Final Exam (part 2) - Computational Physics I

Deadline: Tuesday 18 June 2024 (by 17h00)

Credits: 10 points

18.5/20

Name: Males-Araujo Yorlan

6. The Orszag-Tang vortex: Calculus and Fourier analysis (10 points)

We want to study the properties of turbulent flows using numerical calculus and Fourier analysis. Let us consider the Orszag-Tang vortex system, which describes a doubly periodic fluid configuration leading to 2D supersonic magnetohydrodynamical (MHD) turbulence. Although an analytical solution is not known, its simple and reproducible set of initial conditions has made it a widespread benchmark for the comparison of MHD numerical solvers.

The computational domain is periodic box with dimensions: $[0, 2\pi]^D$ where D is the number of spatial dimensions.

In code units, the initial conditions are given by:
$\vec{v} = (-\sin y,\ \sin x)\ , \qquad \vec{B} = (-\sin y,\ \sin 2x)\ , \qquad \rho = 25/9\,, \qquad p = 5/3.$

The numerical simulation produces 61 VTK files stored in:

- the **Orszag-Tang-MHD** folder:

[https://github.com/wbandabarragan/computational-physics-1/blob/main/sample-data/Orszag_Tang-MHD.zip (https://github.com/wbandabarragan/computational-physics-1/blob/main/sample-data/Orszag_Tang-MHD.zip)](https://github.com/wbandabarragan/computational-physics-1/blob/main/sample-data/Orszag_Tang-MHD.zip)

jointly with:

- a **units.out** file that contains the CGS normalisation values.
- a **vtk.out** file whose second column contains the times in code units.
- a **grid.out** file that contains information on the grid structure.

You can use VisIt to inspect the data. The written fields are:

- density (rho)
- thermal pressure (prs)
- velocity_x (vx1)
- velocity_y (vx2)
- magnetic_field_x (Bx1)
- magnetic_field_y (Bx2)

Reference paper: [https://arxiv.org/pdf/1001.2832.pdf (https://arxiv.org/pdf/1001.2832.pdf)](https://arxiv.org/pdf/1001.2832.pdf)

"High-order conservative finite difference GLM-MHD schemes for cell-centered MHD", Mignone, Tzeferacos & Bodo, JCP (2010) 229, 5896.

Numerical calculus:

(a) Create a set of Python functions that reads in the simulation data, normalises the data fields to CGS units (using units.out), interpolates them into a mesh, and sequentially prints the following figures into a folder called "output_n" for all times:

- Density, ρ
- Velocity divergence, $\vec{\nabla} \cdot \vec{v}$
- Velocity curl (vorticity) magnitude, $|\vec{\nabla} \times \vec{v}|$

Add time-stamps in CGS units to the above maps (using information from vkt.out).

In this part, we will reuse some code from the second homework.

In [1]:
```
# Importing libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import pyvista as pv
import os
from skimage.transform import resize
```

In [2]:
```python
# First of all, the units:
def units(units_file):
    """
    Stores the normalisation values for length, velocity, density, pre
    magnetic field and time into callable objects, and then returns th
    Inputs: units_file -> variable storing the units file
            time_file -> variable storing the time data
    Outputs:
    Author: MAY
    """
    # Read the cgs units file:
    df_units = pd.read_csv(units_file)

    # Get the units into python objects:
    rho_0 = np.array(df_units.loc[df_units["variable"] == "rho_0"]["no
    v_0 = np.array(df_units.loc[df_units["variable"] == "v_0"]["norma
    l_0 = np.array(df_units.loc[df_units["variable"] == "L_0"]["norma

    # And derive the others:
    # Time:
    t_0 = l_0/v_0

    # Pressure:
    p_0 = rho_0*v_0**2

    # Magnetic field:
    b_0 = v_0*np.sqrt(4*np.pi*rho_0)

    return rho_0, v_0, l_0, t_0, p_0, b_0
```

In [3]:
```python
# Path
units_file = "./data/Orszag_Tang-MHD/units.out"

# Call the function
den_units, vel_units, len_units, time_units, press_units, _ = units(ur
```

In [4]:
```
# To normalise the arrays:
def normalised_arrays(vtk_file, time_file):
    """
    Reads the data arrays, and returns 2D CGS-normalised arrays.
    Inputs: vtk_file -> variable storing the vtk file
    Outputs: rho_2d -> 2D density array
             prs_2d -> 2D pressure array
             vx1_2d -> 2D velocity array along x
             vx2_2d -> 2D velocity array along y
    Author: MAY
    """
    # A. SPATIAL QUANTITIES
    # Obtain the 2D data in code units:
    mesh = pv.read(vtk_file)

    # Get data arrays in code units:
    rho = pv.get_array(mesh, "rho", preference = 'cell')
    vx1 = pv.get_array(mesh, "vx1", preference = 'cell')
    vx2 = pv.get_array(mesh, "vx2", preference = 'cell')

    # Normalise them:
    rho_cgs = rho*den_units
    vx1_cgs = vx1*vel_units
    vx2_cgs = vx2*vel_units

    # Reshape them:
    rho_2d = rho_cgs.reshape(mesh.dimensions[0] - 1, mesh.dimensions[1
    vx1_2d = vx1_cgs.reshape(mesh.dimensions[0] - 1, mesh.dimensions[1
    vx2_2d = vx2_cgs.reshape(mesh.dimensions[0] - 1, mesh.dimensions[1

    # B. TIME
    # Read the file
    df_time = pd.read_csv(time_file, sep = "\s+", header = None)

    # Get the values we're interested in
    time_array = np.array(df_time.iloc[:,1])

    # And normalise it
    time = time_array*time_units

    return rho_2d, vx1_2d, vx2_2d, time, mesh
```

In [5]:
```
# We will need the mesh and time file. So
any_vtk_file = "./data/Orszag_Tang-MHD/data.0021.vtk"
time_file = "./data/Orszag_Tang-MHD/vtk.out"

# and call the function
_, _, _, time_cgs, mesh = normalised_arrays(any_vtk_file, time_file)
```

In [6]:
```python
# To get divergence of the velocity field:
def divergence(vx1_2d, vx2_2d):
    """
    Computes the divergence of the velocity field
    given the components along x and y.
    Inputs: vx1_2d, vx2_2d -> 2D velocity components
    Output: div -> 2D divergence
    Author: MAY.
    """
    # Spacing:
    x2 = len_units*np.linspace(mesh.bounds[0], mesh.bounds[1], mesh.d:
    y2 = len_units*np.linspace(mesh.bounds[0], mesh.bounds[1], mesh.d:
    dv = abs(x2[10]-y2[11])

    # Compute the derivatives of the data
    grad_x = np.array(np.gradient(vx1_2d, axis=1))
    grad_y = np.array(np.gradient(vx2_2d, axis=0))

    # And get the divergence
    div = grad_x + grad_y

    return div
```

In [7]:
```python
# To get the norm of the curl:
def curl_norm(vx1_2d, vx2_2d):
    """
    Computes the curl of the velocity field
    given the components along x and y.
    Inputs: vx1_2d, vx2_2d -> 2D velocity components
    Outputs: norm -> 2D norm of the velocity curl
             curlt -> 2D velocity curl
    Author: MAY.
    """
    # Spacing:
    x2 = len_units*np.linspace(mesh.bounds[0], mesh.bounds[1], mesh.d:
    y2 = len_units*np.linspace(mesh.bounds[0], mesh.bounds[1], mesh.d:
    dx = abs(x2[10]-y2[11])

    # Compute the derivatives
    curl1 = np.gradient(vx1_2d, dx, axis = 0) # axis = 0 is along y
    curl2 = np.gradient(vx2_2d, dx, axis = 1) # axis = 1 is along x
    curlt = curl2 - curl1

    # Norm
    norm = np.sqrt(curlt**2)

    return norm, curlt
```

