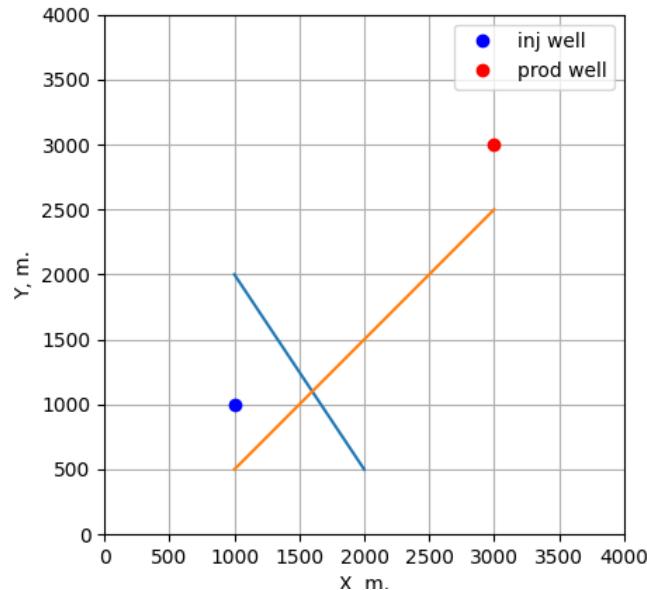
En la malla generada anteriormente, vemos que, por default, el dominio se extiende solo hasta abarcar la red de fracturas en su totalidad, y no del yacimiento completo. Para especificar el límite del dominio, se debe incluir el argumento box_data:

```
frac_preprocessing(output_dir=output_dir, frac_data_raw=np.genfromtxt(input_data['frac_file']),
                   char_len=input_data['char_len'], # partition_fractures_in_segs=False,
                   input_data=input_data,
                   apertures_raw=np.array([.01, .01]),
                   box_data=np.array(input_data['box_data']),
```

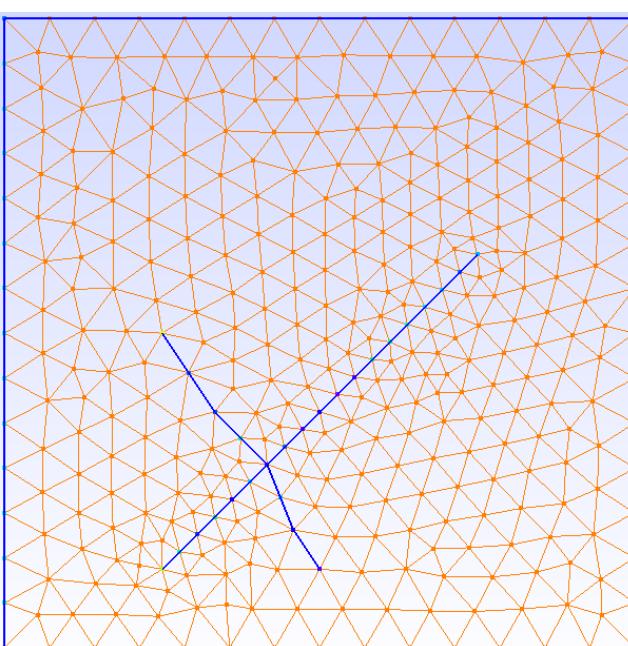
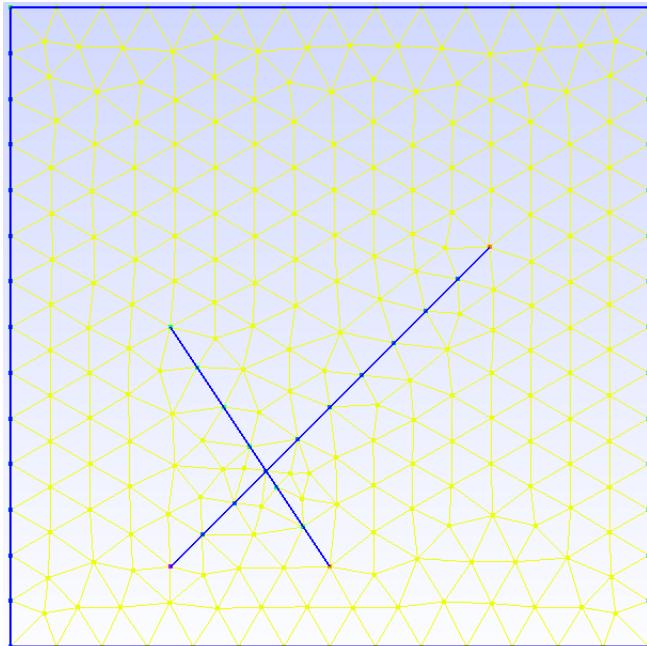
Definido de la siguiente forma dentro del archivo inputs:

```
input_data['box_data']=[  
[input_data['x1'], input_data['y1']], # bottom left  
[input_data['x2'], input_data['y1']], # bottom right  
[input_data['x2'], input_data['y2']], # top right  
[input_data['x1'], input_data['y2']] # top left  
]
```

