PARCELY USER GUIDE

Hello, and thank you for using PARCELY for your cloud parcel modelling needs. This document is a brief and informal instruction manual for first-time users. It will guide you on how to run the model, how to edit the main file to input your initial conditions, and explain the output file. The functions should be commented thoroughly enough that should you want to go looking around in there, it'll be clear what does what. For a theoretical explanation of the model and its physics (including equations and validations), I refer you to the "Model Description" file from my master's thesis on the GitHub page.

The Main and Input File: PARCELY_MAIN.py

Domain Set-Up

```
# Total cubic centimeters of the domain (e.g 5cm3, 4.3cm3 etc.)
Cubes = 1
# Height of boundary box side (m)
BoxHeight = (Cubes**(1/3))*(1e-2)
# Setting a random generator seed for reproducibility
SeedNo = 1996
RNG = np.random.default_rng(SeedNo)
```

The variables are explained in the comments, but a word on SeedNo (short for seed number, set to the year of my birth by default because I'm an egomaniac) and RNG (short for random generator): If you want to run the model multiple times and want to keep the same aerosol distribution, **you must start from RNG**. Running the model without this line (say, selecting only from line 64 because all you did was change the run time) will result in numpy using a different generator (essentially a random one) and you won't be using the same aerosol distribution.

Time Set-Up

```
# Initial timestep, seconds
dt = 0.000001
# Total model runtime, seconds
RunTime = 0.00001
# Number of evenly spaced time instances to save
M = 10
# Times of instances
Instances = np.arange(0, RunTime+RunTime/M, RunTime/M)
# Actually measured time instances
Time = np.zeros_like(Instances)
```

```
# Tolerance
mtol = 1e-23
```

If this is the first time running the model, I **strongly recommend running it first with these time settings.** This is due to possible compiling issues from the way the numba package works (imported in PARCELY_FUNCTIONS). Cached files will be missing, and thus the code needs to be compiled first. I have found that the most efficient way to compile is to run the model at these settings, and then once again, changing only RunTime to 10 seconds. Once you have done that, the model should be fully compiled and will run as fast as it can.

The time-step (dt) is an adaptive time-step (if it weren't I would still be running a simulation from September 2022) and so there should be no reason to change the initial value when performing basic runs of the model. The tolerance (mtol) is what limits the time-step size. As with most any adaptive solver you'll likely have to play around with the values, depending on your solute distribution. Distributions with smaller aerosols will require a high tolerance (I've settled on 10^{-23}) while a monotonic distribution with large radii can have a lower one (e.g. 10^{-21}). Be careful, *changing the tolerance will eventually alter your results.*

Organic Components Set-Up

```
# Surface tension calculation method
Modes = ['OrgFilm', 'Constant', 'MixRule', 'WaterTemp']
# Surface tension of pure water J/m^2
sft = 0.072
# Selected method
SurfMode = Modes[0]

# Total concentration (all phases)
# ug/m3
Scaling = 1
Concentrations = np.array([0.1, 0.1, 0.1, 0.12, 0.12, 0.12, 0.15,
                           0.24, 0.48, 1.18])*Scaling

# Volatility/Saturation Concentration, ug/m3
LogVols = np.arange(-6, 4, 1, dtype=np.float64)
Cstar = 10.0**(LogVols)

# Organic parameters
# Surface tension
SurfTension = 0.04*np.ones(Concentrations.size)

EstimateParameters = False

if EstimateParameters is False:
```

```
    # unitless
    Kappas = np.ones(Concentrations.size)*0.1
    # kg/m3
    Densities = np.ones(Concentrations.size)*1400
    # kg/mol
    MoMass =  np.linspace(400, 200, Concentrations.size)*1e-3

if EstimateParameters is True:
    MoMass, Densities, Kappas = DF.MolecularCorridor(LogVols, RNG)

# OA parameter table
OAParameters = DF.OrganicsArray(MoMass, Concentrations, Kappas,
                                SurfTension, Densities, Cstar)
```