In [8]:
```python
# And create the figures:
def figures(vtks, folder, time_array, boolean):
    """
    Runs over all the vtks files in a folder, get the pressure and
    velocity arrays, and computes the divergence and curl of the veloc
    Finally, saves maps of all of them in a folder.
    Inputs: vtks -> string storing all the vtks files
            folder -> name of the folder
            time_array -> normalized time array
            boolean -> True or False to show the plots
    Outputs: None.
    Author: MAY.
    """
    # Folder for the images:
    if os.path.isdir(folder):
        print("The folder already exists.")
    else:
        os.mkdir(folder)

    # Create the normalized grid:
    x = len_units*np.linspace(mesh.bounds[0], mesh.bounds[1], (mesh.d
    y = len_units*np.linspace(mesh.bounds[2], mesh.bounds[3], (mesh.d

    # Meshgrid:
    x_2d, y_2d = np.meshgrid(x, y)

    for i in range(0, len(time_array)):
        # Call the function to normalise the arrays:
        rho_2d, vx1_2d, vx2_2d, _, _  = normalised_arrays(vtks.format(

        # Get the divergence
        div_vel = divergence(vx1_2d, vx2_2d)

        # Get the norm of the curl
        norma_del_curl, _ = curl_norm(vx1_2d, vx2_2d)

        # Density plots:
        fig1, ax = plt.subplots()
        dens = ax.pcolor(x_2d, y_2d, rho_2d, cmap = "magma", vmin=0, v
        fig1.colorbar(dens, label = r"$Density\, magnitude\, [g/cm^3]$
        ax.set(title = f'Density at t = {time_array[i]:.2e} seconds',
        if boolean:
            plt.show()
        plt.close()

        # Divergence plots:
        fig2, ax = plt.subplots()
        diver = ax.pcolor(x_2d, y_2d, div_vel, cmap = "inferno", vmin=
        fig2.colorbar(diver, label = r"$Magnitude\,of\,the\,divergence
        ax.set(title = f'Divergence at t = {time_array[i]:.2e} seconds

        if boolean:
            plt.show()
        plt.close()

        # Norm of curl plots:
        fig3, ax = plt.subplots()
        curl = ax.pcolor(x_2d, y_2d, norma_del_curl, cmap = "magma_r",
        fig3.colorbar(curl, label = r"$Magnitude\,of\,the\,curl\,of\,v
        ax.set(title = f'Norm of the curl at \n t = {time_array[i]:.2e
        if boolean:
            plt.show()
```

```
        plt.close()

        # Save the figures:
        fig1.savefig(os.path.join(folder, "dens_figure{:03d}.png".forr
        fig2.savefig(os.path.join(folder, "dive_figure{:03d}.png".forr
        fig3.savefig(os.path.join(folder, "curl_figure{:03d}.png".forr
```

In [9]:
```
# Paths and names:
vtks = "./data/Orszag_Tang-MHD//data.0{:03d}.vtk"
folder_1 = "./output_n"

# Call the function
figures(vtks, folder_1, time_cgs, False)
```

To visualize them:

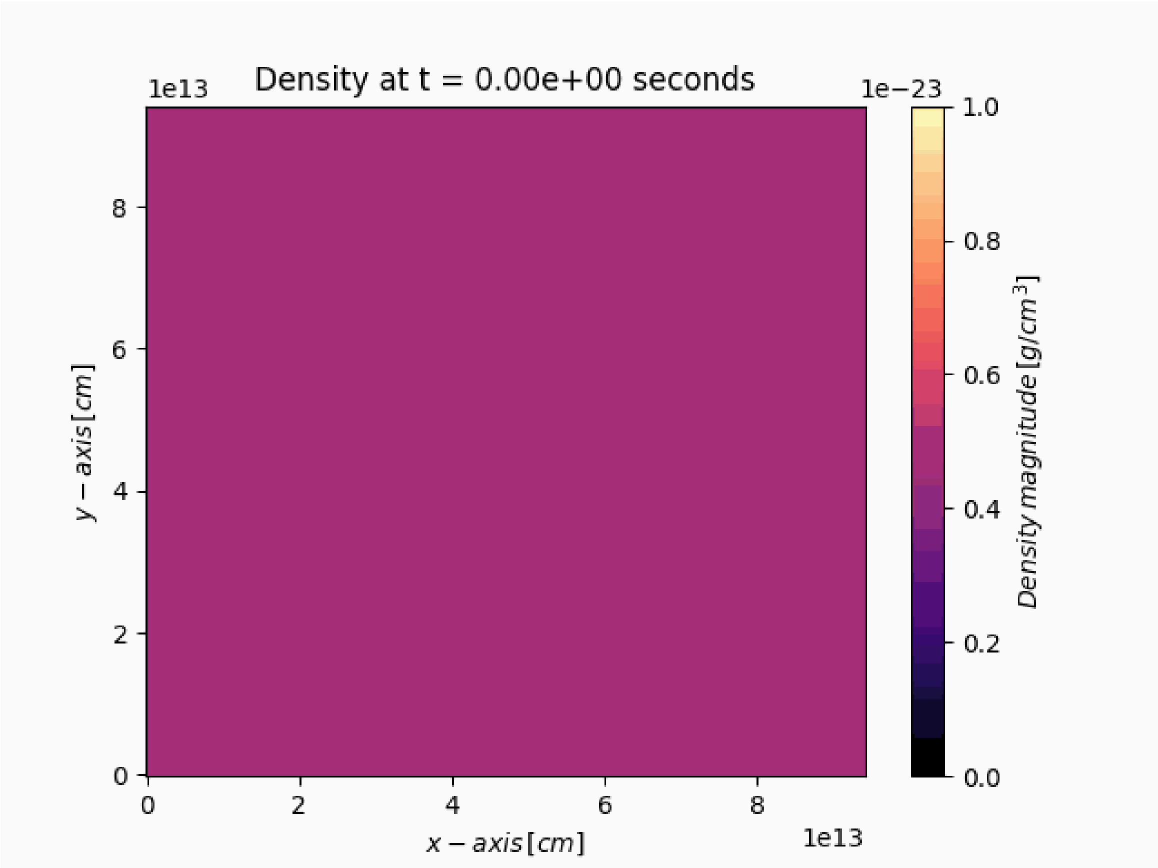
In [10]:
```
# Importing libraries
from PIL import Image, ImageDraw
from IPython import display
import glob
```

In [11]:
```
def movies(images_input, imgif_output):
    """
    Creates movies for a simulation showing the
    time evolution of maps (PNGs) and attaches to
    them the corresponding value of an array.
    Inputs: images_input -> str containing ALL the maps (***)
            imgif_output -> str with the name of the resulting movie
    No outputs. The function itself.
    Author: MAY.
    Date: 25/04/2024
    """
    # Get the images.
    # Define an empty list:
    images = []

    # The loop.
    for i in sorted(glob.glob(images_input)):

        # Getting the images:
        img = Image.open(i)

        # And append all the new images to the empty list.
        images.append(img)

    # Finally saving them in a gift:

    images[0].save(fp = imgif_output, format = "GIF", append_images =
                   save_all = True, duration = 150, loop =0)
```

In [12]:
```
# Paths and names
all_images_density = "./output_n/dens_figure***.png"
gif_density = "./output_n/dens_figure.gif"

all_images_diver = "./output_n/dive_figure***.png"
gif_diver = "./output_n/dive_figure.gif"

all_images_curl = "./output_n/curl_figure***.png"
gif_curl = "./output_n/curl_figure.gif"

# Call the function for each
movies(all_images_density, gif_density)
movies(all_images_diver, gif_diver)
movies(all_images_curl, gif_curl)
```

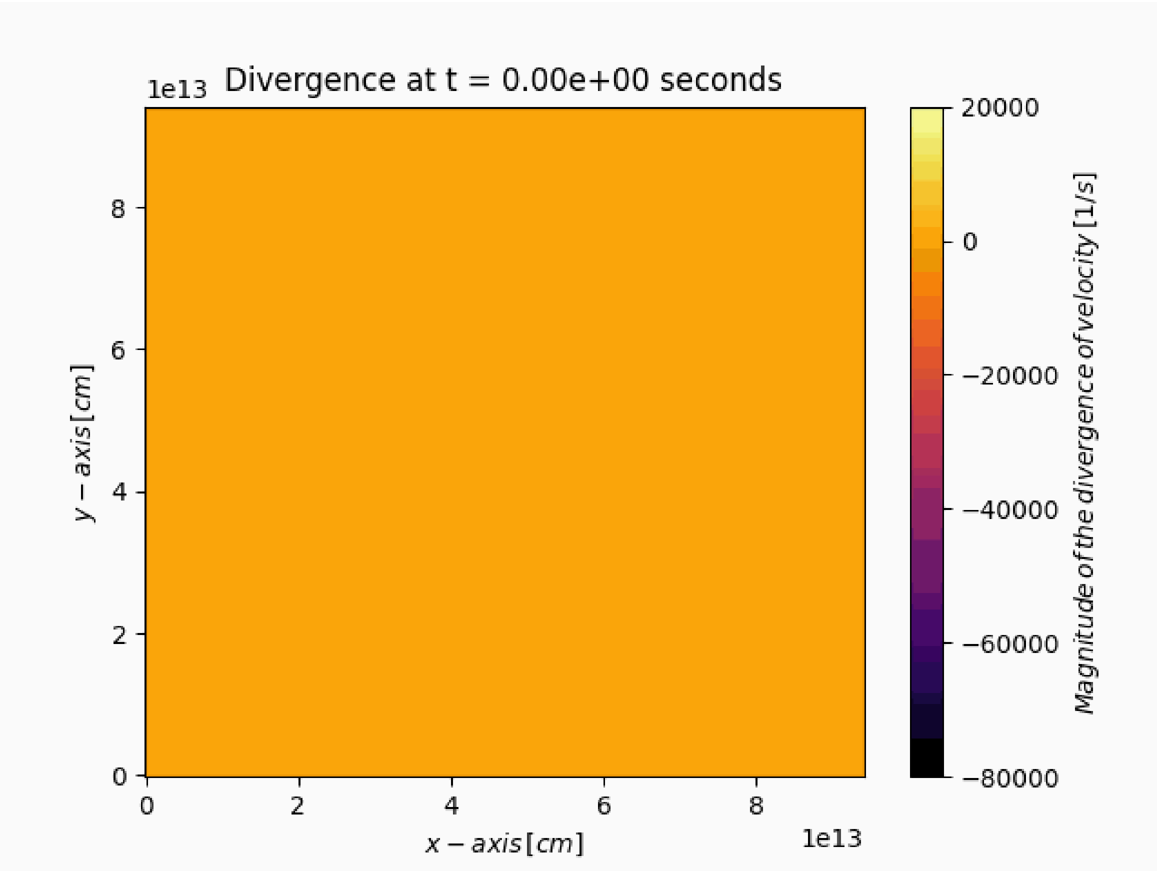
In [149]:
```
# The results are the following.
# Density:
display.Image(open(gif_density,'rb').read())
```