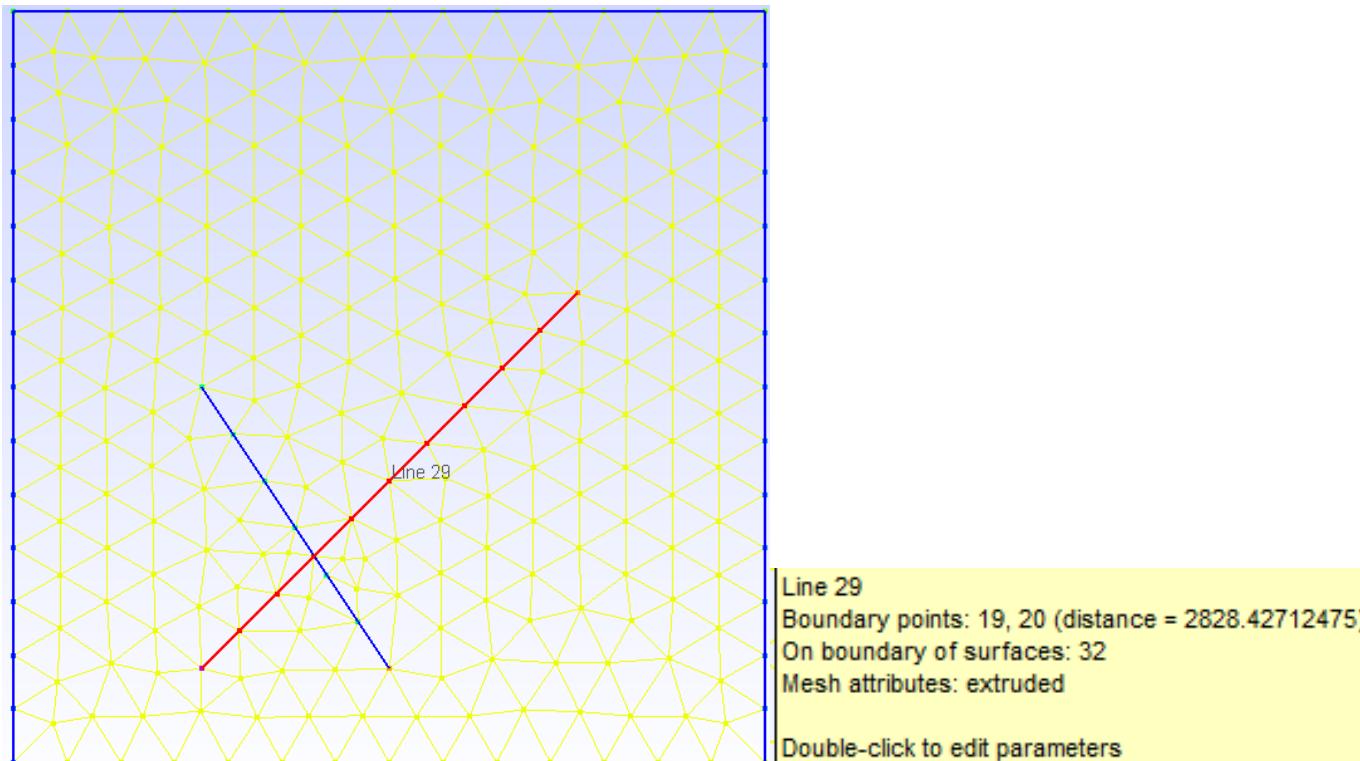
De esta manera la malla resultante tendrá el dominio deseado:



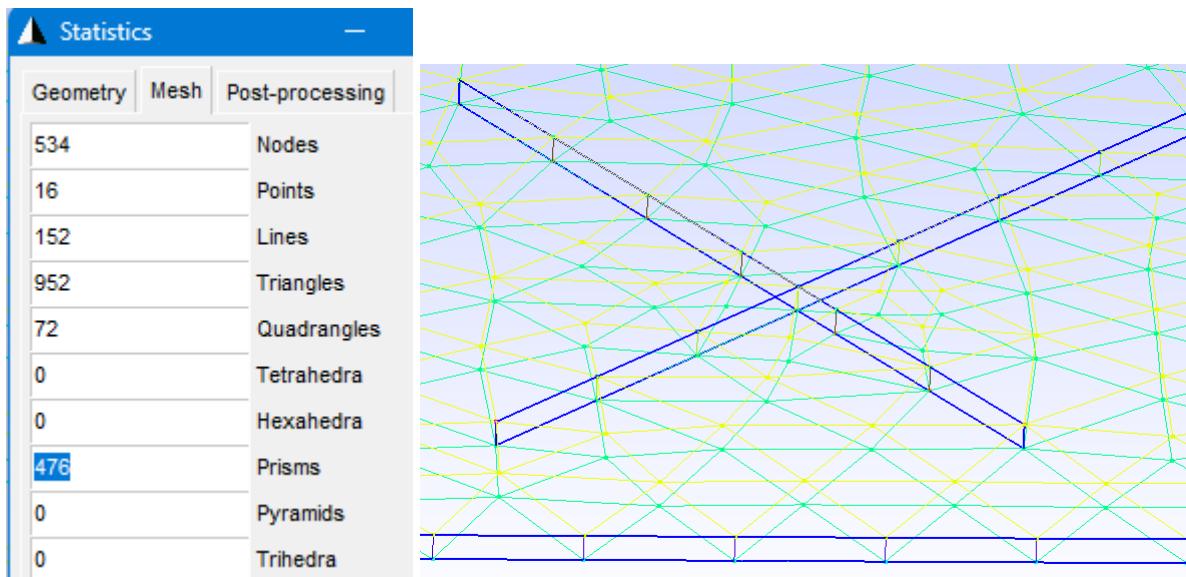
Si comparamos los dos archivos de malla .geo (raw y clean):



Vemos que, debido a que el sistema de fracturas es tan simple, para este caso particular, ambas mallas son casi idénticas (no existe una limpieza como tal de la malla). Trabajaremos con la malla raw y más adelante veremos más detalles sobre la malla limpia. También podemos notar que los vértices de las celdas alrededor de las fracturas han sido establecidos de acuerdo con el valor de `char_length` establecido previamente. Por ejemplo, la fractura en rojo (línea 29) tiene una longitud de 2838 [m] y el valor de `char_length` es de 300 [m]. Entonces, el pre procesador tratará de partir dicha fractura en $2838/300 = \pm 10$ elementos. Cada uno de estos 10 elementos representará una celda en el dominio computacional del tipo fractura. Se hará lo mismo para la otra fractura (6 elementos), por lo que el total de celdas de fractura serán de 16.



Si analizamos la malla resultante y las estadísticas, notamos que contiene 476 prismas (celdas de matriz). Este número no es el total de celdas en el modelo, ya que debemos sumarle las celdas que corresponden a las fracturas (16), por lo que el total de celdas será de $476+16=492$. Es importante mencionar que las fracturas son representadas mediante elementos 'quad', pero no todos los 'quad' son fracturas. Las fronteras norte, sur, este y oeste están construidos mediante elementos 'quad' también, mientras que los triángulos representan las fronteras inferior y superior.



Nota: Hay veces que el número de triángulos mostrados es el doble, es decir, para cada celda o prisma, se cuenta la tapa inferior y superior.

Una vez creada la malla, el siguiente paso es cargarla como parte DARTS. Sin embargo, DARTS no utiliza el archivo .geo como tal, sino que requiere de un archivo del tipo .msh, el cual tiene la información de la malla. Para generararlo, se debe agregar el argumento `mesh_raw` y/o `mesh_clean` al momento de llamar `frac_preprocessing`:

```
frac_preprocessing(output_dir=output_dir, frac_data_raw=np.genfromtxt(input_data['frac_file']),
                   char_len=input_data['char_len'], # partition_fractures_in_segs=False,
                   input_data=input_data,
                   apertures_raw=np.array([.01, .01]),
                   box_data=np.array(input_data['box_data']),
                   mesh_raw=True,
                   mesh_clean=True,
)
```

Esto generará los archivos .msh que utiliza DARTS para la simulación:

- `output_mergefac_0.66_clean_lc_80.geo`
- `output_mergefac_0.66_clean_lc_80.msh`
- `output_mergefac_0.66_clean_lc_80_aperture.txt`
- `output_mergefac_0.66_clean_lc_80_fracsystxt`
- `output_raw_lc_80.geo`
- `output_raw_lc_80.msh`
- `output_raw_lc_80_aperture.txt`
- `output_raw_lc_80_fracsystxt`

El siguiente paso es cargar estos archivos al modelo creado. Para esto, se utiliza la clase `UnstructReservoir`:

```
from darts.reservoirs.unstruct reservoir import UnstructReservoir
```

También se utiliza la siguiente librería para leer formatos de malla:

```
import meshio
```

Se define el nombre del archivo de malla:

```
fname = f"output_raw_lc_{self.input_data['char_len']}.msh"
mesh_file = os.path.join('meshes', fname)
```

Y se utiliza la clase `UnstructReservoir`:

El siguiente paso es leer la malla y leer las etiquetas que tiene cada tipo de celda del archivo .msh, para posteriormente asignar a cada celda un tag físico ('matriz', 'fracture' o 'boundary') dentro del modelo de DARTS. Se asume que la malla está extruida (es decir, que fue generada extendiendo 2D a 3D) y que las fracturas tienen forma cuadrilateral.

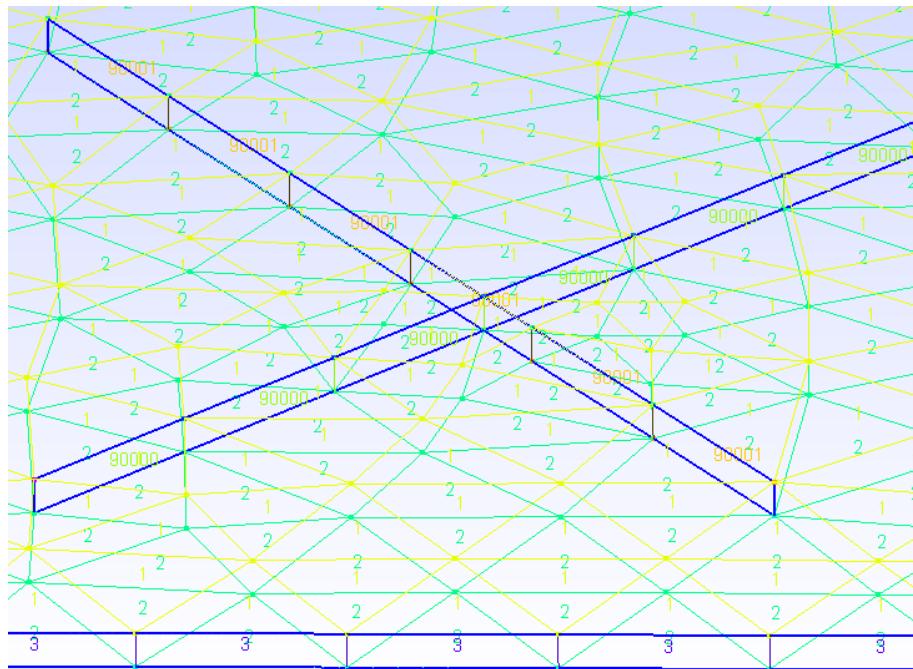
```
msh = meshio.read(mesh_file)
c = msh.cell_data_dict['gmsh:physical']
```

c es un diccionario donde cada clave es un tipo de figura geométrica (como 'triangle', 'quad', 'tetra', etc.), y los valores son arrays con los *physical tags* de esas celdas. Los elementos de las fronteras tiene *physical tags* del 1 al 6 (cara inferior, superior , norte , sur, este y oeste), mientras que las etiquetas de fracturas comienzan desde 90000 según el código de generación del archivo .geo. Los prismas, que corresponden a la matriz, tienen un *physical tag* correspondiente a 9991

	tamaño	<i>physical tag</i>	Corresponde a
'triangle'	952	1,2	Fronteras (inferior y superior)
'quad'	72	3,4,5,6, 90000, 90001 ***Nota: en este ejemplo solo hay 2 fracturas principales	Fronteras (norte, sur, este, oeste) + Fracturas (fractura #1, fractura #2)
'wedge'	476	9991	matriz principal (reservorio)

***Nota: Hay veces que se la malla solo genera una de las fronteras inferiores o superiores, por lo que se tendrá la mitad de triángulos y su valor únicamente será 1 o 2.

Para ver los physical tags en gmsh:



Como se dijo anteriormente, las fracturas son representadas mediante elementos ‘quad’, pero no todos los ‘quad’ son fracturas (los bordes del dominio están construidos mediante elementos ‘quad’). Para conocer el número de fracturas, se obtienen todos los tags únicos asignados a celdas cuadriláteras y filtra solo los que corresponden a fracturas. Las etiquetas de fracturas comienzan desde 90000 según el código de generación del archivo .geo.

