Jet formation from a drop impacting a cone : Instruction for running simulations

Valentin Laplaud, August 29, 2024

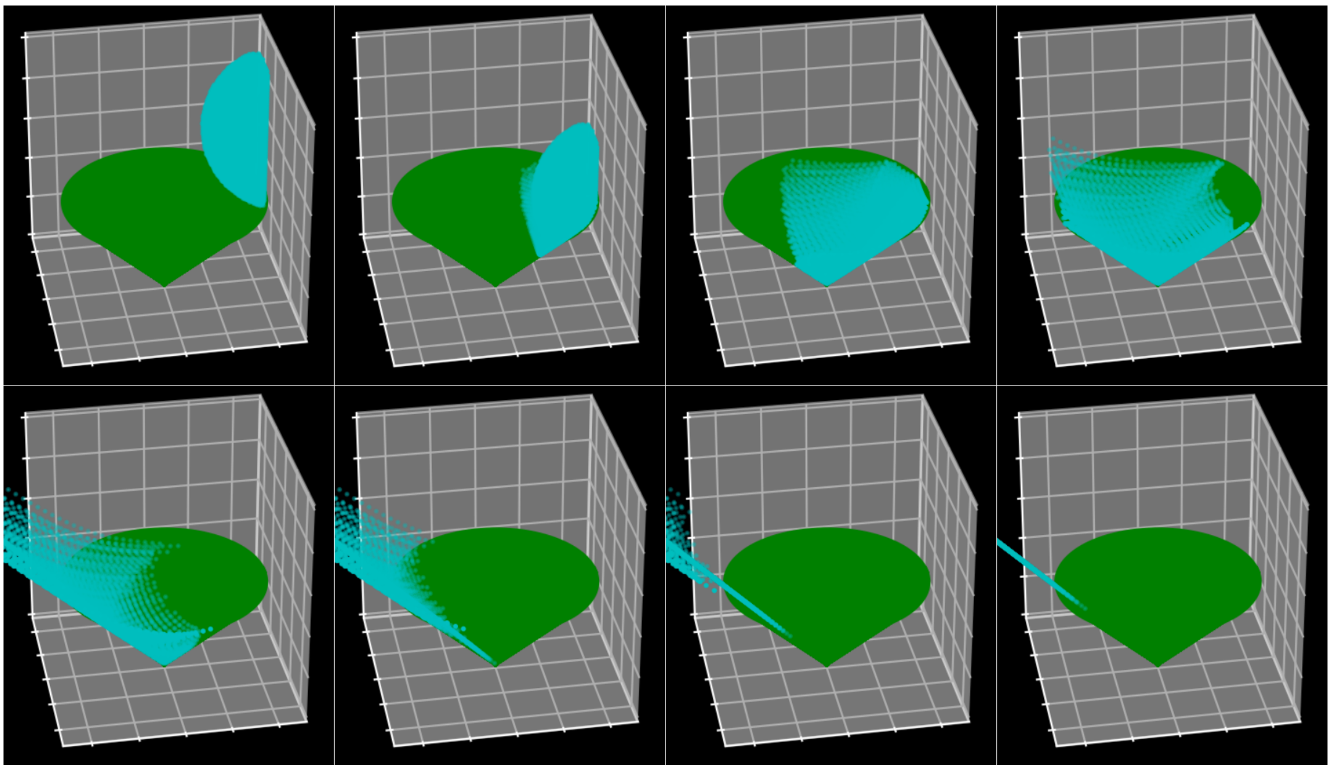
Figure 0: 3D visualisation of a drop impacting a cone.

Contents

1 How to use

The purpose of this document is to present the code for running simulations based on the model presented in the document 'Jet formation from a drop impacting a cone'. The first part of the document will give a general overview of the structure of the code and present the different user defined parameters and output variables. The second part of the document is a guide to run simulations and produce phase diagrams and trajectories visualisation in 2D and 3D. The last part describes in details (some of) the functions and methods in the code.

