pyrcel documentation

Release 1.3.1.dev-2da4611

Daniel Rothenberg

Aug 25, 2019

CONTENTS

1 Documentation Outline **3**

1.1 Scientific Description 3
1.2 Installation 6
1.3 Example: Activation 8
1.4 Example: Basic Run 10
1.5 Parcel Model Details 15
1.6 Reference 19

Bibliography **37**

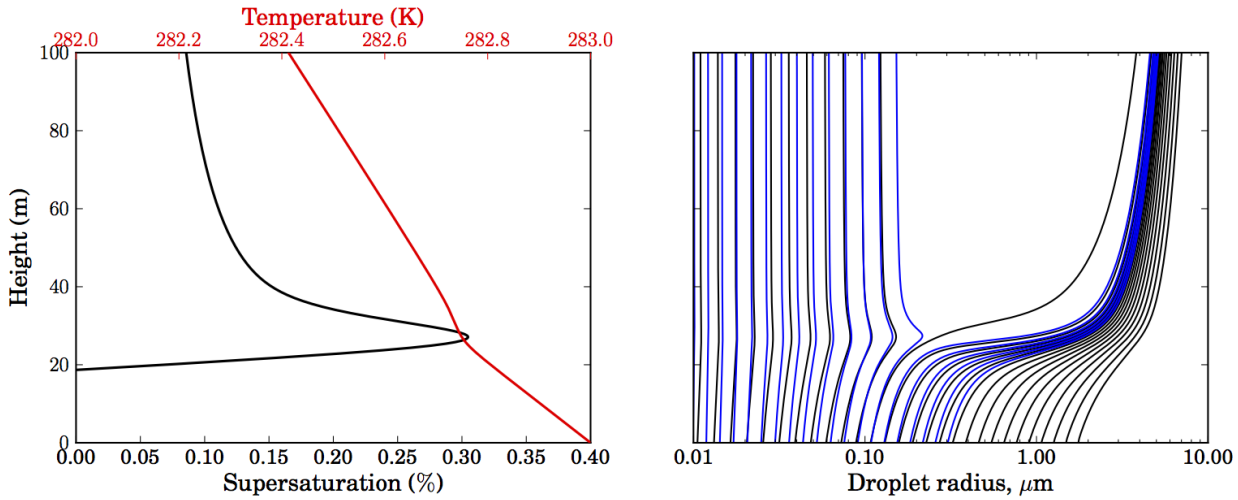
Python Module Index **39**

Index **41**

.svg

target https://github.com/python/black

This is an implementation of a simple, 0D adiabatic cloud parcel model tool (following Nenes et al, 2001 and Pruppacher and Klett, 1997). It allows flexible descriptions of an initial aerosol population, and simulates the evolution of a proto-cloud droplet population as the parcel ascends adiabatically at either a constant or time/height-dependent updraft speed. Droplet growth within the parcel is tracked on a Lagrangian grid.



You are invited to use the model (in accordance with the licensing), but please get in touch with the author via e-mail or on twitter. p-to-date versions can be obtained through the model's github repository or directly from the author. If you use the model for research, please cite this journal article which details the original model formulation:

Daniel Rothenberg and Chien Wang, 2016: Metamodeling of Droplet Activation for Global Climate Models. *J. Atmos. Sci.*, **73**, 1255–1272. doi: http://dx.doi.org/10.1175/JAS-D-15-0223.1

CHAPTER

ONE

DOCUMENTATION OUTLINE

1.1 Scientific Description

The simplest tools available for describing the growth and evolution of a cloud droplet spectrum from a given population of aerosols are based on zero-dimensional, adiabatic cloud parcel models. By employing a detailed description of the condensation of ambient water vapor onto the growing droplets, these models can accurately describe the activation of a subset of the aerosol population by predicting how the presence of the aerosols in the updraft modify the maximum supersaturation achieved as the parcel rises. Furthermore, these models serve as the theoretical basis or reference for parameterizations of droplet activation which are included in modern general circulation models ([Ghan2011]) .

The complexity of these models varies with the range of physical processes one wishes to study. At the most complex end of the spectrum, one might wish to accurately resolve chemical transfer between the gas and aqueous phase in addition to physical transformations such as collision/coalescence. One could also add ice-phase processes to such a model.

1.1.1 Model Formulation

The adiabatic cloud parcel model implemented here is based on models described in the literature ([Nenes2001], [SP2006],) with some modifications and improvements. For a full description of the parcel model, please see ([Rothenberg2016]) The conservation of heat in a parcel of air rising at constant velocity V without entrainment can be written as

$$\frac{dT}{dt} = -\frac{gV}{c_p} - \frac{L}{c_p}\frac{dw_v}{dt} \tag{1.1}$$

where T is the parcel air temperature. Assuming adiabaticity and neglecting entrainment is suitable for studying cloud droplet formation near the cloud base, where the majority of droplet activation occurs. Because the mass of water must be conserved as it changes from the vapor to liquid phase, the relationship

$$\frac{dw_v}{dt} = -\frac{dw_c}{dt} \tag{1.2}$$

must hold, where w_v and w_c are the mass mixing ratios of water vapor and liquid water (condensed in droplets) in the parcel. The rate of change of water in the liquid phase in the parcel is governed solely by condensation onto the existing droplet population. For a population of N_i droplets of radius r_i, where $i = 1, \ldots, n$, the total condensation rate is given by

$$\frac{dw_c}{dt} = \frac{4\pi\rho_w}{\rho_a}\sum_{i=1}^{n} N_i r_i^2 \frac{dr_i}{dt} \tag{1.3}$$

Here, the particle growth rate, $\frac{dr_i}{dt}$ is calculated as

$$\frac{dr_i}{dt} = \frac{G}{r_i}(S - S_{eq}) \tag{1.4}$$

where G is a growth coefficient which is a function of the physical and chemical properties of the particle receiving condensate, given by

$$G = \left(\frac{\rho_w RT}{e_s D'_v M_w} + \frac{L\rho_w[(LM_w/RT) - 1]}{k'_a T} \right)^{-1} \tag{1.5}$$

Droplet growth via condensation is modulated by the difference between the environmental supersaturation, S, and the droplet equilibrium supersaturation, S_{eq}, predicted from Kohler theory. To account for differences in aerosol chemical properties which could affect the ability for particles to uptake water, the κ-Köhler theory parameterization ([PK2007]) is employed in the model. κ-Kohler theory utilizes a single parameter to describe aerosol hygroscopicity, and is widely employed in modeling of aerosol processes. The hygroscopicity parameter κ is related to the water activity of an aqueous aerosol solution by

$$\frac{1}{a_w} = 1 + \kappa \frac{V_s}{V_w}$$

where V_s and V_w are the volumes of dy particulate matter and water in the aerosol solution. With this parameter, the full κ-Kohler theory may be expressed as

$$S_{eq} = \frac{r_i^3 - r_{d,i}^3}{r_i^3 - r_{d,i}^3(1 - \kappa_i)} \exp\left(\frac{2M_w \sigma_w}{RT\rho_w r_i} \right) - 1 \tag{1.6}$$

where r_d and r are the dry aerosol particle size and the total radius of the wetted aerosol. The surface tension of water, σ_w, is dependent on the temperature of the parcel such that $\sigma_w = 0.0761 - 1.55 \times 10^{-4}(T - 273.15)$ J/m^2 . Both the diffusivity and thermal conductivity of air have been modified in the growth coefficient equation to account for non-continuum effects as droplets grow, and are given by the expressions

$$D'_v = D_v \Big/ \left(1 + \frac{D_v}{a_c r} \sqrt{\frac{2\pi M_w}{RT}} \right)$$

and

$$k'_a = k_a \Big/ \left(1 + \frac{k_a}{a_T r \rho_a c_p} \sqrt{\frac{2\pi M_a}{RT}} \right)$$