Out[149]:



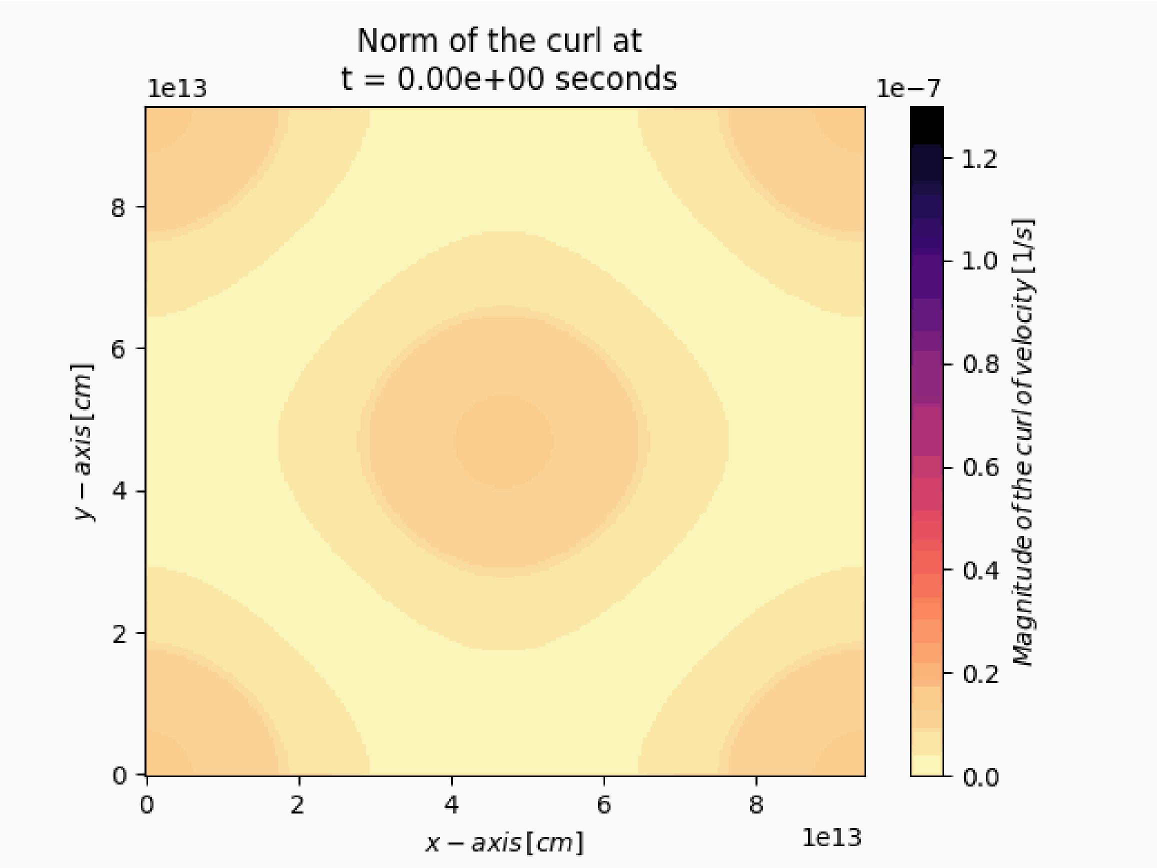
In [150]:
```python
# Divergence:
display.Image(open(gif_diver,'rb').read())
```

Out[150]:



In [151]:
```python
# Curl:
display.Image(open(gif_curl,'rb').read())
```

Out[151]:



(b) Create a set of Python functions that loops over all the simulation VTK files, computes the flow circulation, $\Gamma = \iint_S \vec{\nabla} \times \vec{v} \cdot \mathrm{d}\mathbf{S}$, and returns:

- a CSV file with 2 columns (time and flow circulation).
- a figure of the flow circulation versus time.

The units of flow circulation should be dS->L^2

$$\nabla \times \mathbf{v} \cdot \mathrm{d}\mathbf{S} = \left[\frac{1}{\mathrm{L}}\right] \left[\frac{\mathrm{L}}{\mathrm{T}}\right] [\mathrm{L}] = \left[\frac{\mathrm{L}}{\mathrm{T}}\right],$$

that is, $[\mathrm{cm/s}]$. Additionally, as in the previous problems, we assume $dx = dy$.

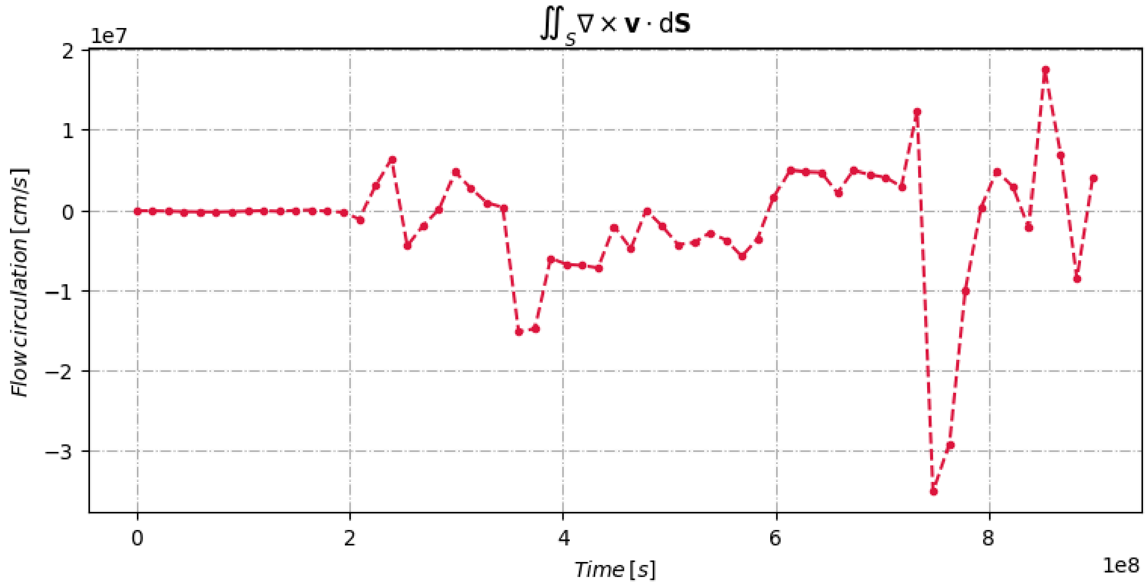
In [16]:
```python
# To calculate the flow circulation
def flow_circulation(curlt):
    """
    Computes the flow circulation given the velocity curl.
    Input: curlt -> 2D data with the velocity curl
    Output: integral -> flow circulation
    Author: MAY.
    """
    # Spacing:
    x2 = len_units*np.linspace(mesh.bounds[0], mesh.bounds[1], mesh.d
    y2 = len_units*np.linspace(mesh.bounds[0], mesh.bounds[1], mesh.d
    dx = abs(x2[10]-y2[11])

    # 2D array with dx
    ds = dx*np.ones((curlt.shape[0], curlt.shape[0]))

    # Dot product between the curl and differential
    dot_product = np.dot(curlt, ds)

    # And the double integral
    integral = np.sum(dot_product)

    return integral
```