2 General overview

2.1 Language and packages

The code is written in Python and executed in Jupyter Notebook. The necessary packages are :

- numpy
- matplotlib
- IPython
- pandas
- seaborn
- os
- mpmath
- scipy
- shutil

The IPython package is only usefull if you also work with Notebook, otherwise you shouldn't need it. In addition to these, the code is splitted in four different files :

- VallapFunc_Sim.py
- DropGeometryClasses.py
- DropGeometryFuncs.py
- ImpactSimulationFuncs.py

All the subpackages, functions and aliases used in those 4 files are summarized in the following :

```
import os
import shutil

import numpy as np
import numpy.matlib as ml
import pandas as pd
import mpmath as mpm

from scipy.interpolate import LinearNDInterpolator
from scipy.spatial.distance import directed_hausdorff
from scipy import optimize

import matplotlib.pyplot as plt
import matplotlib.colors as colors
import seaborn as sns

from IPython import get_ipython

import DropGeometryClasses as dgc
import VallapFunc_Sim as vf
import DropGeometryFuncs as dgf
```

2.2 File organisation and code structure

VallapFunc_Sim.py contains basic geometric and plots functions, some (not all) are used in 'DropGeometryFuncs' and 'DropGeometryClasses'.

DropGeometryFuncs.py is a set of functions related to the geometric model such as transformation of positions and velocity from one configuration to the other. They are used in 'DropGeometryClasses'.

DropGeometryClasses.py is the file where the classes for Drop, Cone, and Impact are defined. The Drop and Cone classes create objects containing shape parameters, geometry computation methods, visualisation methods, and an impact simulation method. The Impact object is initialized from a Drop and a Cone and simulates the trajectories of fluid particles resulting from the impact at initilization. It contains methods to simulate the trajectories, to compute the variables of interest from the simulation, and to display various aspects of the simulation process such as initial velocities, trajectories in 2D or 3D. The classes defined in this file can be directly used to compute individual impact simulations and look specifically at trajectories for example. They are also used in 'ImpactSimulationFuncs'.

ImpactSimulationFuncs.py contains the wrapping functions that are called to run large numbers of simlations and generate diagrams agregating the results.

2.3 Parameters and output variables

In order to run simulations and get graphed results, the user needs to use the classes from 'DropGeometryClasses' for single impact or the functions from 'ImpactSimulationFuncs' for multiple impacts. We will cover here the input parameters and outputs of the different class methods and wrapping functions.

The Drop class is initialized with the call :

```
D = dgc.Drop(radius, offcent, npts, vel)
```

Where *radius* is the radius of the drop in $[mm]$, $offcent$ is the distance between the drop center and the cone center, called off-centering, in $[mm]$, *npts* is the number of point on one side of the 2D drop mesh ($\approx npts^2$ points in the final circular mesh), *vel* is the falling velocity of the drop in $[m/s]$. The output D is an object containing the input parameters, the computed mesh, and computations methods.

The Cone class is initialized with the call :

```
C = dgc.Cone(radius, angle)
```

Where *radius* is the radius of the cone in $[mm]$, and *angle* the internal angle of the cone with the vertical in $[rad]$. The output C is an object containig the input parameters and computation and display methods.

The Impact class can be initialized directly or by using internal methods. In both cases you need a Cone and Drop objects:

```
D = dgc.Drop(...)
C = dgc.Cone(...)


I = dgc.Impact(D,C,oriType,velIni,velType,meshType)
I = D.impact(C,oriType,velIni,velType,meshType)
I = C.impact(D,oriType,velIni,velType,meshType)
```

Where *oriType* is a string that can take the values '`Hgrad`', `Hmax`', `Drop`', or `Central`' and defines the method for computing initial orientation of trajectories (default : '`Hgrad`').

velIni is a string that can take the values '`VelNorm`' or '`Radial`' and defines the initial velocity distribution for the normal impact (all trajectories have the same velocity, or the norm is proportional to the distance to the 'drop center', default : '`Radial`').

velType is a string that can take the values '`full`','`tan`', or '`norm`' and defines the velocity components used to compute the trajectories (respectively complete velocity, only tangential velocity, only normal velocity. default : '`full`').

meshType is a string that can take the values '`point`' or '`zone`' and defines the dimension of the mesh for the drop (default : '`zone`').

The output I is an object with the Drop and Cone objects stored in as well as the trajectories resulting from the impact simulation. It contains methods to compute and return various quantification variable, and display method to illustrate the impact.

The PhaseDiagrams function call signature is :

```
PhaseDiagrams(RelOffCents,ConeSize,ConeSizeType,Angles,RelDropDiams,
    ↪ oriType,velIni,velType,meshType,path,label)
```

Where *oriType*, *velIni*, *velType*, *meshType* are the same as previously.

$RelOffCents$, $RelDropDiams$ and *Angles* are the vectors defining the parameter space, respectively for relative offcentering, relative drop diameter and cone angle (in [rad]).

ConeSizeType and *ConeSize* respectively define the type of fixed dimension (cone surface [mm^2] or radius [mm]) and the value of said dimension.

path is a path string for the location where the figure folder will be created, *label* is a string that will be the name of the figure folder.