In these expressions, the thermal accommodation coefficient, a_T, is assumed to be 0.96 and the condensation coefficient, a_c is taken as unity (see *Constants*). Under the adiabatic assumption, the evolution of the parcel's supersaturation is governed by the balance between condensational heating as water vapor condenses onto droplets and cooling induced by the parcel's vertical motion,

$$\frac{dS}{dt} = \alpha V - \gamma \frac{w_c}{dt} \tag{1.7}$$

where α and γ are functions which are weakly dependent on temperature and pressure :

$$\alpha = \frac{gM_w L}{c_p RT^2} - \frac{gM_a}{RT}$$

$$\gamma = \frac{PM_a}{e_s M_w} + \frac{M_w L^2}{c_p RT^2}$$

The parcel's pressure is predicted using the hydrostatic relationship, accounting for moisture by using virtual temperature (which can always be diagnosed as the model tracks the specific humidity through the mass mixing ratio of water vapor),

$$\frac{dP}{dt} = \frac{-gPV}{R_d T_v} \tag{1.8}$$

The equations (1.8), (1.7), (1.3), (1.2), and (1.1) provide a simple, closed system of ordinary differential equations which can be numerically integrated forward in time. Furthermore, this model formulation allows the simulation of an arbitrary configuration of initial aerosols, in terms of size, number concentration, and hygroscopicity. Adding additional aerosol size bins is simply accomplished by tracking one additional size bin in the system of ODE's. The important application of this feature is that the model can be configured to simulate both internal or external mixtures of aerosols, or some combination thereof.

1.1.2 Model Implementation and Procedure

The parcel model described in the previous section was implemented using a modern modular and object-oriented software engineering framework. This framework allows the model to be simply configured with myriad initial conditions and aerosol populations. It also enables model components - such as the numerical solver or condensation parameterization - to be swapped and replaced. Most importantly, the use of object-oriented techniques allows the model to be incorporated into frameworks which grossly accelerate the speed at which the model can be evaluated. For instance, although models like the one developed here are relatively cheap to execute, large ensembles of model runs have been limited in scope to several hundred or a thousand runs. However, the framework of this particular parcel model implementation was designed such that it could be run as a black box as part of a massively-parallel ensemble driver.

To run the model, a set of initial conditions needs to be specified, which includes the updraft speed, the parcel's initial temperature, pressure, and supersaturation, and the aerosol population. Given these parameters, the model calculates an initial equilibrium droplet spectrum by computing the equilibrium wet radii of each aerosol. This is calculated numerically from the Kohler equation for each aerosol/proto-droplet, or numerically by employing the typical Kohler theory approximation

$$S \approx \frac{A}{r} - \kappa \frac{r_d^3}{r^3}$$

These wet radii are used as the initial droplet radii in the simulation.

Once the initial conditions have been configured, the model is integrated forward in time with a numerical solver (see `ParcelModel.run()` for more details). The available solvers wrapped here are:

- LSODA(R)
- LSODE
- (C)VODE

During the model integration, the size representing each aerosol bin is allowed to grow via condensation, producing something akin to a moving grid. In the future, a fixed Eulerian grid will likely be implemented in the model for comparison.

1.1.3 Aerosol Population Specification

The model may be supplied with any arbitrary population of aerosols, providing the population can be approximated with a sectional representation. Most commonly, aerosol size distributions are represented with a continuous lognormal distribution,

$$n_N(r) = \frac{dN}{d\ln r} = \frac{N_t}{\sqrt{2\pi}\ln\sigma_g} \exp\left(-\frac{\ln^2(r/\mu_g)}{2\ln^2\sigma_g}\right) \tag{1.9}$$

which can be summarized with the set of three parameters, (N_t, μ_g, σ_g) and correspond, respectively, to the total aerosol number concentration, the geometric mean or number mode radius, and the geometric standard deviation. Complicated multi-modal aerosol distributions can often be represented as the sum of several lognormal distributions. Since the parcel model describes the evolution of a discrete aerosol size spectrum, can be broken into n bins, and the continuous aerosol size distribution approximated by taking the number concentration and size at the geometric mean value in each bin, such that the discrete approximation to the aerosol size distribution becomes

$$n_{N,i}(r_i) = \sum_{i=1}^{n} \frac{N_i}{\sqrt{2\pi}\ln\sigma_g} \exp\left(-\frac{\ln^2(r_i/\mu_g)}{2\ln^2\sigma_g}\right)$$

If no bounds on the size range of r_i is specified, then the model pre-computes n equally-spaced bins over the logarithm of r, and covers the size range $\mu_g/10\sigma_g$ to $10\sigma_g\mu_g$. It is typical to run the model with 200 size bins per aerosol mode. Neither this model nor similar ones exhibit much sensitivity towards the density of the sectional discretization .

Typically, a single value for hygroscopicity, κ is prescribed for each aerosol mode. However, the model tracks a hygroscopicity parameter for each individual size bin, so size-dependent aerosol composition can be incorporated into the aerosol population. This representation of the aerosol population is similar to the external mixing state assumption. An advantage to using this representation is that complex mixing states can be represented by adding various size bins, each with their own number concentration and hygroscopicity.

1.1.4 References

1.2 Installation

1.2.1 From conda

Coming soon!

1.2.2 From PyPI

If you already have all the dependencies satisfied, then you can install the latest release from PyPI by using `pip`:

```
$ pip install pyrcel
```

1.2.3 From source code

To grab and build the latest bleeding-edge version of the model, you should use `pip` and point it to the source code repository on github:

```
$ pip install git+git://github.com/darothen/pyrcel.git
```

This should automatically build the necessary Cython modules and export the code package to your normal package installation directory. If you wish to simply build the code and run it in place, clone the repository, navigate to it in a terminal, and invoke the build command by hand:

```
$ python setup.py build_ext --inplace
```

This should produce the compiled file *parcel_aux.so* in the model package. You can also install the code from the cloned source directory by invoking `pip install` from within it; this is useful if you're updating or modifying the model, since you can install an "editable" package which points directly to the git-monitored code:

```
$ cd path/to/pyrcel/
$ pip install -e .
```

1.2.4 Dependencies

This code was originally written for Python 2.7, and then futurized to Python 3.3+ with hooks for backwards compatibility. By far, the simplest way to run this code is to grab a scientific python distribution, such as Anaconda. This code should work out-of-the box with almost all dependencies filled (exception being numerical solvers) on a recent version (1.2+) of this distribution. To facilitate this, conda environments for Python versions 2.7 and 3.5+ are provided in the `pyrcel/ci` directory.

Necessary dependencies

- Assimulo
- numba
- numpy
- scipy
- pandas

Note: As of version 1.2.0, the model integration components are being re-written and only the CVODE interface is exposed. As such, Assimulo is temporarily a core and required dependency; in the future the other solvers will be re-enabled. You should first try to install Assimulo via conda

```
$ conda install -c conda-forge assimulo
```

since this will automatically take care of obtaining necessary compiled dependencies like sundials. However, for best results you may want to manually install Assimulo, since the conda-forge recipe may default to a sundials/OpenBLAS combination which could degare the performance of the model.

Numerical solver dependencies

- **LSODA** - scipy or odespy
- **VODE**, **LSODE** - odespy
- **CVODE** - Assimulo

Recommended additional packages

- matplotlib
- seaborn
- PyYAML
- xarray

1.2.5 Testing

A nose test-suite is under construction. To check that your model is configured and running correctly, you copy and run the notebook corresponding to the *basic run example*, or run the command-line interface version of the model with the pre-packed simple run case:

```
$ cd path/to/pyrcel/
$ ./run_parcel examples/simple.yml
```

1.2.6 Bugs / Suggestions

The code has an issue tracker on github and I strongly encourage you to note any problems with the model there, such as typos or weird behavior and results. Furthermore, I'm looking for ways to expand and extend the model, so if there

is something you might wish to see added, please note it there or send me an e-mail. The code was written in such a way that it should be trivial to add physics in a modular fashion.