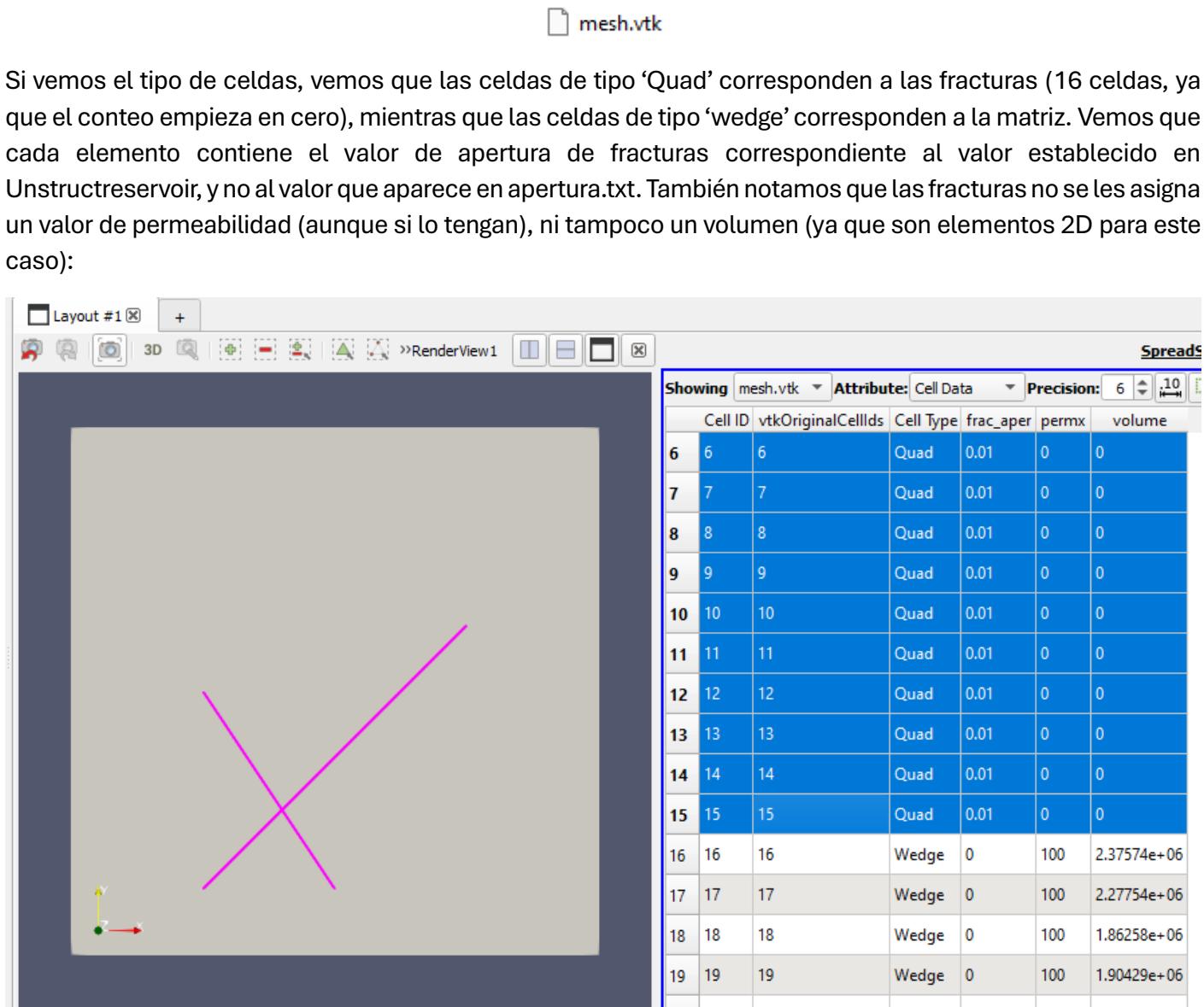
```
n_fractures = (np.unique(c['quad']) >= 90000).sum()
```

en este caso será igual a 2 ya que solo hay 2 fracturas (aunque hayan sido particionadas, solo cuenta el número de fracturas originales)

El siguiente paso es asignar los tags físicos a DARTS

```
# Asigna los tags físicos [9991, 9992, 9993, 9994, 9995] para la matriz de roca.
# 9991 - rsv, 9992 - overburden, 9993 - underburden, 9994 - overburden2, 9995 - underburden2
# • 9991: matriz principal (reservorio)
# • 9992: sobrecarga
# • 9993: bajo carga
# • 9994 y 9995: capas extra si las hay
self.reservoir.physical_tags['matrix'] = [9991 + i for i in range(5)]
# multiplied by 3 because physical surfaces for fracture are also in underburden and overburden
self.reservoir.physical_tags['fracture'] = [90000 + i for i in range(n_fractures)]
self.reservoir.physical_tags['boundary'] = [2, 1, 3, 4, 5, 6] # order: Z- (bottom); Z+ (top) ; Y-; X+; Y+; X-
```

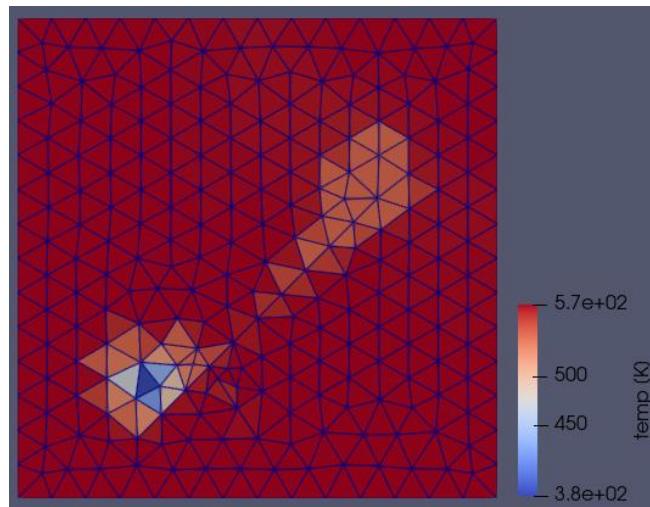
Si vemos el archivo de la malla



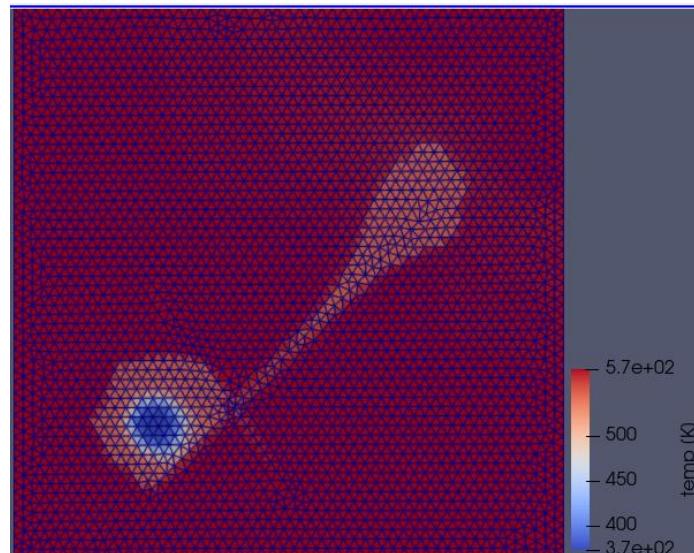
El número total de celdas son 492 (de nuevo, hay que sumar uno ya que el conteo empieza en cero)

491	491	491	Wedge	0	100	1.0936e+06
-----	-----	-----	-------	---	-----	------------

Si corremos la simulación y visualizamos el campo de entalpia dentro del archivo .vtk resultante, podemos ver como el agua de inyección se canaliza por las fracturas, resultando en una caída de energía y/o temperatura



Si reducimos el número de char_length (de 300 a 80):