In [17]:
```python
# The loop over all the files
def flow_of_all(vtks, name_csv, time_file, boolean):
    """
    Runs over all the vtks files and computes the flow circulation
    of each. Returns a csv file with tww columns: time and flow circ.
    and a plot.
    Inputs: vtks -> string storing all the vtk files
            name_csv -> name of the csv
            time_file -> string storing the vtk.out file
            boolean -> True or False to show the plots
    Outputs: df -> csv file
             fig -> plot of the flow vs. time
    Author: MAY.
    """
    # List to store the flow circulation values
    flow_cir_list = []

    # The loop
    for i in range(0, len(time_cgs)):
        # Get all the arrays:
        _, vx1_2d, vx2_2d, _, _ = normalised_arrays(vtks.format(i), t

        # Compute the curl of each
        _, curl = curl_norm(vx1_2d, vx2_2d)

        # Get the flow circulation of each and append the values:
        flow_circ = flow_circulation(curl)
        flow_cir_list.append(flow_circ)

    # Get the result into an array:
    flow_circ_array = np.array(flow_cir_list)

    # Data frame:
    df = pd.DataFrame({"Time [s]": time_cgs, "Flow circulation [cm/s]"

    # Save the data frame:
    df.to_csv(f"{name_csv}", sep=',', float_format='{:.4e}'.format, in

    # Plot:
    fig, ax = plt.subplots(figsize = (9,4))
    ax.plot(time_cgs, flow_circ_array,linestyle = "--", marker =".", c
    ax.set(xlabel = r"$Time\,[s]$", ylabel = r"$Flow\,circulation\,[cm
          title = r"$\iint_S \nabla\times \mathbf{v} \cdot \mathrm{d}\
    plt.grid(linestyle='-.')
    if boolean:
        plt.show()
    plt.close()

    return df, fig
```

In [18]:
```python
# Name of the csv
csv_name = "./flow_circulation.csv"

# Call the function
csv_file, figure = flow_of_all(vtks, csv_name, time_file, False)
```

In [19]:
```
# The result is
figure
```

Out[19]:



Fourier analysis:

(c) Create a set of Python functions that Fourier transforms the density of any VTK file in 2D, applies high-pass and low-pass filters, smooths the density map using a 2D Gaussian, and sequentially prints the following figures into a folder called "output_f" for all times:

- the 2D Fourier image of the density
- the high-pass filter map
- the low-pass filter map
- the Gaussian-blurred map

In [20]:
```
# To get the Fourier transform
def fourier_transform_2d(rho_2d):
    """
    Computes the FT of a 2D array.
    Input: rho_2d -> 2D density values
    Outputs: norm_fourier_rho_2d -> norm of the FT of the density arra
             shifted_fourier -> shifted FT of the density array
    Author: MAY.
    """
    # We'll use a set of numpy functions:
    fourier_rho_2d = np.fft.fft2(rho_2d)
    shifted_fourier = np.fft.fftshift(fourier_rho_2d)
    norm_fourier_rho_2d = np.abs(shifted_fourier)

    return norm_fourier_rho_2d, shifted_fourier
```

In [21]:
```python
# The low-pass filter
def low_high_filters(shifted_array):
    """
    Applies high-pass and low-pass filters to the FT density
    in the Fourier space, then performs the inverse of the result.
    Input: shifted_array -> shifted FT of the density array
    Outputs: low_pass_image -> result from filtering the high frequenc
             high_pass_image -> result from fitering the low frequenci
    Author: MAY.
    """
    # We need a mask.
    # Define the center and radius:
    center = [shifted_array.shape[0]//2, shifted_array.shape[1]//2]
    radius = 15

    # Then construct the mask
    mask = Image.new(mode="RGB", size=(256, 256))
    draw = ImageDraw.Draw(mask)
    draw.ellipse((center[0]-radius, center[1]-radius, center[0]+radius
                 fill=(255, 0, 0), outline=(0, 0, 0))

    # And convert it to binary.
    mask_low = np.array(mask)[:,:,0]//255
    mask_high = -mask_low + 1

    # Now multiply it by the shifted array and get the IFFT:
    low_pass_image = np.fft.ifft2(np.fft.ifftshift(shifted_array*mask_
    high_pass_image = np.fft.ifft2(np.fft.ifftshift(shifted_array*mask

    return low_pass_image, high_pass_image
```

In [22]:
```
# The Gaussian mask
def gaussian_mask(shifted_array, sigma_x, sigma_y):
    """
    Smooths the density map using a 2D Gaussian in the Fourier space.
    Inputs: shifted_array -> shifted FT of the density array
            sigma_x, sigma_y -> sigmas along each direction
    Outputs: inv_in_xy -> result (real space) from carrying out
                          Gaussian smoothing.
    Author: MAY.
    """
    def gaussian(x, y, sigma_x, sigma_y):
        """
        Function to get a 2D gaussian to be used as mask.
        Inputs: x, y -> x and y-components of the grid
                sigma_x, sigma_y -> sigmas along each direction
        Outputs: The gaussian.
        """
        return (1/(2*np.pi*sigma_x*sigma_y)*np.exp(-(x**2/(2*sigma_x*

    # Generate the vectors for x and y
    x = np.linspace(-10, 10, (mesh.dimensions[0] - 1))
    y = np.linspace(-10, 10, (mesh.dimensions[0] - 1))

    # Create the meshgrid
    x_2d, y_2d = np.meshgrid(x, y)

    # Use the previous function to get the gaussian
    gauss = gaussian(x_2d, y_2d, sigma_x, sigma_y)

    # Fourier transform it
    fourier_gauss = np.fft.fftshift(np.fft.fft2(gauss))

    # Multiply it by the shifted array and get the inverse FT:
    gaussian_masked_image = np.fft.ifft2(np.fft.ifftshift(fourier_gaus

    # And roll the image
    inv_in_x = np.roll(gaussian_masked_image.real, gaussian_masked_ima
    inv_in_xy = np.roll(inv_in_x, gaussian_masked_image.shape[0]//2, a

    return inv_in_xy
```

In [23]:
```python
# The loop function:
def fourier_loop(folder, vtks, time_file, boolean):
    """
    Runs over all vtk files and applies high-pass, low-pass filters,
    and Gaussian smoothing to the 2D density array in the Fourier spac
    Then, saves the maps as figures.
    Inputs: folder -> name of the folder to save the maps to.
            vtks -> string storing all the vtk files
            time_file -> string with the time file
            boolean -> True or False to show the maps
    Outputs: None. The images are saved.
    Author: MAY.
    """
    # Folder for the images:
    if os.path.isdir(folder):
        print("The folder already exists.")
    else:
        os.mkdir(folder)

    # Grid in cm^{-1}
    x1 = np.linspace(mesh.bounds[0], mesh.bounds[1], (mesh.dimensions
    y1 = np.linspace(mesh.bounds[2], mesh.bounds[3], (mesh.dimensions

    # Meshgrid in cm^{-1}:
    x1_2d, y1_2d = np.meshgrid(x1, y1)

    # Grid in cm
    x2 = x1*len_units**2
    y2 = y1*len_units**2

    # Meshgrid in cm:
    x2_2d, y2_2d = np.meshgrid(x2, y2)

    # Sigmas:
    sigma_x = 0.3
    sigma_y = 0.3

    # The loop:
    for i in range(0, len(time_cgs)):
        # Get all the density arrays:
        rho_2d, _, _, _, _ = normalised_arrays(vtks.format(i), time_f

        # Compute the Fourier transform:
        norm, shifted_arr = fourier_transform_2d(rho_2d)

        # Apply the low & high filters:
        low, high = low_high_filters(shifted_arr)

        # Gaussian smoothing:
        gauss_image = gaussian_mask(shifted_arr, sigma_x, sigma_y)

        # Plots:

        # FT of density
        fig1, ax = plt.subplots()
        dens = ax.pcolor(x1_2d, y1_2d, np.log10(norm, out=np.zeros_lik
                         cmap = "magma_r", vmin=-28, vmax = -18)
        fig1.colorbar(dens, label = r"$\log_{10}(\mathcal{FT}\,[Densit
        ax.set(title = f'Fourier Transform of density at \n t = {time_
               xlabel = r"$x-axis\, [cm^{-1}]$", ylabel = r"$y-axis\,
        if boolean:
```

```
            plt.show()
        plt.close()

        # Low-pass filter result
        fig2, ax = plt.subplots()
        dens = ax.pcolor(x2_2d, y2_2d, low.real, cmap = "magma", vmin=
        fig2.colorbar(dens, label = r"$Density\, magnitude\, [g/cm^3]$
        ax.set(title = f'Low-pass filter of density at \n t = {time_cc
               xlabel = r"$x-axis\, [cm]$", ylabel = r"$y-axis\, [cm]$
        if boolean:
            plt.show()
        plt.close()

        # High-pass filter
        fig3, ax = plt.subplots()
        dens = ax.pcolor(x2_2d, y2_2d, high.real, cmap = "magma", vmin
        fig3.colorbar(dens, label = r"$Density\, magnitude\, [g/cm^3]$
        ax.set(title = f'High-pass filter of density at \n t = {time_c
               xlabel = r"$x-axis\, [cm]$", ylabel = r"$y-axis\, [cm]$
        if boolean:
            plt.show()
        plt.close()

        # Gaussian smoothing
        fig4, ax = plt.subplots()
        dens = ax.pcolor(x2_2d, y2_2d, gauss_image.real,  cmap = "magn
        fig4.colorbar(dens, label = r"$Density\, magnitude\, [g/cm^3]$
        ax.set(title = f'Gaussian smoothing of density at \n t = {time
               xlabel = r"$x-axis\, [cm]$", ylabel = r"$y-axis\, [cm]$
        if boolean:
            plt.show()
        plt.close()

        # Save the maps:
        fig1.savefig(os.path.join(folder, "ft_dens_figure{:03d}.png".f
        fig2.savefig(os.path.join(folder, "low_pass_dens_figure{:03d}.
        fig3.savefig(os.path.join(folder, "high_pass_dens_figure{:03d}
        fig4.savefig(os.path.join(folder, "gaussian_dens_figure{:03d}.
```