1.3 Example: Activation

In this example, we will study the effect of updraft speed on the activation of a lognormal ammonium sulfate accumulation mode aerosol.

```
# Suppress warnings
import warnings
warnings.simplefilter('ignore')

import pyrcel as pm
import numpy as np

%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
```

First, we indicate the parcel's initial thermodynamic conditions.

```
P0 = 100000. # Pressure, Pa
T0 = 279.    # Temperature, K
S0 = -0.1    # Supersaturation, 1-RH
```

We next define the aerosol distribution to follow the reference simulation from Ghan et al, 2011

```
aer =  pm.AerosolSpecies('ammonium sulfate',
                         pm.Lognorm(mu=0.05, sigma=2.0, N=1000.),
                         kappa=0.7, bins=100)
```

Loop over updraft several velocities in the range 0.1 - 10.0 m/s. We will peform a detailed parcel model calculation, as well as calculations with two activation parameterizations. We will also use an accommodation coefficient of $\alpha_c = 0.1$, following the recommendations of Raatikainen et al (2013).

First, the parcel model calculations:

```
from pyrcel import binned_activation

Vs = np.logspace(-1, np.log10(10,), 11.)[::-1] # 0.1 - 5.0 m/s
accom = 0.1

smaxes, act_fracs = [], []
for V in Vs:
    # Initialize the model
    model = pm.ParcelModel([aer,], V, T0, S0, P0, accom=accom, console=False)
    par_out, aer_out = model.run(t_end=2500., dt=1.0, solver='cvode',
                                 output='dataframes', terminate=True)
    print(V, par_out.S.max())

    # Extract the supersaturation/activation details from the model
    # output
    S_max = par_out['S'].max()
    time_at_Smax = par_out['S'].argmax()
    wet_sizes_at_Smax = aer_out['ammonium sulfate'].ix[time_at_Smax].iloc[0]
    wet_sizes_at_Smax = np.array(wet_sizes_at_Smax.tolist())
```

(continues on next page)

(continued from previous page)

```
    frac_eq, _, _, _ = binned_activation(S_max, T0, wet_sizes_at_Smax, aer)

    # Save the output
    smaxes.append(S_max)
    act_fracs.append(frac_eq)
```

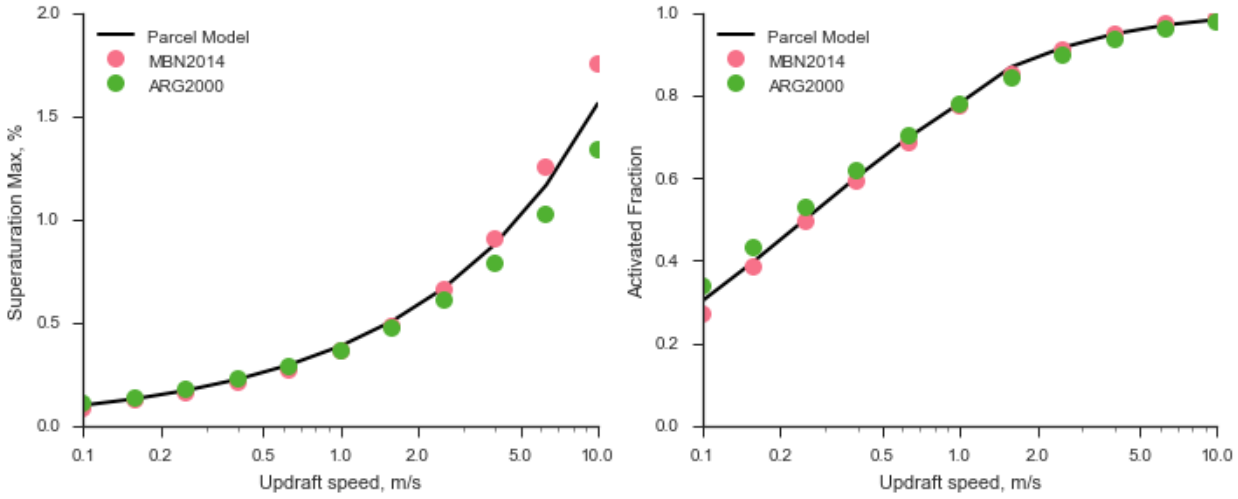
```
[CVode Warning] b'At the end of the first step, there are still some root functions␣
↪identically 0. This warning will not be issued again.'
10.0 0.0156189147154
[CVode Warning] b'At the end of the first step, there are still some root functions␣
↪identically 0. This warning will not be issued again.'
6.3095734448 0.0116683910368
[CVode Warning] b'At the end of the first step, there are still some root functions␣
↪identically 0. This warning will not be issued again.'
3.98107170553 0.00878287310116
[CVode Warning] b'At the end of the first step, there are still some root functions␣
↪identically 0. This warning will not be issued again.'
2.51188643151 0.00664901290831
[CVode Warning] b'At the end of the first step, there are still some root functions␣
↪identically 0. This warning will not be issued again.'
1.58489319246 0.00505644091867
[CVode Warning] b'At the end of the first step, there are still some root functions␣
↪identically 0. This warning will not be issued again.'
1.0 0.00385393398982
[CVode Warning] b'At the end of the first step, there are still some root functions␣
↪identically 0. This warning will not be issued again.'
0.63095734448 0.00293957320198
[CVode Warning] b'At the end of the first step, there are still some root functions␣
↪identically 0. This warning will not be issued again.'
0.398107170553 0.00224028774582
[CVode Warning] b'At the end of the first step, there are still some root functions␣
↪identically 0. This warning will not be issued again.'
0.251188643151 0.00170480101361
[CVode Warning] b'At the end of the first step, there are still some root functions␣
↪identically 0. This warning will not be issued again.'
0.158489319246 0.0012955732509
[CVode Warning] b'At the end of the first step, there are still some root functions␣
↪identically 0. This warning will not be issued again.'
0.1 0.000984803827635
```

Now the activation parameterizations:

```
smaxes_arg, act_fracs_arg = [], []
smaxes_mbn, act_fracs_mbn = [], []

for V in Vs:
    smax_arg, _, afs_arg = pm.arg2000(V, T0, P0, [aer], accom=accom)
    smax_mbn, _, afs_mbn = pm.mbn2014(V, T0, P0, [aer], accom=accom)

    smaxes_arg.append(smax_arg)
    act_fracs_arg.append(afs_arg[0])
    smaxes_mbn.append(smax_mbn)
    act_fracs_mbn.append(afs_mbn[0])
```

Finally, we compile our results into a nice plot for visualization.

```
sns.set(context="notebook", style='ticks')
sns.set_palette("husl", 3)
fig, [ax_s, ax_a] = plt.subplots(1, 2, sharex=True, figsize=(10,4))

ax_s.plot(Vs, np.array(smaxes)*100., color='k', lw=2, label="Parcel Model")
ax_s.plot(Vs, np.array(smaxes_mbn)*100., linestyle='None',
          marker="o", ms=10, label="MBN2014" )
ax_s.plot(Vs, np.array(smaxes_arg)*100., linestyle='None',
          marker="o", ms=10, label="ARG2000" )
ax_s.semilogx()
ax_s.set_ylabel("Superaturation Max, %")
ax_s.set_ylim(0, 2.)

ax_a.plot(Vs, act_fracs, color='k', lw=2, label="Parcel Model")
ax_a.plot(Vs, act_fracs_mbn, linestyle='None',
          marker="o", ms=10, label="MBN2014" )
ax_a.plot(Vs, act_fracs_arg, linestyle='None',
          marker="o", ms=10, label="ARG2000" )
ax_a.semilogx()
ax_a.set_ylabel("Activated Fraction")
ax_a.set_ylim(0, 1.)

plt.tight_layout()
sns.despine()

for ax in [ax_s, ax_a]:
    ax.legend(loc='upper left')
    ax.xaxis.set_ticks([0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0])
    ax.xaxis.set_ticklabels([0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0])
    ax.set_xlabel("Updraft speed, m/s")
```



1.4 Example: Basic Run

In this example, we will setup a simple parcel model simulation containing two aerosol modes. We will then run the model with a 1 m/s updraft, and observe how the aerosol population bifurcates into swelled aerosol and cloud droplets.

```
# Suppress warnings
import warnings
warnings.simplefilter('ignore')

import pyrcel as pm
import numpy as np

%matplotlib inline
import matplotlib.pyplot as plt
```

```
Could not find GLIMDA
```

First, we indicate the parcel's initial thermodynamic conditions.

```
P0 = 77500. # Pressure, Pa
T0 = 274.   # Temperature, K
S0 = -0.02  # Supersaturation, 1-RH (98% here)
```

Next, we define the aerosols present in the parcel. The model itself is agnostic to how the aerosol are specified; it simply expects lists of the radii of wetted aerosol radii, their number concentration, and their hygroscopicity. We can make container objects (:class:`AerosolSpecies`) that wrap all of this information so that we never need to worry about it.