The output is a folder named `label`, saved at the location `path`, and containing the phase diagrams for all quantification variables in the parameter space defined for the run, and graphes of maximum value by angle for the main quantifications. The data are also saved in this folder, so re-running `PhaseDiagrams` with the same parameters will load the data directly and make the graphs. This can be used to plot differently, or to add graphs to the results of the same simulations if necessary.

The OptiDiagrams function call signature is :

```
OptiDiagrams(coneSurface,coneAngles,npts,ndrops,dropRadii,dropScaling,
    ↪ path,label)
```

Where $coneSurface$ is the surface of the cone in [mm^2].

coneAngles and *dropScaling* are vectors defining the parameter space for the diagram. *coneAngle* is in [rad] and *dropScaling* is the surface ration between the cone and the average (median) drop in the rain.

dropRadii is a vector containing the size distribution for drops (radius, in [mm])

npts and *ndrops* are respectively the resolution of the diagrams and the number of drops simulated in the rain. *npts* must correspond to the length of *coneAngles* and *dropScaling*

path is a path string for the location where the figure folder will be created, *label* is a string that will be the name of the figure folder.
The output is a folder named *label* saved in the location *path*. It contains the optimization diagrams and datasets for all quatification variables.

3 User guide

This section will describe with examples how to use the functions listed in section 2.3. For more details about the signification of the parameters refer to section 2.3

3.1 Simulate an impact and retrieve output variables

To simulate the impact of a drop on a cone, you only need to define the drop and cone, and use the built in `impact` method :

```
npts = 21 # points in the mesh

ConeRadius = 2.7 # cone radius in [mm]

ConeAngle = np.pi/4 # in [rad]

DropRadius= 0.9*ConeRadius # Drop size in [mm]

DropVel = 5 # impact velocity in [m/s]

Roc = 1.2 # relative off-centering

D = dgc.Drop(DropRadius,Roc*ConeRadius,npts,DropVel) # drop creation
C = dgc.Cone(ConeRadius,ConeAngle) # cone creation

oriType = 'Hgrad' # direction of velocities
velIni = 'Radial' # type of initial velocities distribution
velType = 'full' # Component of velocity to use
meshType = 'zone' # Type of drop mesh



# impact and trajectories simulation
I = C.impact(D,oriType,velIni,velType,meshType)
```

You now have an object `I` that contains the parameters of the drop and cone, as well as the trajectories resulting from the impact. You can now compute and get acces to the various output parameters :

```
JF1 = I.get_JetFrac()[0] # Fraction of impact volume in primary jet
JF2 = I.get_JetFrac()[1] # fraction of impact volume in secondary jet

JV = I.compute_JetVel() # Primary jet velocity at ejection

KNRJ = I.compute_JetNRJ()[0] # kinetic energy of the primary jet
```

```
BNRJ = I.compute_JetNRJ()[1] # balistic energy of the primary jet

DD = I.compute_DispertionDist()[0] # Dispersion distance for the primary
    ↪  jet
DDv = I.compute_DispertionDist()[1] # Variability (per trajectory) of
    ↪ dispersion distance for the primary jet
DH = I.compute_DispertionDist()[2] # Height reached by the primary jet

SO = I.SheetOpening()[0] # Fluid sheet opening angle
SF = I.compute_ShapeFactor() # Fluid sheet shape factor
```

3.2 Plot velocities and trajectories

The Impact class has several display methods to show initial velocities, trajectories in 2D, and animations of the splash in 2D and 3D.

3.2.1 Inital velocities

To display the initial velocity of each impact point of the mesh, you need to use the `plot_splash_init(velType,**kwargs)` method :

```
...
# impact and trajectories simulation
I = C.impact(D,oriType,velIni,velType,meshType)

# Velocity components to show ['full','tan','norm']
velType_Display = 'full'

figtitle = oriType + '_' + velIni + '_' + meshType

I.plot_splash_init('full',nolabels=False,title=figtitle,xlabelCi='Jet␣
    ↪ axis',ylabelCi='Other␣axis', xlabelCo='Jet␣axis␣(mm)',ylabelCo='
    ↪ Other␣axis␣(mm)')
```

The resulting figure is not saved automatically !!
The only argument here is `velType` and is the components of the velocity that will be displayed. The keywords argument allow you to turn ON and OFF the labels and to define each of them. The computation time for a single impact is almost instantaneous. Figure 1 show the figure obtained with the previous call.