Here, you must first choose your selected mode. If you are running a bare-bones model with no organics whatsoever, you should select 'Constant' - meaning using the value for the surface tension of pure water (sft); this can be any value you choose, so it doesn't have to be for pure water - or if you want to be more physically accurate, 'WaterTemp'. 'WaterTemp' uses the surface tension of pure water but as a function of the ambient temperature. With organics, you can either select 'MixRule', which is simply a volume-weighted mixing rule for the surface tension, or 'OrgFilm', which solves for an organic film around the aerosol (you're referred to the theoretical explanation if curious).

Scaling simply scales the concentration of organics (most likely to a smaller value) if needed. The concentrations, as mentioned in the comments, is what you'd like to have in total, not just gas-phase.

Surface tension, Kappas, Densities, and MoMass can easily be changed to have different values for each component, you don't need to stick to the way it is here. As long as it in array-form (and the same length as Concentrations), you're good.

EstimateParameters activates the molecular-corridor approximations if you don't have specific values you want to use for the organic parameters. If you have exact values you want to use, keep it False.

Solute Set-Up

```
# Dirichlet mass fractions for non-monotone distribution
Dirichlet = False
# Allow organics
Organics = True
# Allow co-condensation
CoCond = True
# Allow distribution in kappa values
KappaRand = False
```

```
9
10 # Distribution type
11 Distribution = 'lognormal'
12
13 Ns = np.array([125, 65])*Cubes
14 Rs = np.array([0.011, 0.060])
15 Stds = np.array([1.2, 1.7])
16
17 Inorgs = np.array([[0],[0]])
18 InorgPopFrac = np.array([[100],[100]])
19 Orgs = [[0,1,2,3,4,5,6,7,8,9],[0,1,2,3,4,5,6,7,8,9]]
20
21 # If irrelevant, set to 0
22 PercCond = 0
23 MassFractions = [np.array(0.6, 0.1, 0.05, 0.05, 0.05, 0.05,
24                           0.03, 0.03, 0.02, 0.01, 0.01,),
25                  np.array(0.6, 0.1, 0.05, 0.05, 0.05, 0.05,
26                           0.03, 0.03, 0.02, 0.01, 0.01,)]
27
28 # Number of droplets
29 NumberDrops = Ns.sum()
30
31 start = time.time()
32
33 # Get solute arrays
34 Solutes, SoluteFilter, VaporConcs = DF.SolPopInitialize(Ns, Rs, Stds,
35                                          Inorgs, InorgPopPerc, Orgs,
   PercCond,
36                                          OAParameters, RNG, MassFractions,
37                                          Dirichlet, KappaRand, Organics,
38                                          Cubes, Distribution)
39
40 # Get vapor concentrations
41 OAParameters[:,4] = VaporConcs
42
43 end = time.time()
44 print('Solute/OA Initialization: ', np.round(end - start, 4), 's')
```

This section is the last of the inputs (you're almost done!). After setting your preferences with the four Boolean inputs at the beginning, you must choose either a "mono", "normal", or "lognormal" Distribution (string). The example above is for an ammonium sulphate Pristine aerosol distribution, with the inorganic commanding $60\,\%$ of the mass of each aerosol, and the rest taken up by the organic species. Each element in Ns/Rs/Stds is a component of the total distribution (one lognormal distribution with 125 particles, 0.011 geometric mean radius in μm, and geometric standard deviation in μm.)

For Inorgs, the integers indicate the index of the requested row from the INORGANICS CSV file. 0 is ammonium sulphate, in this example. This is an array of lists in order to allow for more freedom in determining which species are in which sub-distribution, and must be lists even if there's only one species per sub-distribution. The same goes for InorgPopPerc, which is the percentage of each species in each sub-distribution.

Orgs is a list of lists, where each list represents which organic species taken from the OAParameters table is present in condensed form in the aerosol sub-distributions.