Here, let's construct two aerosol modes:

Mode	κ (hygroscopicity)	Mean size (micron)	Std dev	Number Conc (cm**-3)
sulfate	0.54	0.015	1.6	850
sea salt	1.2	0.85	1.2	10

We'll define each mode using the :class:`Lognorm` distribution packaged with the model.

```
sulfate =  pm.AerosolSpecies('sulfate',
                             pm.Lognorm(mu=0.015, sigma=1.6, N=850.),
                             kappa=0.54, bins=200)
sea_salt = pm.AerosolSpecies('sea salt',
                             pm.Lognorm(mu=0.85, sigma=1.2, N=10.),
                             kappa=1.2, bins=40)
```

The :class:`AerosolSpecies` class automatically computes gridded/binned representations of the size distributions. Let's double check that the aerosol distribution in the model will make sense by plotting the number concentration in each bin.

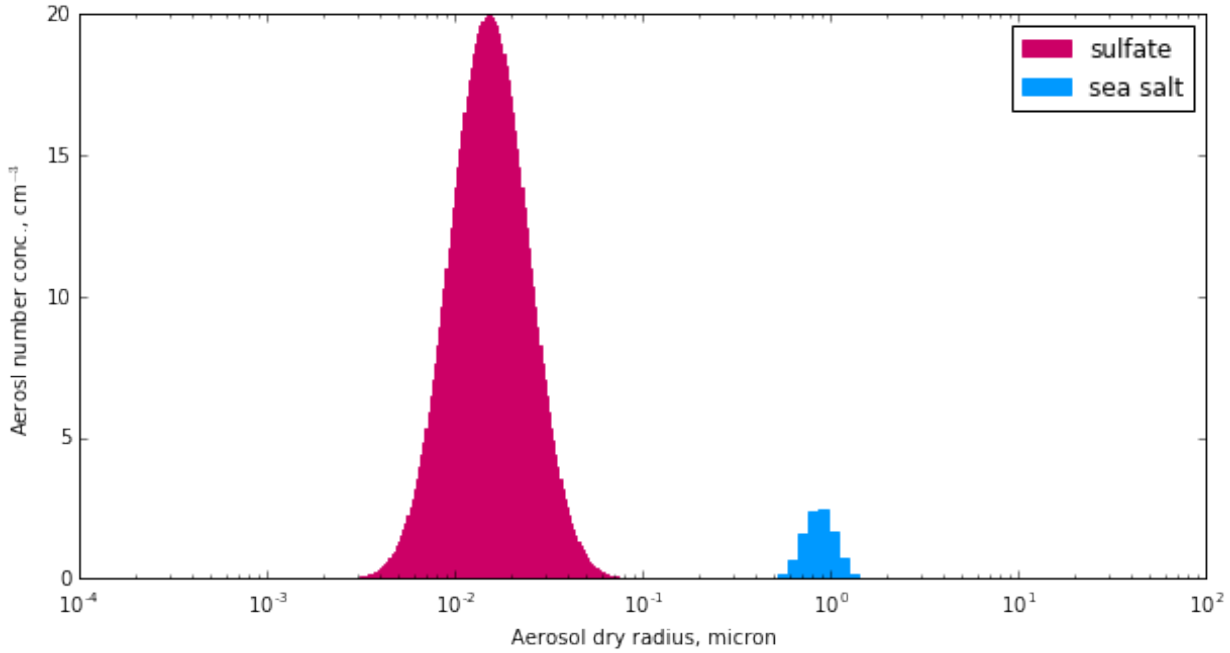
```
fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(111)
ax.grid(False, "minor")

sul_c = "#CC0066"
ax.bar(sulfate.rs[:-1], sulfate.Nis*1e-6, np.diff(sulfate.rs),
        color=sul_c, label="sulfate", edgecolor="#CC0066")
sea_c = "#0099FF"
ax.bar(sea_salt.rs[:-1], sea_salt.Nis*1e-6, np.diff(sea_salt.rs),
        color=sea_c, label="sea salt", edgecolor="#0099FF")
ax.semilogx()

ax.set_xlabel("Aerosol dry radius, micron")
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel("Aerosl number conc., cm$^{-3}$")
ax.legend(loc='upper right')
```

```
<matplotlib.legend.Legend at 0x10f4baeb8>
```



Actually running the model is very straightforward, and involves just two steps:

1. Instantiate the model by creating a :class:`ParcelModel` object.
2. Call the model's :method:`run` method.

For convenience this process is encoded into several routines in the `driver` file, including both a single-strategy routine and an iterating routine which adjusts the the timestep and numerical tolerances if the model crashes. However, we can illustrate the simple model running process here in case you wish to develop your own scheme for running the model.

```
initial_aerosols = [sulfate, sea_salt]
V = 1.0 # updraft speed, m/s

dt = 1.0 # timestep, seconds
t_end = 250./V # end time, seconds... 250 meter simulation

model = pm.ParcelModel(initial_aerosols, V, T0, S0, P0, console=False, accom=0.3)
parcel_trace, aerosol_traces = model.run(t_end, dt, solver='cvode')
```

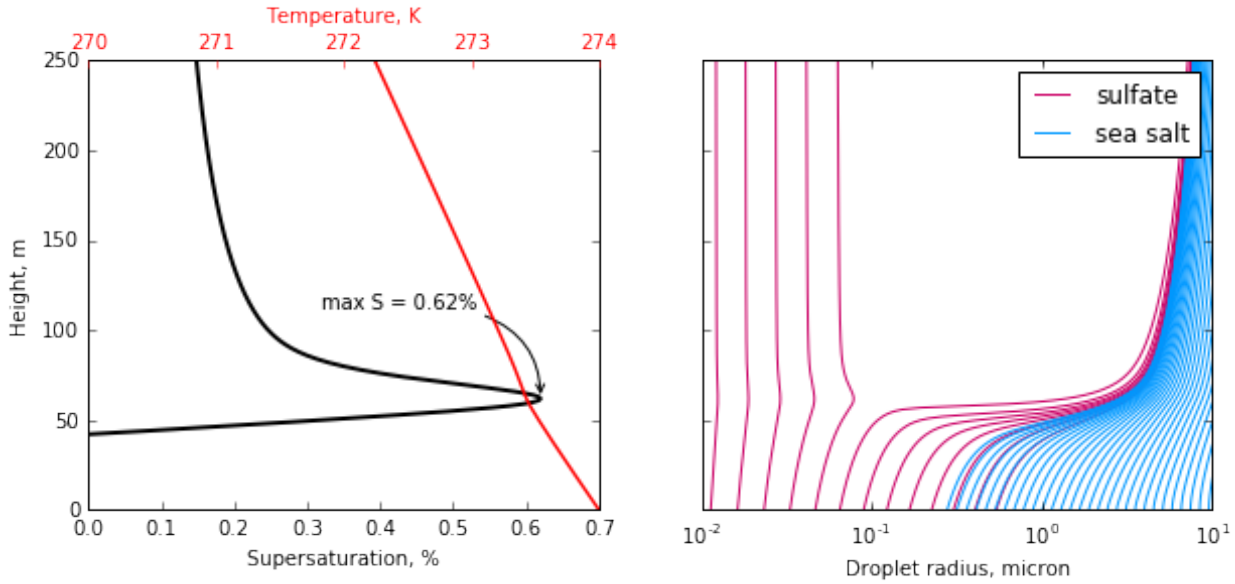
If `console` is set to `True`, then some basic debugging output will be written to the terminal, including the initial equilibrium droplet size distribution and some numerical solver diagnostics. The model output can be customized; by default, we get a DataFrame and a Panel of the parcel state vector and aerosol bin sizes as a function of time (and height). We can use this to visualize the simulation results, like in the package's README.

```
fig, [axS, axA] = plt.subplots(1, 2, figsize=(10, 4), sharey=True)

axS.plot(parcel_trace['S']*100., parcel_trace['z'], color='k', lw=2)
```