3.2.2 Trajectories

To display the trajectories of the fluid particles resulting from the impact you need to use the `plot_splash_traj(Time,**kwargs)` methods :

```
...
# impact and trajectories simulation
I = C.impact(D,oriType,velIni,velType,meshType)

figtitle = oriType + '_' + velIni + '_' + meshType

Time = np.linspace(0,1,20) # time points to plot in [ms]
```

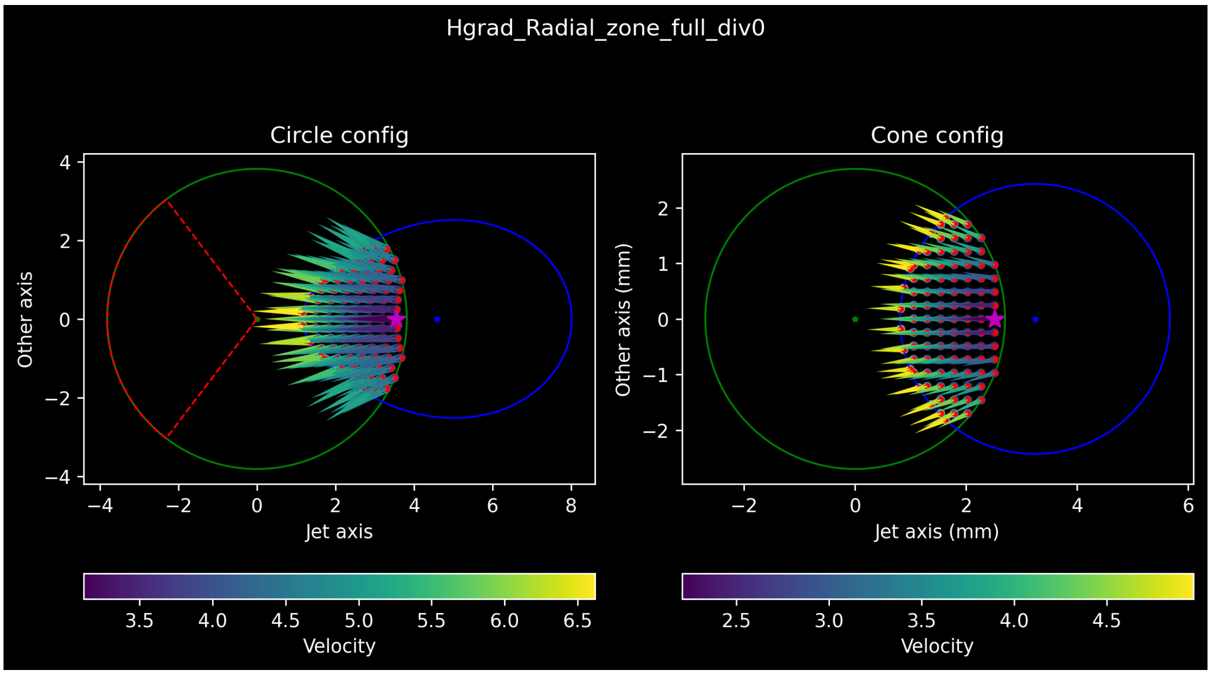

Figure 1: Initial velocities from `plot_splash_init`

```
I.plot_splash_traj(Time,nolabels=True,title=figtitle)
```

The resulting figure is not saved automatically !!
The only argument here is `Time` and is a vector of all the timepoints where the particle position will be displayed. The keywords argument are the same as for the initial velocities. The computation time for a single impact is almost instantaneous. Figure 2 show the figure obtained with the previous call.

3.2.3 2D and 3D animations

Finally, you can get 2D and 3D animation of the splash.

`plot_splash_movie(Time,path,label,xlims,ylims,**kwargs)` creates an animated version (saved as a folder of *.png* images) of `plot_splash_traj` where the particles are colored depending on there final identity : in the primary jet, in the secondary jet, or in the fluid sheet.