PerCond is an alternative way to determine how much of each organic is in its condensed phase. I would stick with defining mass fractions though, as it is more straightforward. If you're persistent, PerCond takes an array of percentages indicating how much of the organic species has condensed.

MassFractions are fractions of each solute component (inorganic and organics) per sub-distribution. If Dirichlet is True, MassFractions is used as a set of parameters for a Dirichlet distribution, so that the solute mass fractions (inorganics, organics) are randomly distributed around the same proportions.

SolPopInitialize returns three things:

1. Solutes : array of your solute population (with or without organics)
2. SoluteFilter : array of your inorganic part of your solute (identical to Solutes if no organics present)
3. VaporConcs : If there are organics, this returns the vapor concentrations of the organics in $\mu g\,m^{-3}$ and are put into the organic parameters table on line 41. If no organics are involved, it returns 0.

That's it! The rest is automatic, and if you want to know what the functions in *Koehler Curves and Critical/Equilibrium parameters, Droplet Initialization*, and *Simulation* do, you can look into their descriptions.

The Output File

In keeping with the pure-Python style, the output file for PARCELY is a .npz file, which you can read about in a more technical manner here. It is easily read into Python by the numpy load function, and consists of the following subfiles:

1. DropOutput - Array for the droplet data. In order, its columns are droplet index, position, vertical velocity, water mass, radius, surface saturation ratio, growth rate, and surface tension. First dimension is droplet, second is data column, third is time.

2. Solute - If co-condensation is included, this is an array similar to DropOutput but for the solute data instead. In order, its columns are solute index, radius, total solute mass, molar mass of inorganic, mean density, effective hygroscopicity, and the remainder are the organic mass of each component. Without co-condensation, the solute doesn't change with time so there is no time-dimension (2D array).

3. CritParams - 2D array of critical parameters assuming a surface tension of pure water. First column is critical radius, second is critical saturation ratio.

4. OAParams - If co-condensation is included, this output is the organic parameter array. Each row is a different organic component, and in order the columns are molar mass, density, diffusivity, saturation concentration, final vapor concentration, hygroscopicity, pure component surface tension, and total concentration.

5. ChemData - If co-condensation is included, this output is a 3D array where each column is an organic component, and the third dimension is time. The first dimension represents the data variables for each organic component, which are in order the vapor concentration in molecules per cubic centimeter, the saturation concentration in $\mu g\,m^{-3}$, the mean surface saturation concentration in $\mu g\,m^{-3}$, and the mean condensed mole fraction.

6. SatTime - The evolution of saturation ratio over time.

7. TempTime - The evolution of ambient temperature over time.

8. PressTime - The evolution of ambient pressure over time.

9. HeightTime - The mean height of the particles over time.

10. Time - Time array.

11. MontVars - Array containing the generator seed number, the updraft velocity, and the mass and thermal accommodation coefficients used in initializing the simulation.

To access one of the output subfiles, for example DropOutput:

```
# Load numpy
import numpy as np
# Load data file into memory
SimulationData = np.load('YourPathHere/DataFile.npz')
# Get droplet data
ExampleDropData = SimulationData['DropOutput']
```

Heterogeneous Extra

```
    # By how much the air parcel's side is subdivided
    SatDivide = 5
    SatField, SatGrid, SatInd = DF.SatFieldInit(S, SatDivide,
                    DomainXLims, DomainYLims, DomainZLims, RNG, 'mono')

    Diffusion = True
    DropMove = True

    TempField = np.ones_like(SatField)*T
    PressField = np.ones_like(SatField)*P
```

In the heterogeneous version of PARCELY, where the domain is subdivided into a grid, we have this extra section above. In this example, SatDivide is set equal to 5, meaning we have 5 divisions per axis, resulting in 125 total sub-grids. Diffusion determines whether you allow the model to diffuse temperature and saturation ratio. DropMove allows the droplets to move with billiard-ball like motion in the horizontal (x,y directions).