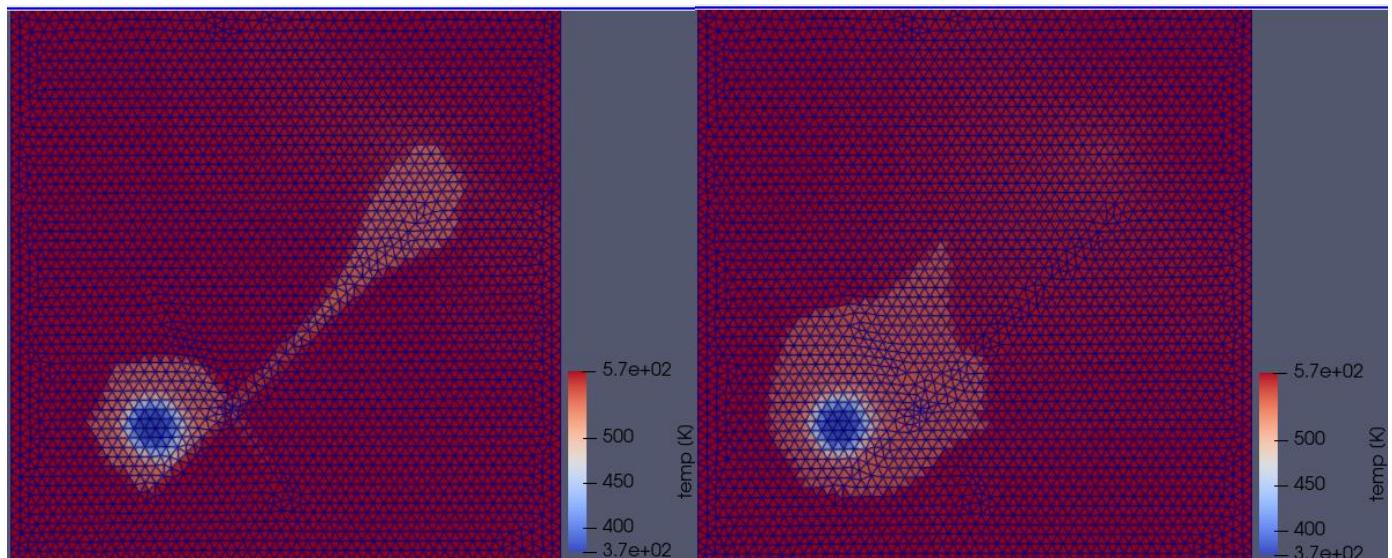
Vemos que las fracturas serán particionadas en un número mayor de elementos, lo que resultara en un refinamiento mayor. Si reducimos la apertura de las fracturas (de 0.001 a 0.0001):

```
frac_aper= 0.0001 # (initial) fracture aperture [m] = 1e-3 [m] = 0.001 [m] = .1 [mm]
# initialize reservoir
self.reservoir = UnstructReservoir(timer=self.timer, mesh_file=mesh_file,
                                     permx=permx, permy=permy, permz=permz,
                                     poro=poro,
                                     rcond=rcond,
                                     hcap=hcap,
                                     frac_aper=frac_aper)
```

$$k_f = a^2/12$$

Donde a=apertura de la fractura en [m]

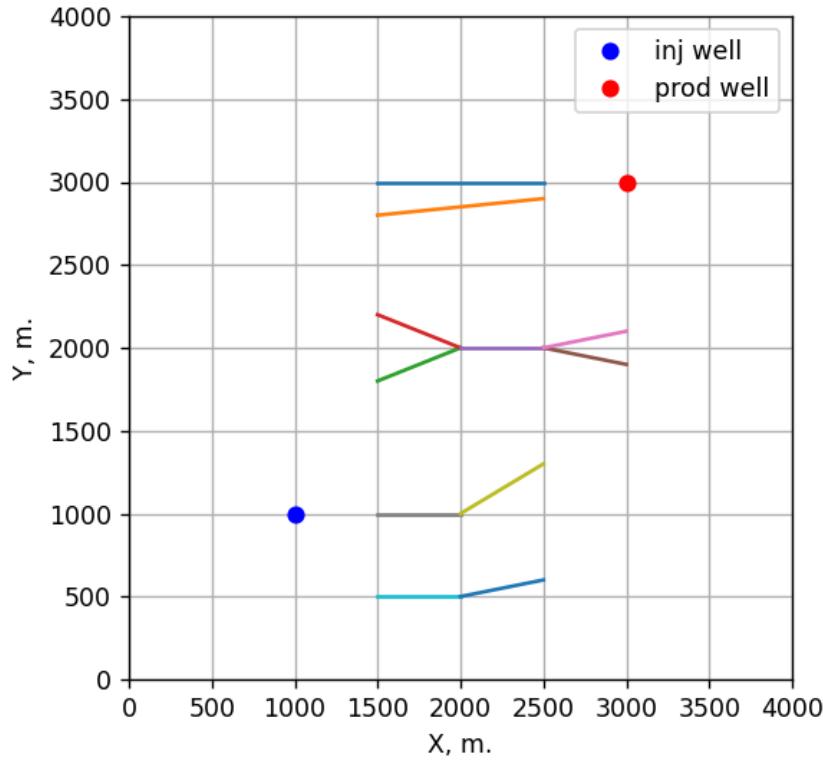
K=permeabilidad de la fractura, en [m²]



Vemos que ahora el medio se comportará más como un medio homogéneo

Ejemplo 18

En este ejemplo se muestra las principales formas en que el pre-procesador logra simplificar la malla. Para esto, se define la siguiente red de fracturas:



A continuación, se seleccionan algunas parámetros clave para el pre-procesador

```
frac_preprocessing(output_dir=output_dir, frac_data_raw=np.genfromtxt(input_data['frac_file']),
                   char_len=input_data['char_len'], # partition_fractures_in_segs=False,
                   input_data=input_data,
                   apertures_raw=np.array([.01, .01]),
                   box_data=np.array(input_data['box_data']),
                   mesh_raw=True,
                   mesh_clean=True,

                   straighten_after_cln=True,
                   angle_tol_straighten=20,      # fracturas con angulo menor, se enderezan

                   small_angle_iter=1,
                   angle_tol_small_intersect=35, # fracturas con angulo menor, se unen en una sola fractura
                   merge_threshold=input_data['merge_threshold'],
```

Se activa la opción de “Straighten_after_cln” y se establece un angulo de 20°. Con esto, el preprocesador tratará de “enderzar” aquellas fracturas cuyo angulo sea menor a este parámetro. También se establece el valor de 1 para la opción de “small_angle_iter” (el valor default es cero). Con esto, el preprocesador tratará de unir aquellas fracturas que formen un ángulo menor al valor establecido (35° para este caso).