In [24]:

```
# Folder
folder = "./output_f"

# Call the function
fourier_loop(folder, vtks, time_file, False)
```

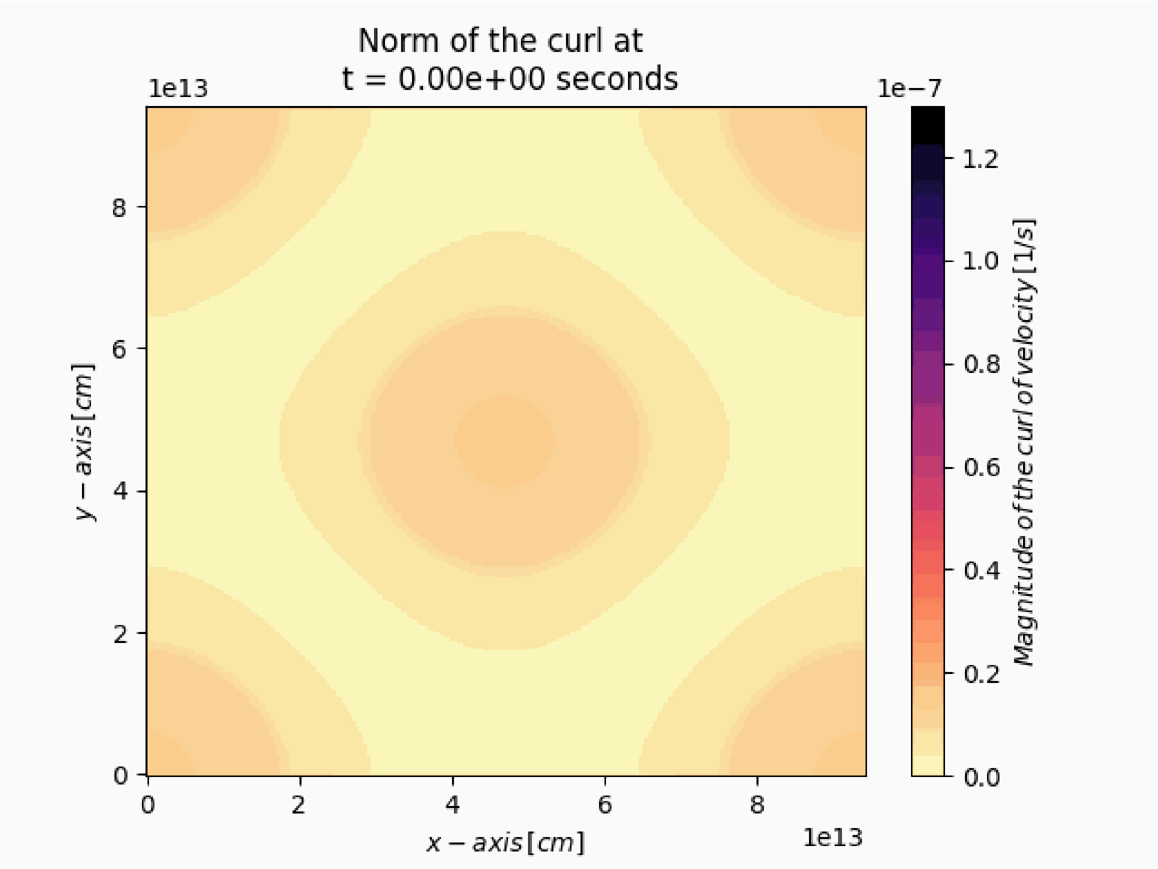
(d) Create a Python function that reads in the images you wrote, and returns movies showing the time evolution of the velocity curl maps computed in (a), the high-pass filter maps in (c), and the flow circulation in (b).

We simply use the function defined above to create the movies. The flow circulation was shown before.

In [25]:
```
# Paths and names
all_images_high_pass = "./output_f/high_pass_dens_figure***.png"
gif_high_pass = "./output_f/high_pass_dens_figure.gif"

all_images_curl = "./output_n/curl_figure***.png"
gif_curl = "./output_n/curl_figure.gif"

# Call the function for each
movies(all_images_high_pass, gif_high_pass)
movies(all_images_curl, gif_curl)
```

In [147]:
```
# Norm of curl
display.Image(open(gif_curl,'rb').read())
```

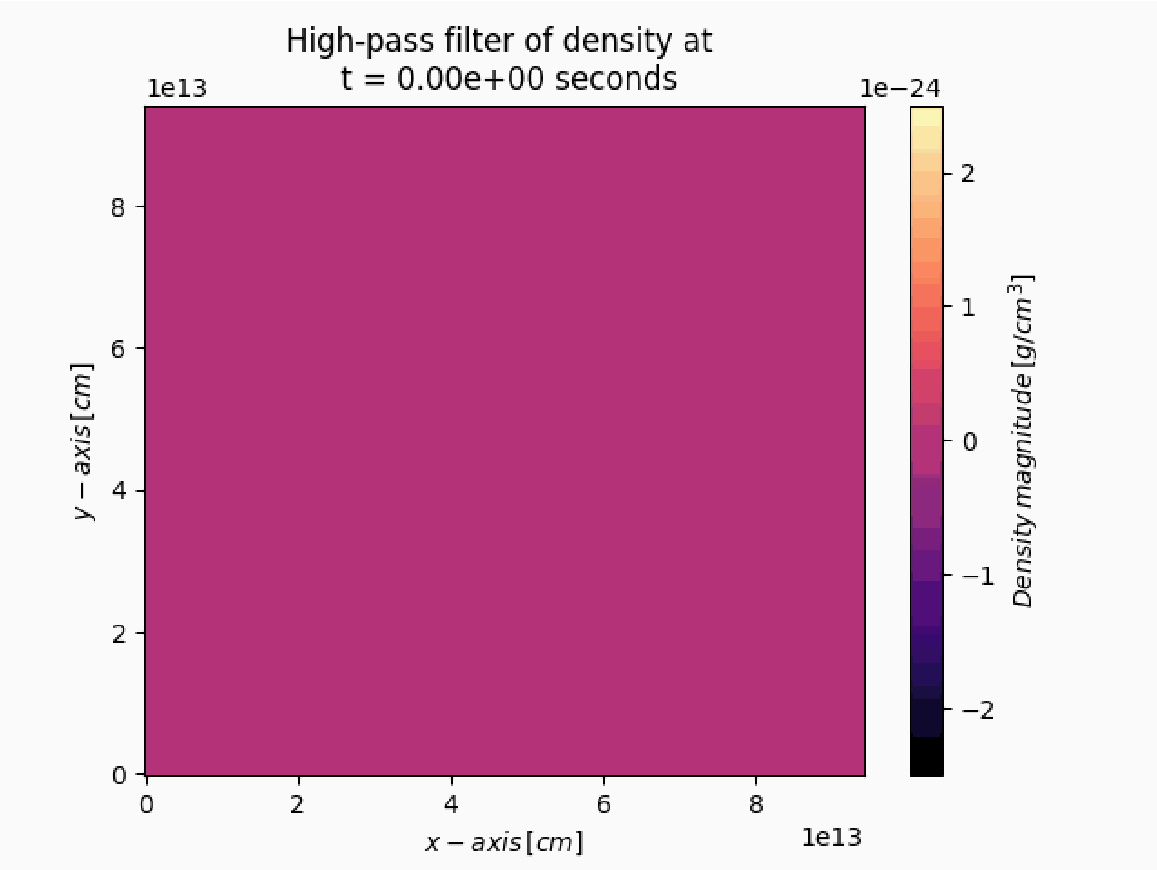
Out[147]:



In [148]:

```
# High-pass filtered:
display.Image(open(gif_high_pass,'rb').read())
```

Out[148]:

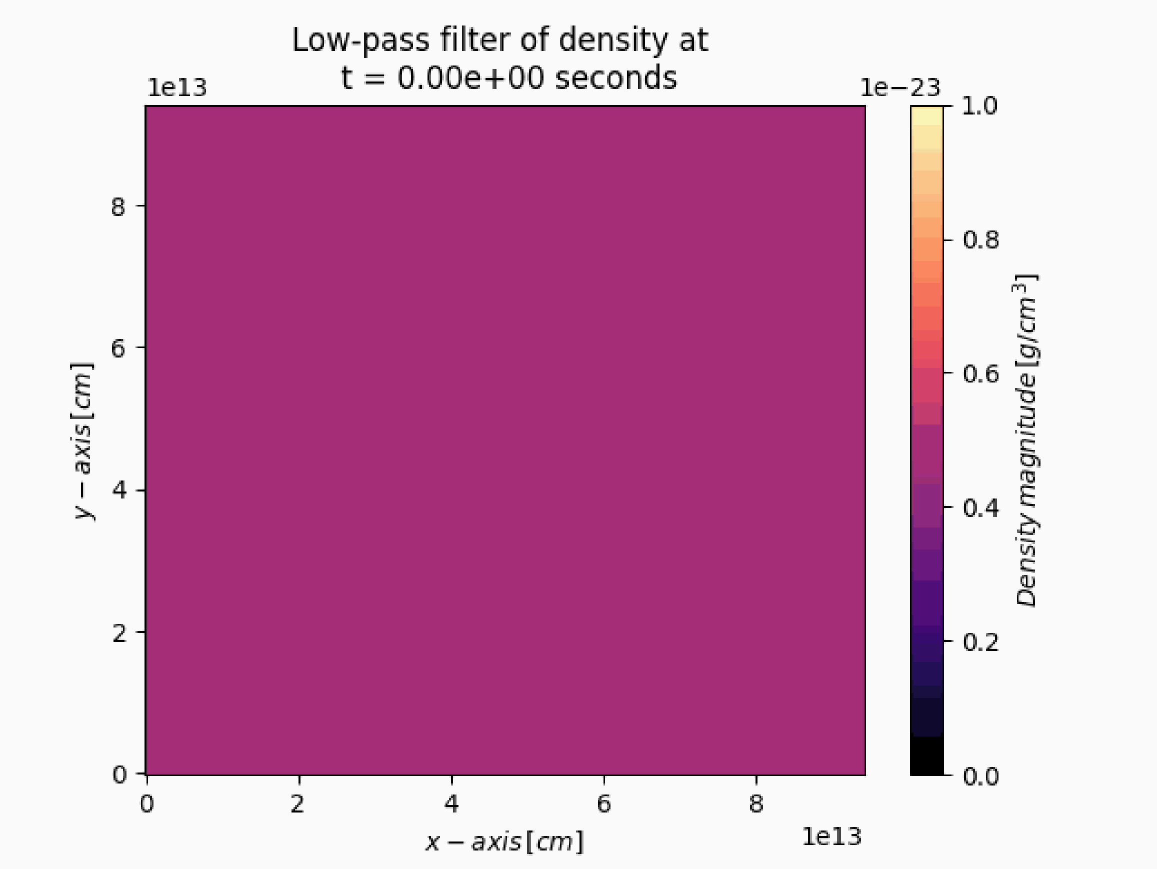


In [28]:

```
# Paths and names we are not asked:
all_images_low_pass = "./output_f/low_pass_dens_figure***.png"
gif_low_pass = "./output_f/low_pass_dens_figure.gif"

all_images_gaussian = "./output_f/gaussian_dens_figure***.png"
gif_gaussian = "./output_f/gaussian_dens_figure.gif"

# Call the function for each
movies(all_images_low_pass, gif_low_pass)
movies(all_images_gaussian, gif_gaussian)
```

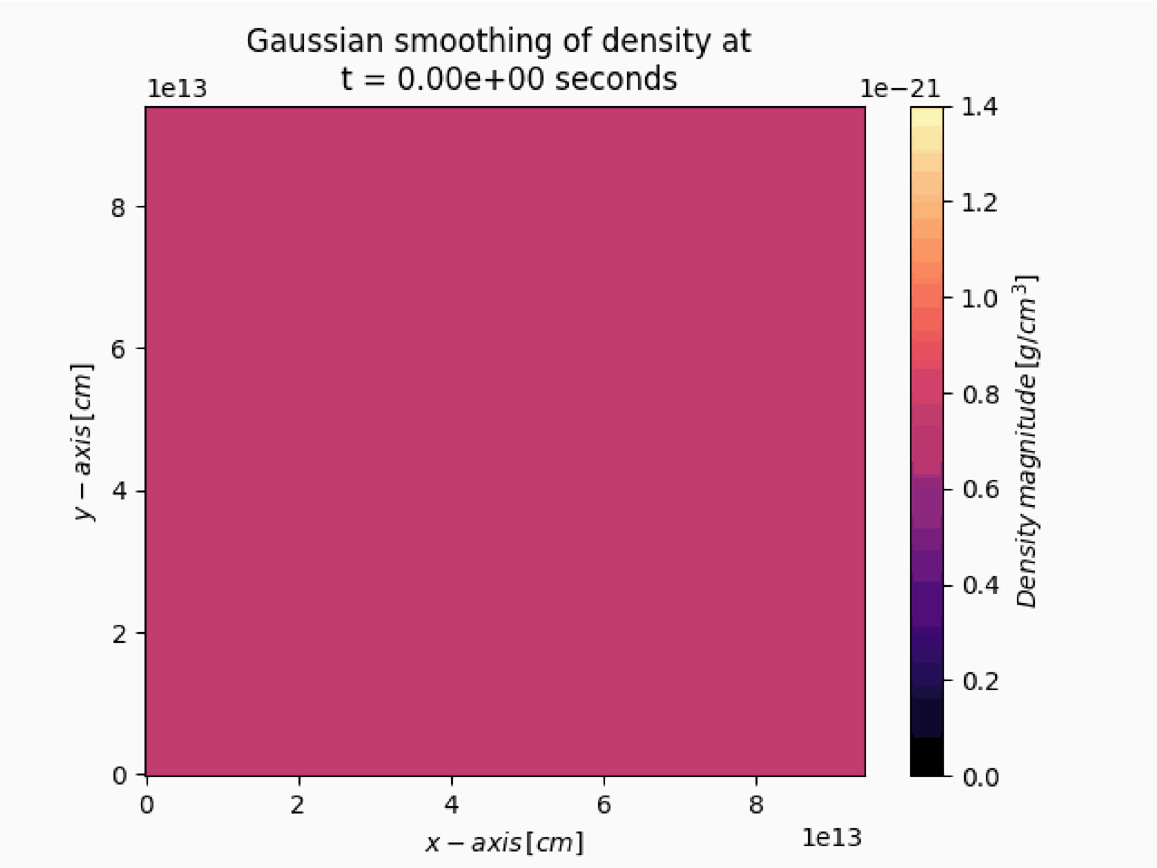
In [145]:
```python
# Low-pass filter:
display.Image(open(gif_low_pass,'rb').read())
```

Out[145]:



In [146]:
```python
# Gaussian smoothing:
display.Image(open(gif_gaussian,'rb').read())
```

Out[146]:



Interpretation:

(e) Based on your analyses above, briefly answer the following questions:

- What information do the velocity divergence and velocity curl provide about the flow?

The velocity divergence provides information about the location of supersonic (and not) regions, that is, where $\nabla \cdot \mathbf{v} < 0$. The velocity curl (vorticity) tells us how much *spinning* there is at each point in the region (the local spinning).

- Does the flow circulation reach steady state?

According to the plot found, it doesn't. Moreover, although it does seem to start at steady state, after some time, it appears to get in an state that works in the opposite way: it changes rapidly from time to time.

- What do the high-pass and low-pass filter maps show?

The high-pass filter let us appreciate the *small-scale structure* of the image (the edges of the image), while the low-pass one shows the opposite, the *large-scale structure* (the colors and general shape of the image).

- Are the low-pass and Gaussing blurring results consistent with one another?

They are very similar. I assume some parameters could be adjusted to make them look almost identical, but, nonetheless, the answer is yes, they are consistent with one another.

7. Understanding selection bias: Monte Carlo simulations (10 points)

Supernovae type Ia (SN Ia) are very energetic astronomical explosions, which have a very similar intrinsic known brightness (i.e. they have a very similar absolute magnitude M), so they can be used as "standard candles" to measure the luminosity distance, d, as a function of redshift, z:

$$d = \frac{cz}{H_0}$$

where c is the speed of light and H_0 is the Hubble constant. Since they have similar absolute magnitudes M, we can estimate distances by comparing how bright or faint they appear on the sky as indicated by the measured apparent magnitude, m, which does differ:

$$m = M + 5\log\left(\frac{d}{\mathrm{Mpc}}\right) + 25$$

Higher m values imply objects are fainter; lower m values imply objects are brighter. Same for M. Unfortunately, selection effects associated with instrumental limitations can bias our measurements. For example, far-away SN Ia can be so faint that they may not be detectable, so the sample will be biased towards brighter objects.

Therefore, to understand selection bias, we want to simulate this effect using a Monte Carlo simulation.