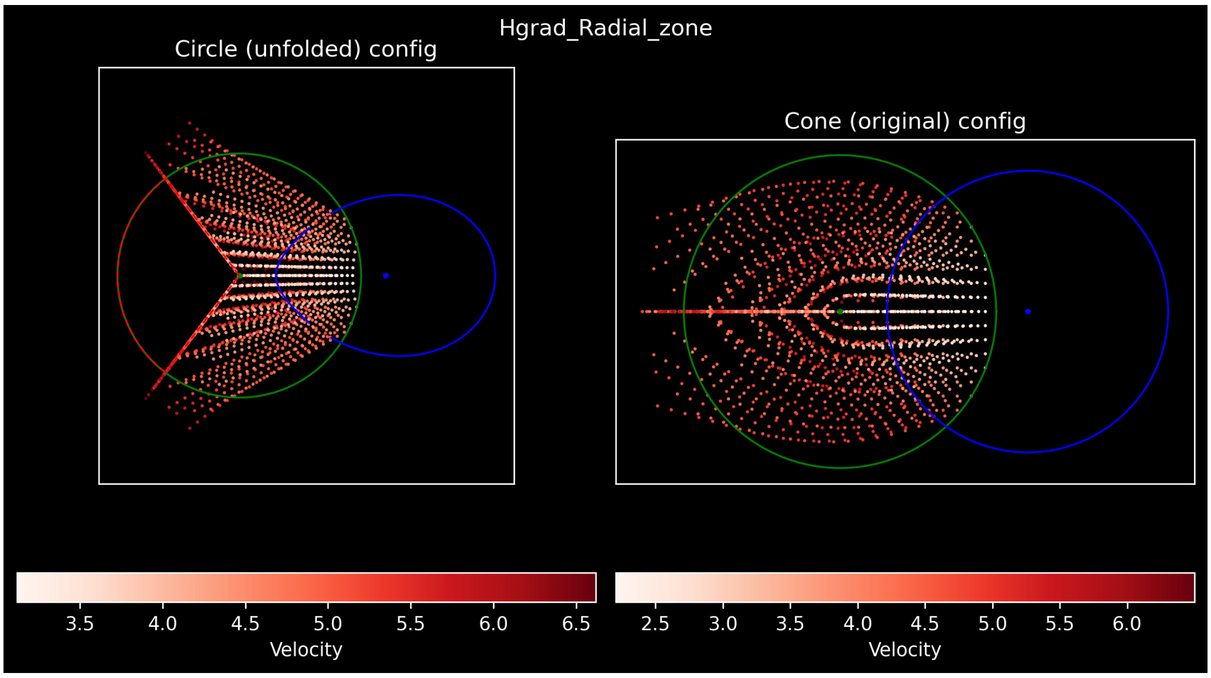

Figure 2: Trajectories from `plot_splash_traj`

```
...
# impact and trajectories simulation
I = C.impact(D,oriType,velIni,velType,meshType)

figtitle = oriType + '_' + velIni + '_' + velType

Time = np.linspace(0,5,50*5) # 50 pts/ms is a minimum resolution to get
    ↪ good trajectories

Xlims = 1.1*2*ConeRadius*np.array([-1.5,0.5])
Ylims = 1.1*2*ConeRadius*np.array([-0.5,0.5]) # limits for the cone
    ↪ config display

path = YOUR SAVING PATH

label = YOUR FOLDER NAME

I.plot_splash_movie(Time,path,label,Xlims,Ylims,nolabels=True,title=
    ↪ figtitle)
```

Figure 3 shows 4 time points images from the previous call. The generation of all the images takes around 10 minutes for the showed parameters.
`plot_3Dview(path,label,figtitle)` and `rotating_3D(path,label,figtitle)` produce 3D animations of the drop impacting the cone. The '3Dview' movie is in a fixed space and has both cone and circle configuration, the 'rotating3D' only has cone configuration and is for nicer display on the simulation result to present. Note that `plot_3Dview` and `rotating_3D` do not have a *Time* argument, it is defined internally from the geometrical parameters.