(continues on next page)

(continued from previous page)

```
axT = axS.twiny()
axT.plot(parcel_trace['T'], parcel_trace['z'], color='r', lw=1.5)

Smax = parcel_trace['S'].max()*100
z_at_smax = parcel_trace['z'].ix[parcel_trace['S'].argmax()]
axS.annotate("max S = %0.2f%%" % Smax,
             xy=(Smax, z_at_smax),
             xytext=(Smax-0.3, z_at_smax+50.),
             arrowprops=dict(arrowstyle="->", color='k',
                             connectionstyle='angle3,angleA=0,angleB=90'),
             zorder=10)

axS.set_xlim(0, 0.7)
axS.set_ylim(0, 250)

axT.set_xticks([270, 271, 272, 273, 274])
axT.xaxis.label.set_color('red')
axT.tick_params(axis='x', colors='red')

axS.set_xlabel("Supersaturation, %")
axT.set_xlabel("Temperature, K")
axS.set_ylabel("Height, m")

sulf_array = aerosol_traces['sulfate'].values
sea_array = aerosol_traces['sea salt'].values

ss = axA.plot(sulf_array[:, ::10]*1e6, parcel_trace['z'], color=sul_c,
         label="sulfate")
sa = axA.plot(sea_array*1e6, parcel_trace['z'], color=sea_c, label="sea salt")
axA.semilogx()
axA.set_xlim(1e-2, 10.)
axA.set_xticks([1e-2, 1e-1, 1e0, 1e1], [0.01, 0.1, 1.0, 10.0])
axA.legend([ss[0], sa[0]], ['sulfate', 'sea salt'], loc='upper right')
axA.set_xlabel("Droplet radius, micron")

for ax in [axS, axA, axT]:
    ax.grid(False, 'both', 'both')
```



In this simple example, the sulfate aerosol population bifurcated into interstitial aerosol and cloud droplets, while the entire sea salt population activated. A peak supersaturation of about 0.63% was reached a few meters above cloud base, where the ambient relative humidity hit 100%.

How many CDNC does this translate into? We can call upon helper methods from the `activation` package to perform these calculations for us:

```
from pyrcel import binned_activation

sulf_trace = aerosol_traces['sulfate']
sea_trace = aerosol_traces['sea salt']

ind_final = int(t_end/dt) - 1

T = parcel_trace['T'].iloc[ind_final]
eq_sulf, kn_sulf, alpha_sulf, phi_sulf = \
    binned_activation(Smax/100, T, sulf_trace.iloc[ind_final],  sulfate)
eq_sulf *= sulfate.total_N

eq_sea, kn_sea, alpha_sea, phi_sea = \
    binned_activation(Smax/100, T, sea_trace.iloc[ind_final], sea_salt)
eq_sea *= sea_salt.total_N

print("  CDNC(sulfate) = {:3.1f}".format(eq_sulf))
print(" CDNC(sea salt) = {:3.1f}".format(eq_sea))
print("-------------------------")
print("          total = {:3.1f} / {:3.0f} ~ act frac = {:1.2f}".format(
      eq_sulf+eq_sea,
      sea_salt.total_N+sulfate.total_N,
      (eq_sulf+eq_sea)/(sea_salt.total_N+sulfate.total_N)
))
```

```
  CDNC(sulfate) = 146.9
 CDNC(sea salt) = 10.0
-------------------------
          total = 156.9 / 860 ~ act frac = 0.18
```

1.5 Parcel Model Details

Below is the documentation for the parcel model, which is useful for debugging and development. For a higher-level overview, see the *scientific description*.

1.5.1 Implementation

class pyrcel.**ParcelModel**(*aerosols*, *V*, *T0*, *S0*, *P0*, *console=False*, *accom=1.0*, *truncate_aerosols=False*)

Wrapper class for instantiating and running the parcel model.

The parcel model has been implemented in an object-oriented format to facilitate easy extensibility to different aerosol and meteorological conditions. A typical use case would involve specifying the initial conditions such as:

```
>>> import pyrcel as pm
>>> P0 = 80000.
>>> T0 = 283.15
>>> S0 = 0.0
>>> V = 1.0
>>> aerosol1 = pm.AerosolSpecies('sulfate',
...                              Lognorm(mu=0.025, sigma=1.3, N=2000.),
...                              bins=200, kappa=0.54)
>>> initial_aerosols = [aerosol1, ]
>>> z_top = 50.
>>> dt = 0.01
```

which initializes the model with typical conditions at the top of the boundary layer (800 hPa, 283.15 K, 100% Relative Humidity, 1 m/s updraft), and a simple sulfate aerosol distribution which will be discretized into 200 size bins to track. Furthermore the model was specified to simulate the updraft for 50 meters (*z_top*) and use a time-discretization of 0.01 seconds. This timestep is used in the model output – the actual ODE solver will generally calculate the trace of the model at many more times.

Running the model and saving the output can be accomplished by invoking:

```
>>> model = pm.ParcelModel(initial_aerosols, V, T0, S0, P0)
>>> par_out, aer_out = pm.run(z_top, dt)
```

This will yield *par_out*, a `pandas.DataFrame` containing the meteorological conditions in the parcel, and *aerosols*, a dictionary of `DataFrame` objects for each species in *initial_aerosols* with the appropriately tracked size bins and their evolution over time.

See also:

_setup_run companion routine which computes equilibrium droplet sizes and sets the model's state vectors.

Attributes

V, T0, S0, P0, aerosols [floats] Initial parcel settings (see **Parameters**).

_r0s [array_like of floats] Initial equilibrium droplet sizes.

_r_drys [array_like of floats] Dry radii of aerosol population.

_kappas [array_like of floats] Hygroscopicity of each aerosol size.

_Nis [array_like of floats] Number concentration of each aerosol size.

_nr [int] Number of aerosol sizes tracked in model.

_model_set [boolean] Flag indicating whether or not at any given time the model initialization/equilibration routine has been run with the current model settings.

_y0 [array_like] Initial state vector.

Methods

run(t_end, dt, max_steps=1000, solver="odeint", output_fmt="dataframes", terminate=False, solver_args={})	Execute model simulation.
set_initial_conditions(V=None, T0=None, S0=None, P0=None, aerosols=None)	Re-initialize a model simulation in order to run it.

run(*self*, *t_end*, *output_dt=1.0*, *solver_dt=None*, *max_steps=1000*, *solver='odeint'*, *output_fmt='dataframes'*, *terminate=False*, *terminate_depth=100.0*, ***solver_args*)
Run the parcel model simulation.

Once the model has been instantiated, a simulation can immediately be performed by invoking this method. The numerical details underlying the simulation and the times over which to integrate can be flexibly set here.

Time – The user must specify two timesteps: *output_dt*, which is the timestep between output snapshots of the state of the parcel model, and *solver_dt*, which is the the interval of time before the ODE integrator is paused and re-started. It's usually okay to use a very large *solver_dt*, as *output_dt* can be interpolated from the simulation. In some cases though a small *solver_dt* could be useful to force the solver to use smaller internal timesteps.

Numerical Solver – By default, the model will use the *odeint* wrapper of LSODA shipped by default with scipy. Some fine-tuning of the solver tolerances is afforded here through the *max_steps*. For other solvers, a set of optional arguments *solver_args* can be passed.

Solution Output – Several different output formats are available by default. Additionally, the output arrays are saved with the *ParcelModel* instance so they can be used later.