The purpose of this problem is to determine the bias as a function of redshift for a sample of objects (SN Ia) via a Monte Carlo calculation. To set up your simulation, assume that:

- $H_0 = 70\,\mathrm{km\,s^{-1}\,Mpc^{-1}}$
- the absolute magnitude of SN Ia $M = -19.5\,\mathrm{mag}$.
- your supernova search will be able to detect 100% of objects as faint as

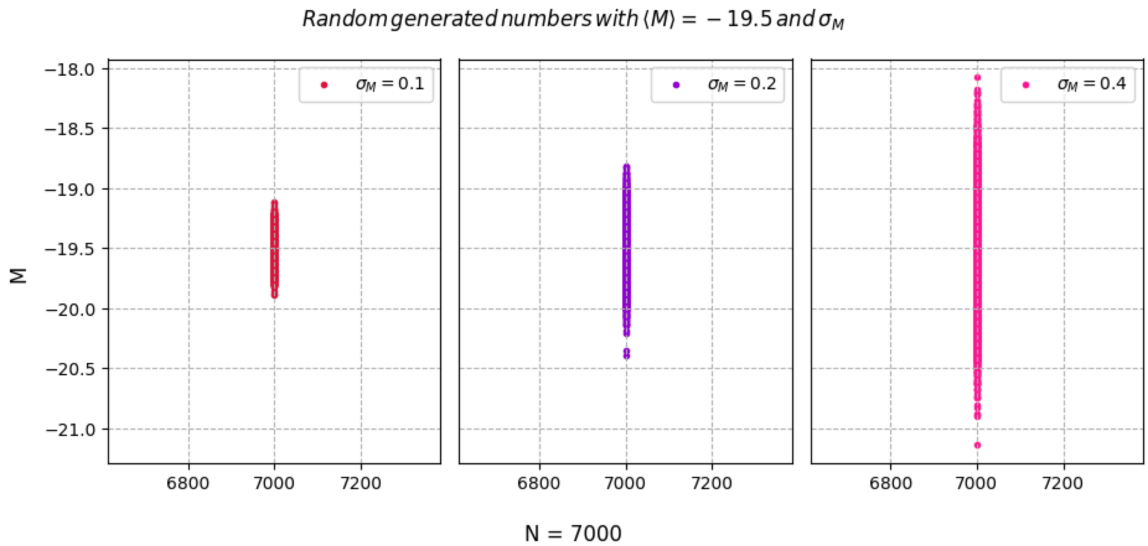
(a) Write a python function to generate N Gaussian random variables with mean $\langle M \rangle = -19.5\,\mathrm{mag}$ and different standard deviations ($\sigma_M = 0.1, 0.2,$ and $0.4\,\mathrm{mag}$). Make 3 plots of M versus N, where N is the number of generated objects, one for each σ_M.

In [31]:
```
# Function
def gaussian_random_numbers(M, sigma, n):
    """
    Generates random numbers normally distributed.
    Inputs: M -> mean of the distribution
            sigma -> sigma of the distribution
            n -> number of numbers to be generated
    Outputs: random_number -> the random numbers
    Author: MAY.
    """
    # Use a numpy function
    random_numbers = np.random.normal(M, sigma, n)

    return random_numbers
```

In [152]:
```
# Define parameters
M = -19.5
n = 7000
sigma_1 = 0.1
sigma_2 = 0.2
sigma_3 = 0.4

# And call the function for each
numbers_1 = gaussian_random_numbers(M, sigma_1, n)
numbers_2 = gaussian_random_numbers(M, sigma_2, n)
numbers_3 = gaussian_random_numbers(M, sigma_3, n)
```

We think it is better if the values are on the line $x = n$, as it makes visualization simpler.

In [153]:
```
# Plotting
fig, axs = plt.subplots(1, 3, figsize = (9.5,4.5), sharey=True)
fig.suptitle(r"$Random\, generated\, numbers\,with\,\langle M\rangle=-
fig.supxlabel(f'N = {n}')
fig.supylabel(f'M')

axs[0].scatter(n*np.ones(n), numbers_1, marker = ".",
               color = "crimson", label = r"$\sigma_M = 0.1$")
#axs[0].set_ylim(-20.5,-18.5)
axs[0].grid(linestyle = "--")
axs[0].legend()

axs[1].scatter(n*np.ones(n), numbers_2, marker = ".",
               color = "darkviolet", label = r"$\sigma_M = 0.2$")
axs[1].grid(linestyle = "--")
axs[1].legend()

axs[2].scatter(n*np.ones(n), numbers_3, marker = ".",
               color = "deeppink", label = r"$\sigma_M = 0.4$")
axs[2].grid(linestyle = "--")
axs[2].legend()

# To get rid of the little horizontal lines in between
for i in axs[1:]:
    i.tick_params(axis='y', length=0)

plt.tight_layout()
plt.show()
```



Expand plot to see M vs. N
-0.25

(b) Write a python function to calculate and return:

- the luminosity distances, d, in Mpc given redshifts between $z = 0$ and $z = 0.1$.
- the apparent magnitudes, m, for the same redshift range.

In [154]:
```
# Define the constants
c = 3*10**6 # km/s
H_0 = 70 # Mpc km/s
```

The set of random numbers will be used to define the z-vector from $z = 0$ to $z = 0.1$.

In [155]:
```
# Function
def d_m(M, sigma, n):
    """
    Given a set of random numbers, creates a vector of values from 0 t
    and computes the luminosity distance, d, and the apparent magnituc
    Inputs: M -> mean of the distribution
            sigma -> sigma of the distribution
            n -> number of numbers to be generated
    Ouputs: d -> luminosity distance
            m -> apparent magnitudes
            z -> random vector from 0 to 0.1
    Author: MAY.
    """
    # Use the previous function to generate random numbers
    numbers = gaussian_random_numbers(M, sigma, n)

    # Construct the z-vector to go from 0 to 0.1 as
    z = 0.1*(numbers - np.min(numbers))/(np.max(numbers) - np.min(numb

    # The luminosity distance reads
    d = (c/H_0)*z # Mpc

    # The apparent magnitude is
    m = M + 5*np.log10(d) + 25 # mag

    return d, m, z
```

(c) Write a python function that:

- reads the resulting m values from item (b),
- removes values with apparent magnitudes larger than the detection threshold $m = 18.5$ mag,
- re-calculates the mean observed magnitude $\langle M_{\text{observed}} \rangle$ of the SN Ia from the actually detected objects for the same redshift range.
- returns the bias as a function of redshift. The bias in M can be calculated with:

$$|\Delta M| = |\langle M_{\text{observed}} \rangle - \langle M \rangle|$$

In [162]:

```
# Function
def bias(M, sigma, n):
    """
    Gets the bias as a function of redshift, z. It uses the
    function defined above.
    Inputs: M -> mean of the distribution
            sigma -> sigma of the distribution
            n -> number of numbers to be generated
    Outputs: bias -> bias function
             M_observed -> mean observed magnitude
    Author: MAY.
    """
    # Use the previous function to get d and m
    d, m, z = d_m(M, sigma, n)

    # Remove larger values than m = 18.5 mag with nan's
    clean_m = np.where(m <= 18.5, m, np.nan)

    # Recalculate the mean observed magnitude for the same redshift ra
    # (d is defined from z)
    M_observed = clean_m - 5*np.log10(d) - 25

    # Obtain the bias as a function of d, which is a function of z
    bias = abs(M_observed-M)

    return bias, M_observed
```

This should take the d, m and z as an input.

✓

✓

✓

✓

(d) Make 3 plots of m versus N, where N is the number of generated objects, one for each $\sigma_M = 0.1, 0.2$, and 0.4 mag, showing the detection threshold and colouring distinctly the objects that would not be detected.

In [186]:

```
# Call the function to get the m arrays
_, m_1, z_1 = d_m(M, sigma_1, n)
_, m_2, z_2 = d_m(M, sigma_2, n)
_, m_3, z_3 = d_m(M, sigma_3, n)
```

✓

```
C:\Users\DELL\AppData\Local\Temp\ipykernel_12720\1373418499.py:24: R
untimeWarning: divide by zero encountered in log10
  m = M + 5*np.log10(d) + 25 # mag
```

In [187]:

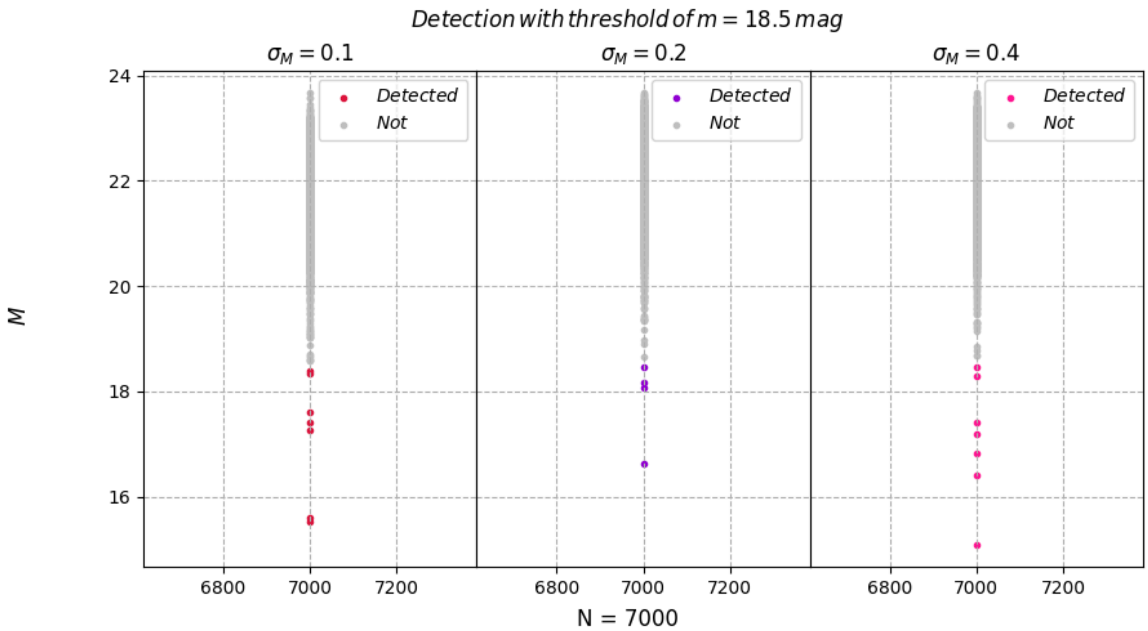
```
# Define the threshold
threshold = 18.5

# And apply it to each m array in order to separate the data
m_1_not_detected = m_1 > threshold
m_2_not_detected = m_2 > threshold
m_3_not_detected = m_3 > threshold
print(m_1)
```