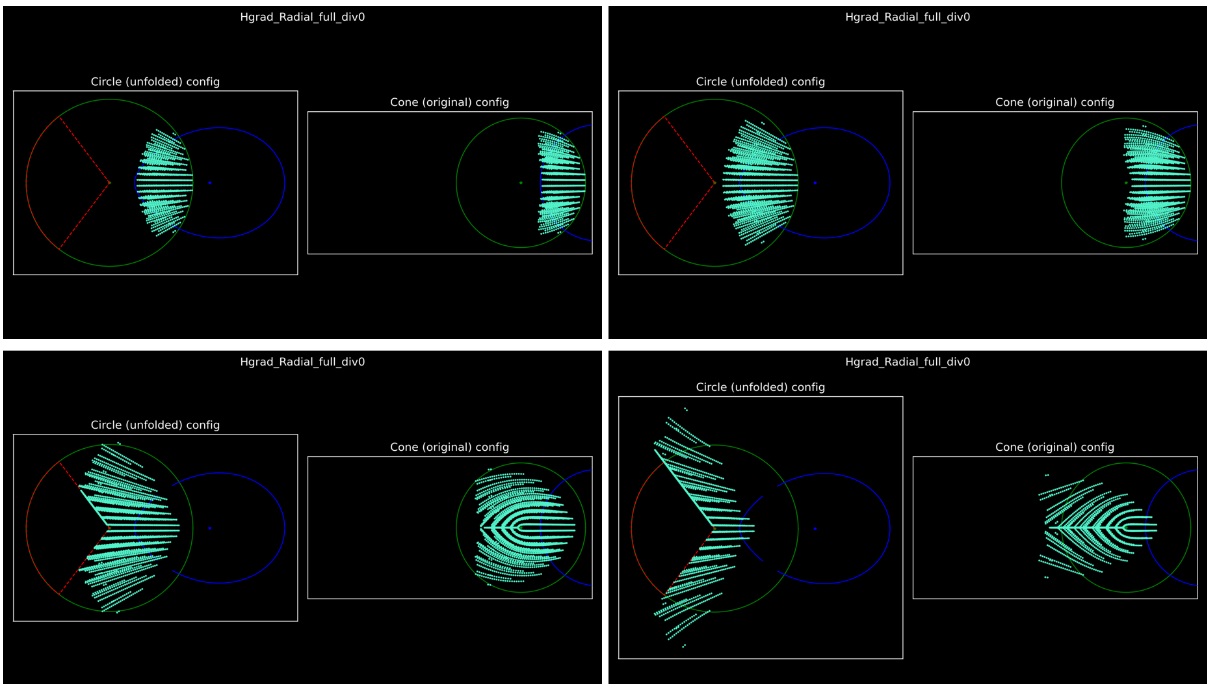

Figure 3: Individual images from the call of `plot_splash_movie`

```
...
# impact and trajectories simulation
I = C.impact(D,oriType,velIni,velType,meshType)

path = YOUR SAVING PATH

label = '3D_Movie_Fixed' + oriType + '_' + velIni + '_' + velType
I.plot_3Dview(path,label)


label = '3D_Movie_Rotating' + oriType + '_' + velIni + '_' + velType
I.rotating_3D(path,label)
```

Figures 4 and 5 show timepoints generated from the previous call. The generation of all the images in each movie takes around 10 minutes for the showed parameters.