Parameters

t_end [float] Total time over interval over which the model should be integrated

output_dt [float] Timestep intervals to report model output.

solver_dt [float] Timestep interval for calling solver integration routine.

max_steps [int] Maximum number of steps allowed by solver to satisfy error tolerances per timestep.

solver [{'odeint', 'lsoda', 'lsode', 'vode', cvode'}] Choose which numerical solver to use: * *'odeint'*: LSODA implementation from ODEPACK via

SciPy's integrate module

- *'lsoda'*: LSODA implementation from ODEPACK via odespy
- *'lsode'*: LSODE implementation from ODEPACK via odespy
- *'vode'* : VODE implementation from ODEPACK via odespy
- *'cvode'* : CVODE implementation from Sundials via Assimulo
- *'lsodar'* : LSODAR implementation from Sundials via Assimulo

output_fmt [str, one of {'dataframes', 'arrays', 'smax'}] Choose format of solution output.

terminate [boolean] End simulation at or shortly after a maximum supersaturation has been achieved

terminate_depth [float, optional (default=100.)] Additional depth (in meters) to integrate after termination criterion eached.

Returns

DataFrames, array, or float Depending on what was passed to the *output* argument, different types of data might be returned:

- **'dataframes': (default) will process the output into** two pandas DataFrames - the first one containing profiles of the meteorological quantities tracked in the model, and the second a dictionary of DataFrames with one for each AerosolSpecies, tracking the growth in each bin for those species.
- **'arrays': will return the raw output from the solver** used internally by the parcel model - the state vector *y* and the evaluated timesteps converted into height coordinates.
- **'smax': will only return the maximum supersaturation** value achieved in the simulation.

Raises

ParcelModelError The parcel model failed to complete successfully or failed to initialize.

See also:

der right-hand side derivative evaluated during model integration.

set_initial_conditions(*self*, *V=None*, *T0=None*, *S0=None*, *P0=None*, *aerosols=None*)

Set the initial conditions and parameters for a new parcel model run without having to create a new *ParcelModel* instance.

Based on the aerosol population which has been stored in the model, this method will finish initializing the model. This has three major parts:

1. concatenate the aerosol population information (their dry radii, hygroscopicities, etc) into single arrays which can be placed into the state vector for forward integration.
2. Given the initial ambient water vapor concentration (computed from the temperature, pressure, and supersaturation), determine how much water must already be coated on the aerosol particles in order for their size to be in equilibrium.
3. Set-up the state vector with these initial conditions.

Once the state vector has been set up, the setup routine will record attributes in the parent instance of the *ParcelModel*.

Parameters

V, T0, S0, P0 [floats] The updraft speed and initial temperature (K), pressure (Pa), supersaturation (percent, with 0.0 = 100% RH).

aerosols [array_like sequence of `AerosolSpecies`] The aerosols contained in the parcel.

Raises

ParcelModelError If an equilibrium droplet size distribution could not be calculated.

Notes

The actual setup occurs in the private method *_setup_run()*; this method is simply an interface that can be used to modify an existing `ParcelModel`.

1.5.2 Derivative Equation

parcel.**parcel_ode_sys**(*y*, *t*, *nr*, *r_drys*, *Nis*, *V*, *kappas*, *accom*)

Calculates the instantaneous time-derivative of the parcel model system.

Given a current state vector *y* of the parcel model, computes the tendency of each term including thermodynamic (pressure, temperature, etc) and aerosol terms. The basic aerosol properties used in the model must be passed along with the state vector (i.e. if being used as the callback function in an ODE solver).

Parameters

y [array_like]

Current state of the parcel model system,

- y[0] = altitude, m
- y[1] = Pressure, Pa
- y[2] = temperature, K
- y[3] = water vapor mass mixing ratio, kg/kg
- y[4] = cloud liquid water mass mixing ratio, kg/kg
- y[5] = cloud ice water mass mixing ratio, kg/kg
- y[6] = parcel supersaturation
- y[7:] = aerosol bin sizes (radii), m

t [float] Current simulation time, in seconds.

nr [Integer] Number of aerosol radii being tracked.

r_drys [array_like] Array recording original aerosol dry radii, m.

Nis [array_like] Array recording aerosol number concentrations, 1/(m**3).

V [float] Updraft velocity, m/s.

kappas [array_like] Array recording aerosol hygroscopicities.

accom [float, optional (default=:const:*constants.ac*)] Condensation coefficient.

Returns

x [array_like] Array of shape (‘‘nr‘‘+7,) containing the evaluated parcel model instaneous derivative.

Notes

This function is implemented using numba; it does not need to be just-in- time compiled in order ot function correctly, but it is set up ahead of time so that the internal loop over each bin growth term is parallelized.

1.6 Reference

1.6.1 Main Parcel Model

The core of the model has its own documentation page, which you can access *here*.

ParcelModel(aerosols, V, T0, S0, P0[, ...])	Wrapper class for instantiating and running the parcel model.

pyrcel.ParcelModel

class pyrcel.**ParcelModel**(*aerosols*, *V*, *T0*, *S0*, *P0*, *console=False*, *accom=1.0*, *truncate_aerosols=False*)

Wrapper class for instantiating and running the parcel model.

The parcel model has been implemented in an object-oriented format to facilitate easy extensibility to different aerosol and meteorological conditions. A typical use case would involve specifying the initial conditions such as:

```
>>> import pyrcel as pm
>>> P0 = 80000.
>>> T0 = 283.15
>>> S0 = 0.0
>>> V = 1.0
>>> aerosol1 = pm.AerosolSpecies('sulfate',
...                              Lognorm(mu=0.025, sigma=1.3, N=2000.),
...                              bins=200, kappa=0.54)
>>> initial_aerosols = [aerosol1, ]
>>> z_top = 50.
>>> dt = 0.01
```

which initializes the model with typical conditions at the top of the boundary layer (800 hPa, 283.15 K, 100% Relative Humidity, 1 m/s updraft), and a simple sulfate aerosol distribution which will be discretized into 200 size bins to track. Furthermore the model was specified to simulate the updraft for 50 meters (*z_top*) and use a time-discretization of 0.01 seconds. This timestep is used in the model output – the actual ODE solver will generally calculate the trace of the model at many more times.

Running the model and saving the output can be accomplished by invoking:

```
>>> model = pm.ParcelModel(initial_aerosols, V, T0, S0, P0)
>>> par_out, aer_out = pm.run(z_top, dt)
```

This will yield *par_out*, a `pandas.DataFrame` containing the meteorological conditions in the parcel, and *aerosols*, a dictionary of `DataFrame` objects for each species in *initial_aerosols* with the appropriately tracked size bins and their evolution over time.

See also:

_setup_run companion routine which computes equilibrium droplet sizes and sets the model's state vectors.

Attributes

V, T0, S0, P0, aerosols [floats] Initial parcel settings (see **Parameters**).

_r0s [array_like of floats] Initial equilibrium droplet sizes.

_r_drys [array_like of floats] Dry radii of aerosol population.

_kappas [array_like of floats] Hygroscopicity of each aerosol size.

_Nis [array_like of floats] Number concentration of each aerosol size.

_nr [int] Number of aerosol sizes tracked in model.

_model_set [boolean] Flag indicating whether or not at any given time the model initialization/equilibration routine has been run with the current model settings.

_y0 [array_like] Initial state vector.

Methods

run(t_end, dt, max_steps=1000, solver="odeint", output_fmt="dataframes", terminate=False, solver_args={})	Execute model simulation.
set_initial_conditions(V=None, T0=None, S0=None, P0=None, aerosols=None)	Re-initialize a model simulation in order to run it.

__init__(*self*, *aerosols*, *V*, *T0*, *S0*, *P0*, *console=False*, *accom=1.0*, *truncate_aerosols=False*)

Initialize the parcel model.

Parameters

aerosols [array_like sequence of `AerosolSpecies`] The aerosols contained in the parcel.