Remove print

```
[22.3683684  22.70021723 22.01185003 ... 21.03024112 22.22182458
 20.30515155]
```

In [188]:

```python
# Plotting
fig, axs = plt.subplots(1, 3, figsize = (10,5), sharey=True)
fig.suptitle(r"$Detection\,with\,threshold\,of\,m=18.5\, mag$")
fig.supxlabel(f"N = {n}")
fig.supylabel(r"$M$")

axs[0].scatter(n*np.ones(n)[~m_1_not_detected], m_1[~m_1_not_detected]
axs[0].scatter(n*np.ones(n)[m_1_not_detected], m_1[m_1_not_detected],
axs[0].grid(linestyle = "--")
axs[0].set_title(r"$\sigma_M = 0.1$")
axs[0].legend()

axs[1].scatter(n*np.ones(n)[~m_2_not_detected], m_2[~m_2_not_detected]
axs[1].scatter(n*np.ones(n)[m_2_not_detected], m_2[m_2_not_detected],
axs[1].grid(linestyle = "--")
axs[1].set_title(r"$\sigma_M = 0.2$")
axs[1].legend()

axs[2].scatter(n*np.ones(n)[~m_3_not_detected], m_3[~m_3_not_detected]
axs[2].scatter(n*np.ones(n)[m_3_not_detected], m_3[m_3_not_detected],
axs[2].grid(linestyle = "--")
axs[2].set_title(r"$\sigma_M = 0.4$")
axs[2].legend()

# To remove the horizontal space between the plots
plt.subplots_adjust(wspace=0)

# And to get rid of the little horizontal lines in between
for i in axs[1:]:
    i.tick_params(axis='y', length=0)

plt.show()
```



The scatter here seems smaller than for 0.1?

Very few very few are detected in each case. Check m calculation.
-0.25

(e) Make 3 plots of $|\Delta M|$ versus z, one for each $\sigma_M = 0.1, 0.2$, and $0.4\,\mathrm{mag}$. At which redshift does selection bias become important in each case?

In [191]:
```
# Call the function
bias_1, _ = bias(M, sigma_1, n)
bias_2, _ = bias(M, sigma_2, n)
bias_3, _ = bias(M, sigma_3, n)
```

✓

```
C:\Users\DELL\AppData\Local\Temp\ipykernel_12720\1373418499.py:24: R
untimeWarning: divide by zero encountered in log10
  m = M + 5*np.log10(d) + 25 # mag
C:\Users\DELL\AppData\Local\Temp\ipykernel_12720\1362377157.py:21: R
untimeWarning: divide by zero encountered in log10
  M_observed = clean_m - 5*np.log10(d) - 25
C:\Users\DELL\AppData\Local\Temp\ipykernel_12720\1362377157.py:21: R
untimeWarning: invalid value encountered in subtract
  M_observed = clean_m - 5*np.log10(d) - 25
```

Suggested approach:

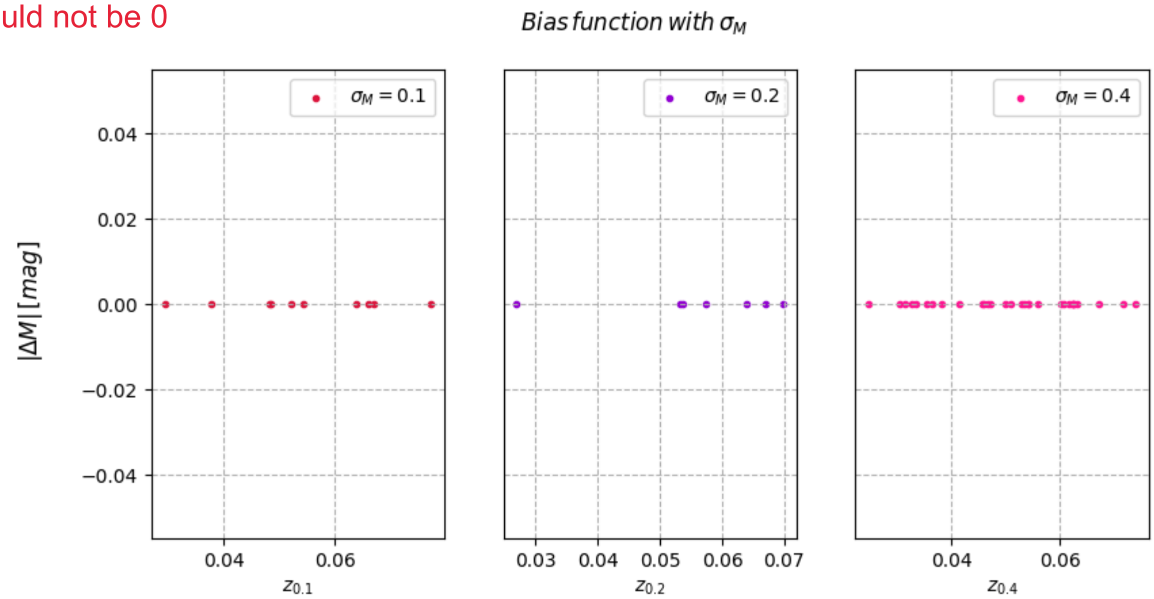
```
#Apparent magnitude m for the N created objets with sigma 0.1
modified_m1 = np.where(m1<18.5, m1, 18.5)

#Apparent magnitude m for the N created objets with sigma 0.2
modified_m2 = np.where(m2<18.5, m2, 18.5)

#Apparent magnitude m for the N created objets with sigma 0.4
modified_m3 = np.where(m3<18.5, m3, 18.5)


#observed magnitude M for the N created objets with sigma 0.1
new_M1 = modified_m1 - 5*np.log(c*z*(1/(H0))) - 25 #we average over the same range z

#observed magnitude M for the N created objets with sigma 0.2
new_M2 = modified_m2 - 5*np.log(c*z*(1/(H0))) - 25 #we average over the same range z

#observed magnitude M for the N created objets with sigma 0.3
new_M3 = modified_m3 - 5*np.log(c*z*(1/(H0))) - 25 #we average over the same range z
```

In [192]:

```
# Plotting
fig, axs = plt.subplots(1, 3, figsize = (9.5,4.5), sharey=True)
fig.suptitle(r"$Bias\,function\,with\,\sigma_M$")
fig.supylabel(r"$|\Delta M|\,[mag]$")
axs[0].scatter(z_1, bias_1, marker = ".", color = "crimson", label = r
axs[0].grid(linestyle = "--")
axs[0].set_xlabel(r'$z_{0.1}$')
axs[0].legend()

axs[1].scatter(z_2, bias_2, marker = ".", color = "darkviolet", label
axs[1].grid(linestyle = "--")
axs[1].set_xlabel(r'$z_{0.2}$')
axs[1].legend()

axs[2].scatter(z_3, bias_3, marker = ".", color = "deeppink", label =
axs[2].grid(linestyle = "--")
axs[2].set_xlabel(r'$z_{0.4}$')
axs[2].legend()

# To get rid of the little horizontal lines in between
for i in axs[1:]:
    i.tick_params(axis='y', length=0)

plt.show()
```

-1

delta M should not be 0



The bias I get is equal to 0 in all cases and regardless of the seed. I assume that happens because I used the random values to generate the z-vector and the data to be compared with the true value, in which case, from what I understand, I'd be comparing a thing with something very very similar to it. However, then, where could the random set of numbers be used in? Check m calculation. Bias is 0 only at small z, then it should increase.

I do not know, now, what else might be causing this. Perhaps I committed a *great* mistake when defining the functions, or assigning the variables, but sadly I couldn't find it (them). I'll look it up, but for now I have to send the notebook.

Truly thank you for the course prof. Wladimir, it's been awesome!