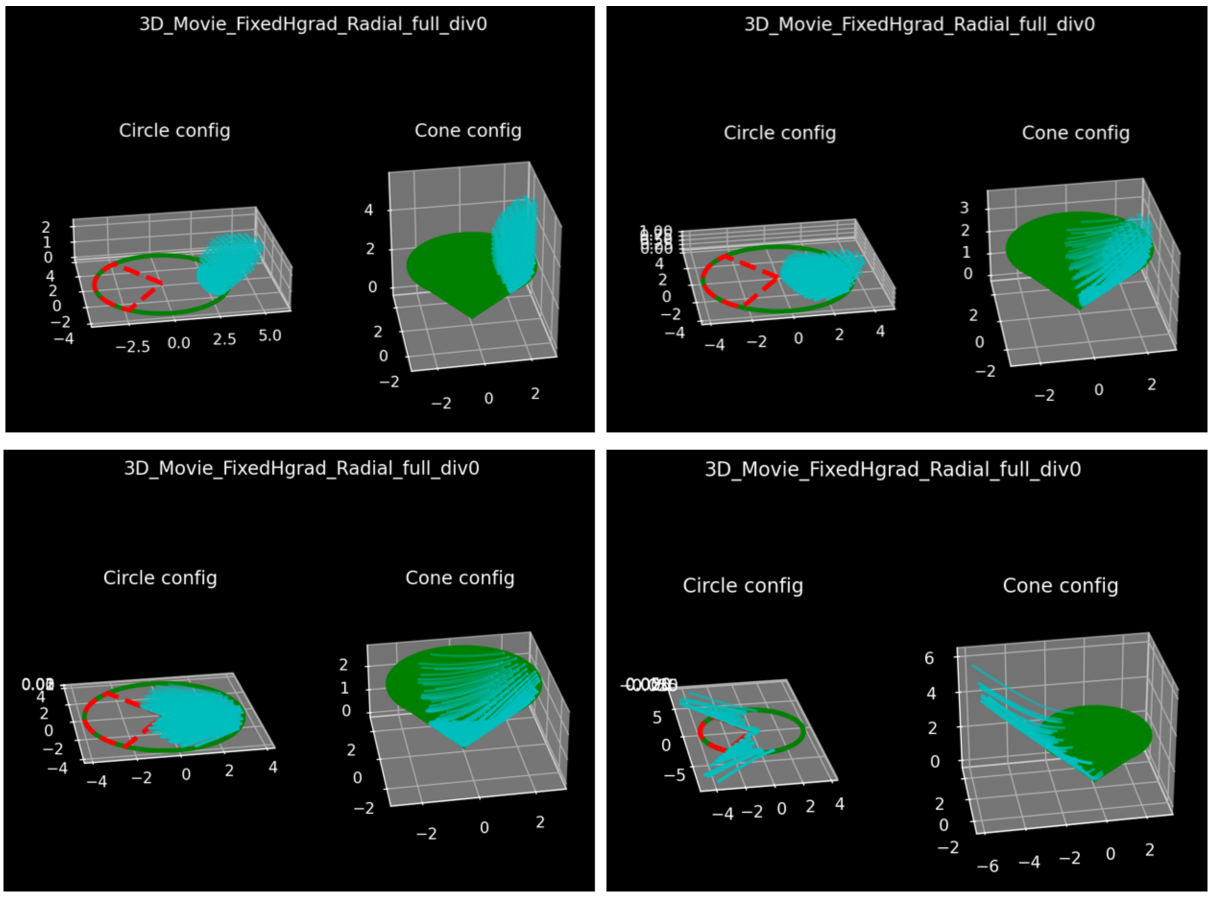

Figure 4: Individual images from the call of `plot_3D_view`

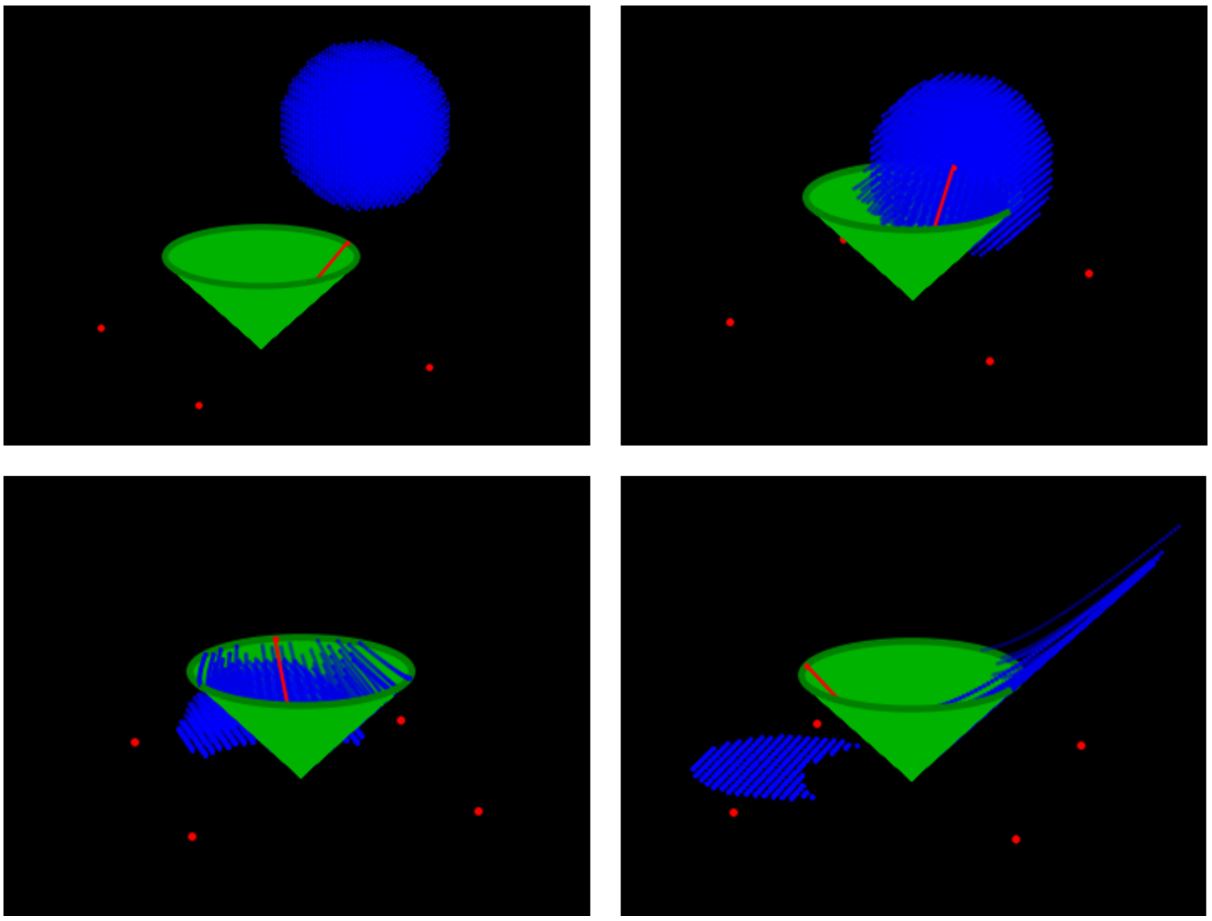
Figure 5: Individual images from the call of `rotating_3D`