V, T0, S0, P0 [floats] The updraft speed and initial temperature (K), pressure (Pa), supersaturation (percent, with 0.0 = 100% RH).

console [boolean, optional] Enable some basic debugging output to print to the terminal.

accom [float, optional (default=:const:*constants.ac*)] Condensation coefficient

truncate_aerosols [boolean, optional (default=**False**)] Eliminate extremely small aerosol which will cause numerical problems

Methods

__init__(self, aerosols, V, T0, S0, P0[, ...])	Initialize the parcel model.
run(self, t_end[, output_dt, solver_dt, ...])	Run the parcel model simulation.
`save`(self[, filename, format, other_dfs])	
set_initial_conditions(self[, V, T0, S0, ...])	Set the initial conditions and parameters for a new parcel model run without having to create a new *ParcelModel* instance.
`write_csv`(parcel_data, aerosol_data[, ...])	Write output to CSV files.
`write_summary`(self, parcel_data, ...)	Write a quick and dirty summary of given parcel model output to the terminal.

1.6.2 Driver Tools

Utilities for driving sets of parcel model integration strategies.

Occasionally, a pathological set of input parameters to the parcel model will really muck up the ODE solver's ability to integrate the model. In that case, it would be nice to quietly adjust some of the numerical parameters for the ODE solver and re-submit the job. This module includes a workhorse function *iterate_runs()* which can serve this

purpose and can serve as an example for more complex integration strategies. Alternatively, :func:‘run_model‘is a useful shortcut for building/running a model and snagging its output.

run_model(V, initial_aerosols, T, P, dt[, ...])	Setup and run the parcel model with given solver configuration.
iterate_runs(V, initial_aerosols, T, P[, ...])	Iterate through several different strategies for integrating the parcel model.

pyrcel.driver.run_model

pyrcel.driver.**run_model**(*V*, *initial_aerosols*, *T*, *P*, *dt*, *S0=-0.0*, *max_steps=1000*, *t_end=500.0*, *solver='lsoda'*, *output_fmt='smax'*, *terminate=False*, *solver_kws=None*, *model_kws=None*)

Setup and run the parcel model with given solver configuration.

Parameters

V, T, P [float] Updraft speed and parcel initial temperature and pressure.

S0 [float, optional, default 0.0] Initial supersaturation, as a percent. Defaults to 100% relative humidity.

initial_aerosols [array_like of `AerosolSpecies`] Set of aerosol populations contained in the parcel.

dt [float] Solver timestep, in seconds.

max_steps [int, optional, default 1000] Maximum number of steps per solver iteration. Defaults to 1000; setting excessively high could produce extremely long computation times.

t_end [float, optional, default 500.0] Model time in seconds after which the integration will stop.

solver [string, optional, default 'lsoda'] Alias of which solver to use; see `Integrator` for all options.

output_fmt [string, optional, default 'smax'] Alias indicating which output format to use; see `ParcelModel` for all options.

solver_kws, model_kws [dicts, optional] Additional arguments/configuration to pass to the numerical integrator or model.

Returns

Smax [(user-defined)] Output from parcel model simulation based on user-specified *output_fmt* argument. See `ParcelModel` for details.

Raises

ParcelModelError If the model fails to initialize or breaks during runtime.

pyrcel.driver.iterate_runs

pyrcel.driver.**iterate_runs**(*V*, *initial_aerosols*, *T*, *P*, *S0=-0.0*, *dt=0.01*, *dt_iters=2*, *t_end=500.0*, *max_steps=500*, *output_fmt='smax'*, *fail_easy=True*)

Iterate through several different strategies for integrating the parcel model.

As long as *fail_easy* is set to *False*, the strategies this method implements are:

1. **CVODE** with a 10 second time limit and 2000 step limit.
2. **LSODA** with up to *dt_iters* iterations, where the timestep *dt* is halved each time.

3. **LSODE** with coarse tolerance and the original timestep.

If these strategies all fail, the model will print a statement indicating such and return either -9999 if *output_fmt* was 'smax', or an empty array or DataFrame accordingly.

Parameters

V, T, P [float] Updraft speed and parcel initial temperature and pressure.

S0 [float, optional, default 0.0] Initial supersaturation, as a percent. Defaults to 100% relative humidity.

initial_aerosols [array_like of `AerosolSpecies`] Set of aerosol populations contained in the parcel.

dt [float] Solver timestep, in seconds.

dt_iters [int, optional, default 2] Number of times to halve *dt* when attempting **LSODA** solver.

max_steps [int, optional, default 1000] Maximum number of steps per solver iteration. Defaults to 1000; setting excessively high could produce extremely long computation times.

t_end [float, optional, default 500.0] Model time in seconds after which the integration will stop.

output [string, optional, default 'smax'] Alias indicating which output format to use; see `ParcelModel` for all options.

fail_easy [boolean, optional, default *True*] If *True*, then stop after the first strategy (**CVODE**)

Returns

Smax [(user-defined)] Output from parcel model simulation based on user-specified *output* argument. See `ParcelModel` for details.

1.6.3 Thermodynamics/Kohler Theory

Aerosol/atmospheric thermodynamics functions.

The following sets of functions calculate useful thermodynamic quantities that arise in aerosol-cloud studies. Where possible, the source of the parameterization for each function is documented.

dv(T, r, P[, accom])	Diffusivity of water vapor in air, modified for non-continuum effects.
ka(T, rho, r)	Thermal conductivity of air, modified for non-continuum effects.
rho_air(T, P[, RH])	Density of moist air with a given relative humidity, temperature, and pressure.
es(T_c)	Calculates the saturation vapor pressure over water for a given temperature.
sigma_w(T)	Surface tension of water for a given temperature.
Seq(r, r_dry, T, kappa)	-Kohler theory equilibrium saturation over aerosol.
Seq_approx(r, r_dry, T, kappa)	Approximate -Kohler theory equilibrium saturation over aerosol.
kohler_crit(T, r_dry, kappa[, approx])	Critical radius and supersaturation of an aerosol particle.
critical_curve(T, r_a, r_b, kappa[, approx])	Calculates curves of critical radii and supersaturations for aerosol.

pyrcel.thermo.dv

pyrcel.thermo.**dv**(*T*, *r*, *P*, *accom=1.0*)

Diffusivity of water vapor in air, modified for non-continuum effects.

The diffusivity of water vapor in air as a function of temperature and pressure is given by

$$D_v = 10^{-4} \frac{0.211}{P} \left(\frac{T}{273} \right)^{1.94} \quad (1.10)$$

where P is in atm [SP2006]. Aerosols much smaller than the mean free path of the air surrounding them ($K_n >> 1$) perturb the flow around them moreso than larger particles, which affects this value. We account for corrections to D_v in the non-continuum regime via the parameterization

$$D_v' = \frac{D_v}{1 + \frac{D_v}{\alpha_c r} \left(\frac{2\pi M_w}{RT} \right)^{1/2}} \quad (1.11)$$

where α_c is the condensation coefficient (`constants.ac`).

Parameters

T [float] ambient temperature of air surrounding droplets, K

r [float] radius of aerosol/droplet, m

P [float] ambient pressure of surrounding air, Pa

accom [float, optional (default=:const:*constants.ac*)] condensation coefficient

Returns

float $D_v'(T, r, P)$ in m^2/s

See also:

dv_cont neglecting correction for non-continuum effects

References

[SP2006]

pyrcel.thermo.ka

pyrcel.thermo.**ka**(*T*, *rho*, *r*)

Thermal conductivity of air, modified for non-continuum effects.

The thermal conductivity of air is given by

$$k_a = 10^{-3}(4.39 + 0.071T) \quad (1.12)$$

Modification to account for non-continuum effects (small aerosol/droplet size) yields the equation

$$k_a' = \frac{k_a}{1 + \frac{k_a}{\alpha_t r_p \rho C_p} \frac{2\pi M_a}{RT}^{1/2}} \quad (1.13)$$

where α_t is a thermal accommodation coefficient (`constants.at`).

Parameters

T [float] ambient air temperature, K

rho [float] ambient air density, kg/m^3

r [float] droplet radius, m

Returns

float $k_a'(T, \rho, r)$ in J/m/s/K

See also:

`ka_cont` neglecting correction for non-continuum effects

References

[SP2006]

pyrcel.thermo.rho_air

`pyrcel.thermo.`**`rho_air`**(*T*, *P*, *RH=1.0*)

Density of moist air with a given relative humidity, temperature, and pressure.