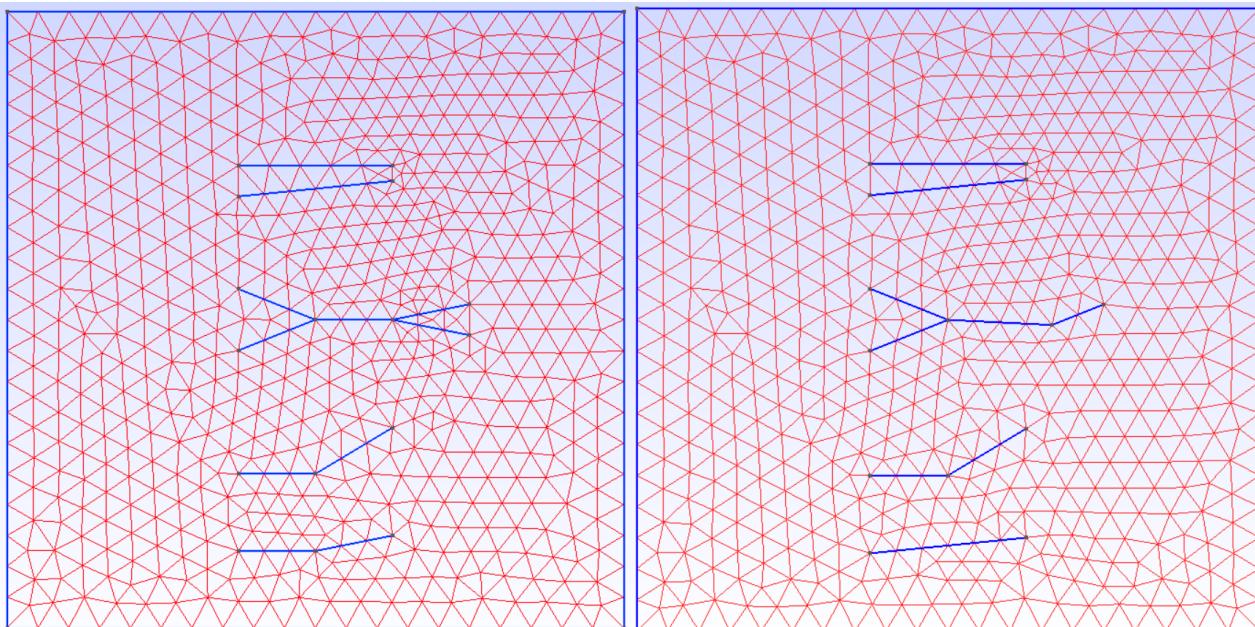
Adicionalmente, el valor de merge_threshold tiene un valor de 0.5 (el más bajo posible):

```
input_data['char_len']=200
input_data['merge_threshold']=.5    # [0.5 -- 0.86],   # 200*.5= 100
```

Recordemos que `merg_threshold` es un parámetro de escalamiento, el cual se relaciona con la distancia entre nodos y dicta la distancia máxima permitida entre dos nodos. Por ejemplo, para este caso en particular se tiene que:

$$d_{max} = l_f * h = 200 * 0.5 = 100$$

Por lo que cualesquier dos nodos que estén a una distancia menor a 100 se unirán en un solo nodo. Entre mayor sea el valor de `merge_threshold`, mayor será la simplificación del sistema resultante. Si correos el arhcivo y visualizamos la malla “cruda” y “limpia”

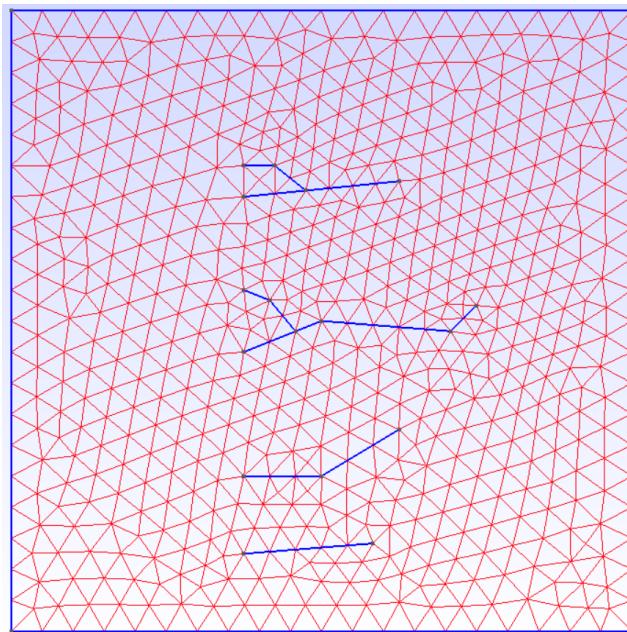


Podemos ver que la red de fracturas ha cambiado. La fractura del fondo se ha “enderezado”, mientras que la fractura superior no (debido al ángulo establecido). También podemos ver que en la fractura de en medio, los tramos del lado derecho se han unido, ya que el ángulo formado entre ellos es menor al ángulo establecido. Si incrementamos el valor al máximo permitido:

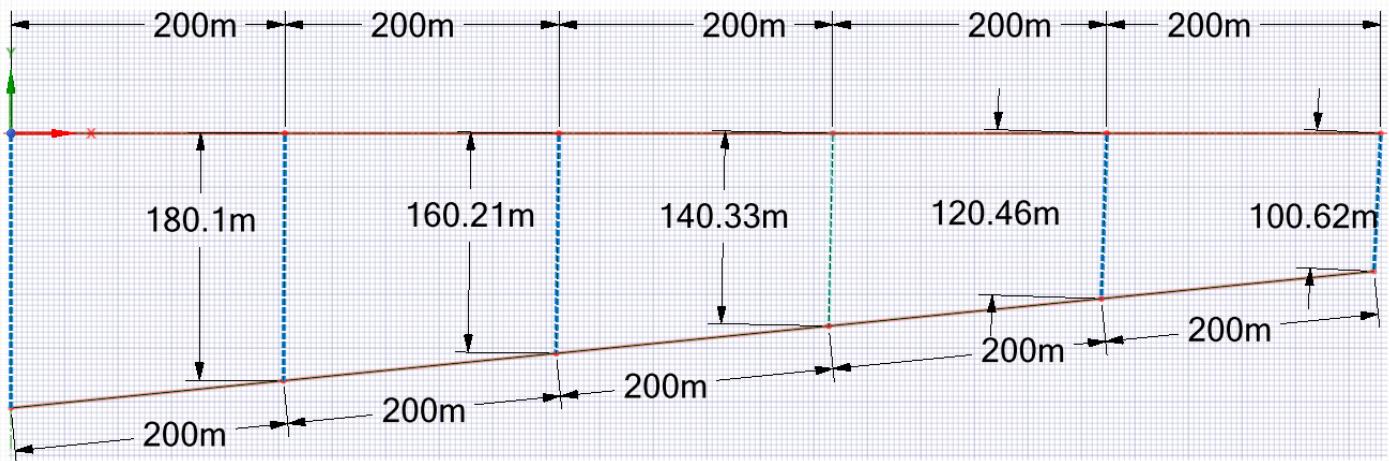
```
input_data['merge_threshold']=.86    # [0.5 -- 0.86],   # 200*.86= 172
```

$$d_{max} = l_f * h = 200 * 0.5 = 100$$

Por lo que cualesquier dos nodos que estén a una distancia menor a 172 se unirán en un solo nodo. Si volvemos a correr el arhcivo:



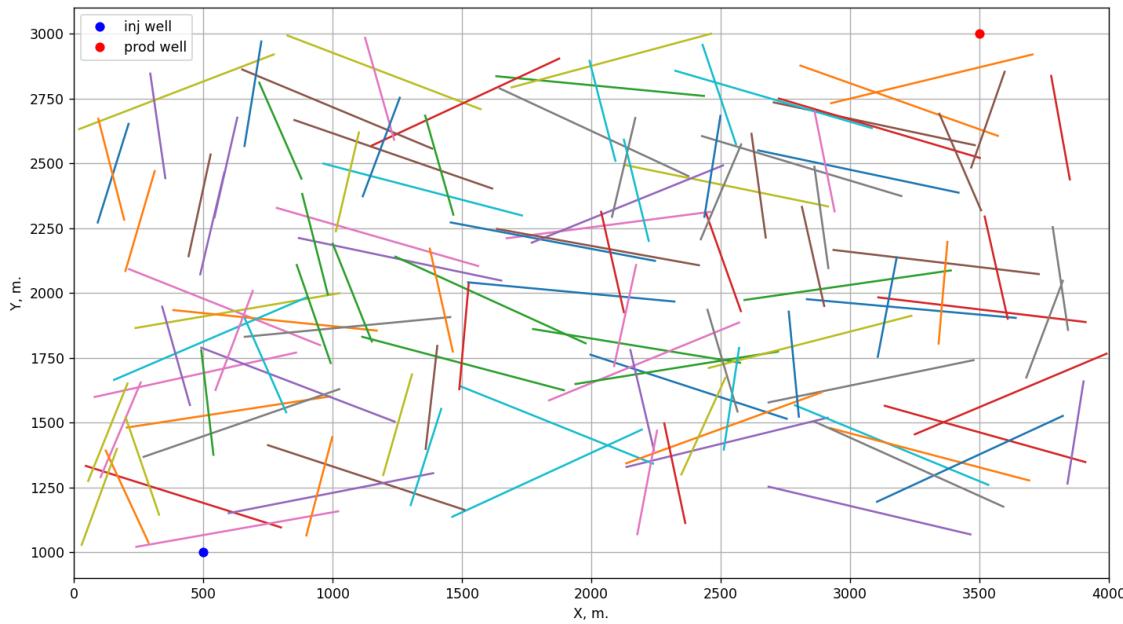
Vemos que la malla resultante se ha simplificado aún más. El pre procesador tratará de unir aquellos nodos que se encuentre muy cerca de sí. Es por eso que las dos primeras fracturas se han unido parcialmente.



Es importante mencionar que el preprocesador utiliza un método iterativo, por lo que la simplificación resultante dependerá del número de iteraciones establecidas.

Ejemplo 19

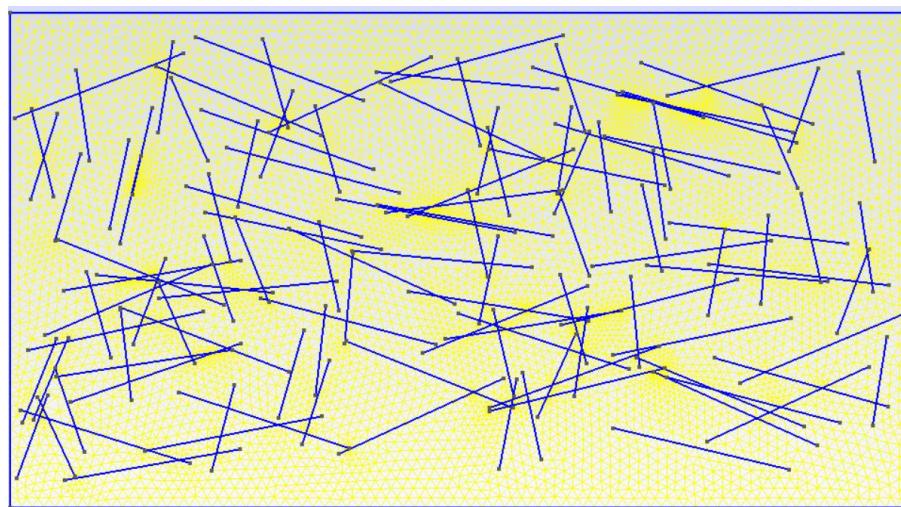
En este ejemplo veremos un sistema más complejo, y como el pre procesador puede ayudar a simplificar la red resultante:

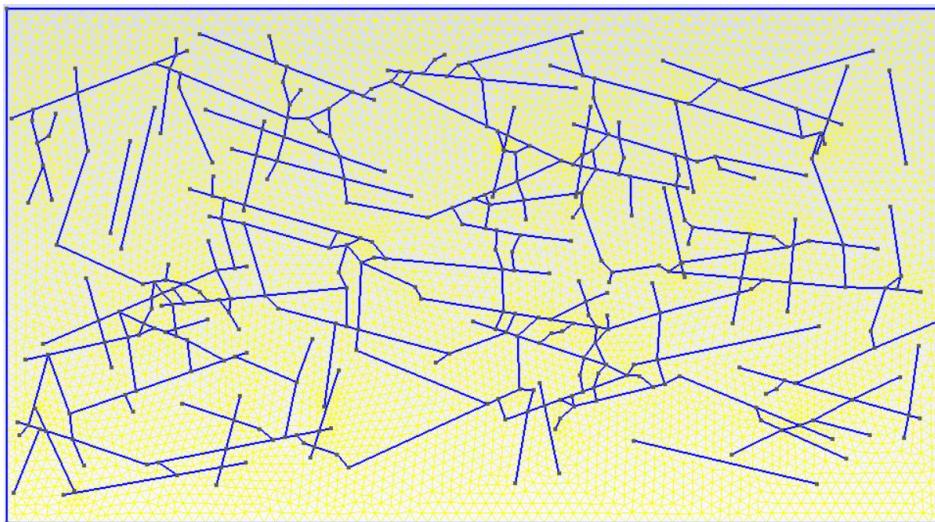


Se utilizan los siguientes valores de:

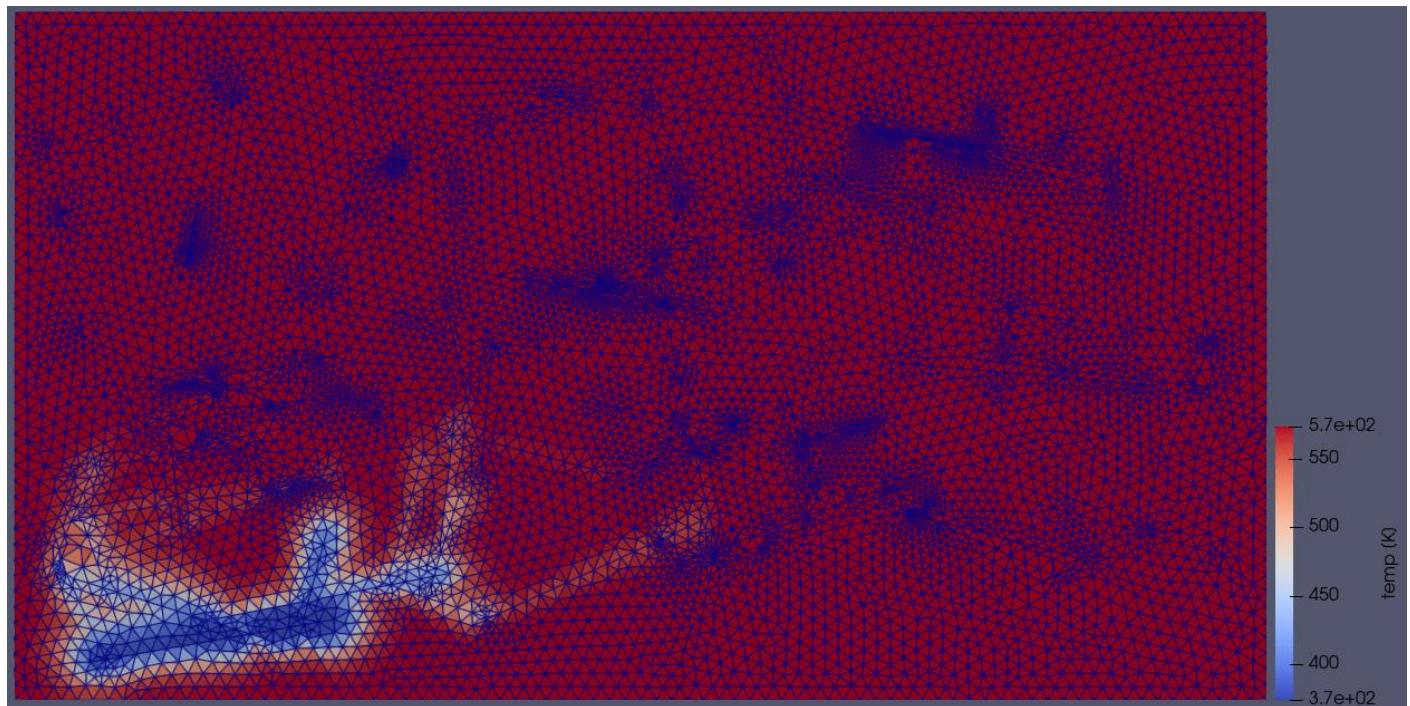
```
input_data['char_len']=50
input_data['merge_threshold']=.86      # [0.5 -- 0.86],   # 200*.5= 100
```

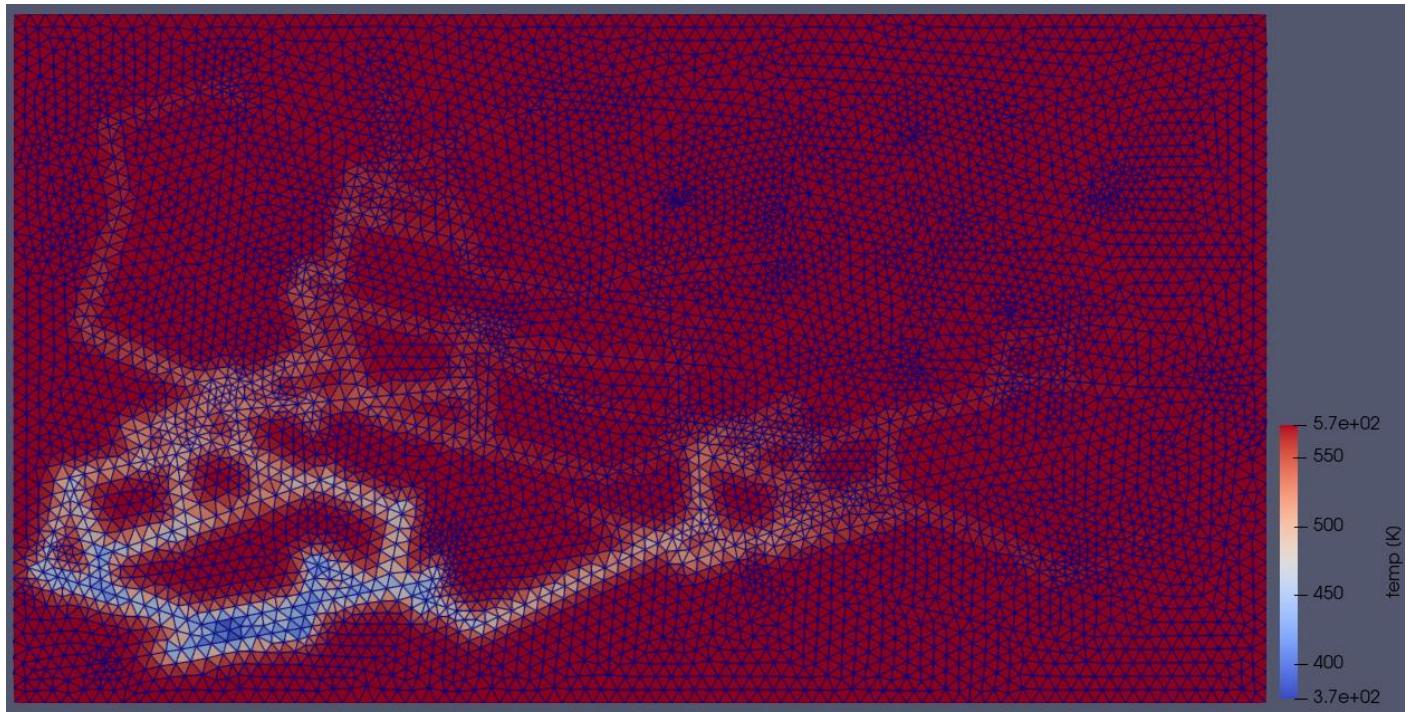
Si vemos las mallas resultantes:





En paraview podemos identificar fácilmente el número de celdas totales. El numero de celdas del modelo se redujo de 17063 a 13435 debido a la simplificación de la malla. Si emos los perfiles de temperatura:





Si cambiamos el valor de char_len:

```
input_data['char_len']=25
```