3.3 Plot phase diagram for single drop impact

In order to plot phase diagrams for the different output variables you need to use the `PhaseDiagrams` function from the **ImpactSimulationFunction.py** file. The parameters of this function are covered in section 2.3. Below is an example of code for simulating $npts^3 = 15^3 = 3375$ impacts and graphing the results.

```
oriTypes = 'Hgrad' # 'Hmax','Central','Drop'

velType = 'full' # 'norm','tan','tan','full'

velIni = 'Radial' # 'VelNorm'

meshType = 'zone' # 'point'

npts = 15 # XY resolution for the diagram


FixedDim = 'radius' #,'surface'

if FixedDim == 'surface':
    ConeSize = 7.3 # [mm^2] polymorpha :7.3 globosa : 12.1
    RelOffCmax = OffCmax/np.sqrt(ConeSize/np.pi)
    ConeSizeType = 'surface'
elif FixedDim == 'radius':
    ConeSize = 1.5 # [mm]
    RelOffCmax = OffCmax
    ConeSizeType = 'radius'

path = YOUR PATH
# parameter space

OffCmax = 3 # [mm] # maximum relative offcentering

RelDropDiams = np.linspace(0.125,RelOffCmax-1,npts) # relative drop
    ↪ diameter

RelOffCents = np.linspace(0.125,RelOffCmax,npts) # relative
    ↪ offcenterings

Angles = np.linspace(0.5,89.5,npts)/360*2*np.pi # in [rad]

label = velIni + '_' + oriType + '_' + velType + '_' + meshType + '_' +
    ↪ FixedDim + '_YOURLABEL'

isf.PhaseDiagrams(RelOffCents,ConeSize,ConeSizeType,Angles,RelDropDiams,
    ↪ oriType,velIni,velType,meshType,path,label)
```

For examples of the resulting phase diagrams you can look in the main document. For this specific call you can expect a running time of around a day.

3.4 Plot optimisation diagrams for a random rain

In order to plot optimisation diagrams for a random rain you need to use the `OptiDiagrams` function from the **ImpactSimulationFunction.py** file. The parameters of this function are covered in section 2.3. Below is an example of code for simulating $npts^2 * ndrops = 21^2 * 2000 = 882000$ impacts and graphing the results.

```
coneSurface = 7.3 # [mm^2] polymorpha :7.3 globosa : 12.1

npts = 21 # resolution for the diagram

coneAngles = np.linspace(0.5,89.5,npts)/360*2*np.pi

ndrops = 2000

ConeScaling = np.linspace(0.05,3.5,npts)

dropScaling = np.sqrt(np.divide(1,ConeScaling))

# Drop size distribution definition

dropRadii = np.random.gamma(3,0.5, ndrops) # Gamma distrib

dropRadii[dropRadii>5] = dropRadii[dropRadii>5]-5 # Max radius [mm]


label = 'FullScale_SecondJet'
path = r'd:\Users\laplaud\Desktop\PostDoc\Code\DropProject_WithAna\
    ↪ Figures\Optimization\\'

isf.OptiDiagrams(coneSurface,coneAngles,npts,ndrops,dropRadii,dropScaling
    ↪ ,path,label)
```

For examples of the resulting optimisation diagrams diagrams you can look in the main document. For this specific call you can expect a running time of more than a few days (more than 3 days for less than half the impacts at the time of writting this sentence)

4 Code description

To fill when there is time if there is time

4.1 VallapFunc

4.2 DropGeometryFunc

4.3 DropGeometryClasses

Drop

Cone

Impact For the trajectories that meet inside the cone, we detect the timepoints where the border is crossed in the circle configuration. We can extract the position and velocity of the fluid particle at time `it-1` (before the crossing) and `it` (after the crossing) and transform it in the cone configuration. Then we force both the position of the trajectory and the velocity of the point at time `it` to have 0 Y-component, effectively placing it at the border and projecting the velocity along the jet direction.

4.4 ImpactSimulationFunction