Uses the traditional formula from the ideal gas law (3.41)[Petty2006].

$$\rho_a = \frac{P}{R_d T_v} \quad (1.14)$$

where $T_v = T(1 + 0.61w)$ and w is the water vapor mixing ratio.

Parameters

T [float] ambient air temperature, K

P [float] ambient air pressure, Pa

RH [float, optional (default=1.0)] relative humidity, decimal

Returns

float ρ_a in kg m**-3

References

[Petty2006]

pyrcel.thermo.es

pyrcel.thermo.**es**(*T_c*)

Calculates the saturation vapor pressure over water for a given temperature.

Uses an empirical fit [Bolton1980], which is accurate to 0.1% over the temperature range $-30^oC \leq T \leq 35^oC$,

$$e_s(T) = 611.2 \exp\left(\frac{17.67T}{T + 243.5}\right) \quad (1.15)$$

where e_s is in Pa and T is in degrees C.

Parameters

T_c [float] ambient air temperature, degrees C

Returns

float $e_s(T)$ in Pa

References

[Bolton1980], [RY1989]

pyrcel.thermo.sigma_w

pyrcel.thermo.**sigma_w**(*T*)

Surface tension of water for a given temperature.

$$\sigma_w = 0.0761 - 1.55 \times 10^{-4}(T - 273.15) \quad (1.16)$$

Parameters

T [float] ambient air temperature, degrees K

Returns

float $\sigma_w(T)$ in J/m^2

pyrcel.thermo.Seq

pyrcel.thermo.**Seq**(*r*, *r_dry*, *T*, *kappa*)

-Kohler theory equilibrium saturation over aerosol.

Calculates the equilibrium supersaturation (relative to 100% RH) over an aerosol particle of given dry/wet radius and of specified hygroscopicity bathed in gas at a particular temperature

Following the technique of [PK2007], classical Kohler theory can be modified to account for the hygroscopicity of an aerosol particle using a single parameter, κ. The modified theory predicts that the supersaturation with respect to a given aerosol particle is,

$$S_{\text{eq}} = a_w \exp\left(\frac{2\sigma_w M_w}{RT\rho_w r}\right)$$
$$a_w = \left(1 + \kappa\left(\frac{{r_d}^3}{r}\right)\right)^{-1}$$

with the relevant thermodynamic properties of water defined elsewhere in this module, r_d is the particle dry radius (`r_dry`), r is the radius of the droplet containing the particle (`r`), T is the temperature of the environment (`T`), and κ is the hygroscopicity parameter of the particle (`kappa`).

Parameters

r [float] droplet radius, m

r_dry [float] dry particle radius, m

T [float] ambient air temperature, K

kappa: float particle hygroscopicity parameter

Returns

float S_{eq} for the given aerosol/droplet system

See also:

Seq_approx compute equilibrium supersaturation using an approximation
kohler_crit compute critical radius and equilibrium supersaturation

References

[PK2007]

pyrcel.thermo.Seq_approx

pyrcel.thermo.**Seq_approx**(*r*, *r_dry*, *T*, *kappa*)

Approximate -Kohler theory equilibrium saturation over aerosol.

Calculates the equilibrium supersaturation (relative to 100% RH) over an aerosol particle of given dry/wet radius and of specified hygroscopicity bathed in gas at a particular temperature, using a simplified expression derived by Taylor-expanding the original equation,

$$S_{\text{eq}} = \frac{2\sigma_w M_w}{RT\rho_w r} - \kappa\frac{r_d^3}{r^3}$$

which is valid when the equilibrium supersaturation is small, i.e. in most terrestrial atmosphere applications.

Parameters

r [float] droplet radius, m

r_dry [float] dry particle radius, m

T [float] ambient air temperature, K

kappa: float particle hygroscopicity parameter

Returns

float S_{eq} for the given aerosol/droplet system

See also:

Seq compute equilibrium supersaturation using full theory
kohler_crit compute critical radius and equilibrium supersaturation

pyrcel.thermo.kohler_crit

pyrcel.thermo.**kohler_crit**(*T*, *r_dry*, *kappa*, *approx=False*)

Critical radius and supersaturation of an aerosol particle.

The critical size of an aerosol particle corresponds to the maximum equilibrium supersaturation achieved on its Kohler curve. If a particle grows beyond this size, then it is said to "activate", and will continue to freely grow even if the environmental supersaturation decreases.

This function computes the critical size and and corresponding supersaturation for a given aerosol particle. Typically, it will analyze *Seq()* for the given particle and numerically compute its inflection point. However, if the `approx` flag is passed, then it will compute the analytical critical point for the approximated kappa-Kohler equation.

Parameters

T [float] ambient air temperature, K

r_dry [float] dry particle radius, m

kappa [float] particle hygroscopicity parameter

approx [boolean, optional (default=False)] use the approximate kappa-kohler equation

Returns

(r_crit, s_crit) [tuple of floats] Tuple of $(r_{\text{crit}}, S_{\text{crit}})$, the critical radius (m) and supersaturation of the aerosol droplet.

See also:

Seq equilibrium supersaturation calculation

pyrcel.thermo.critical_curve

pyrcel.thermo.**critical_curve**(*T*, *r_a*, *r_b*, *kappa*, *approx=False*)

Calculates curves of critical radii and supersaturations for aerosol.

Calls *kohler_crit()* for values of `r_dry` between `r_a` and `r_b` to calculate how the critical supersaturation changes with the dry radius for a particle of specified `kappa`

Parameters

T [float] ambient air temperature, K

r_a, r_b [floats] left/right bounds of parcel dry radii, m

kappa [float] particle hygroscopicity parameter

Returns

rs, rcrits, scrits [np.ndarrays] arrays containing particle dry radii (between `r_a` and `r_b`) and their corresponding criticall wet radii and supersaturations

See also:

kohler_crit	critical supersaturation calculation

1.6.4 Aerosols

Container class for encapsulating data about aerosol size distributions.

AerosolSpecies(species, distribution, kappa)	Container class for organizing aerosol metadata.

pyrcel.aerosol.AerosolSpecies

class pyrcel.aerosol.**AerosolSpecies**(*species*, *distribution*, *kappa*, *rho=None*, *mw=None*, *bins=None*, *r_min=None*, *r_max=None*)

Container class for organizing aerosol metadata.

To allow flexibility with how aerosols are defined in the model, this class is meant to act as a wrapper to contain metadata about aerosols (their species name, etc), their chemical composition (particle mass, hygroscopicity, etc), and the particular size distribution chosen for the initial dry aerosol. Because the latter could be very diverse - for instance, it might be desired to have a monodisperse aerosol population, or a bin representation of a canonical size distribution - the core of this class is designed to take those representations and homogenize them for use in the model.

To construct an *AerosolSpecies*, only the metadata (`species` and `kappa`) and the size distribution needs to be specified. The size distribution (`distribution`) can be an instance of `Lognorm`, as long as an extra parameter `bins`, which is an integer representing how many bins into which the distribution should be divided, is also passed to the constructor. In this case, the constructor will figure out how to slice the size distribution to calculate all the aerosol dry radii and their number concentrations. If `r_min` and `r_max` are supplied, then the size range of the aerosols will be bracketed; else, the supplied `distribution` will contain a shape parameter or other bounds to use.

Alternatively, a `dict` can be passed as `distribution` where that slicing has already occurred. In this case, *distribution* must have 2 keys: `r_drys` and `Nis`. Each of the values stored to those keys should fit the attribute descriptors above (although they don't need to be arrays - they can be any iterable.)

Parameters

species [string] Name of aerosol species.

distribution [{ LogNorm, MultiLogNorm, dict }] Representation of aerosol size distribution.

kappa [float] Hygroscopicity of species.

rho [float, optional] Density of dry aerosol material, kg m**-3.

mw [float, optional] Molecular weight of dry aerosol material, kg/mol.

bins [int] Number of bins in discretized size distribution.

Examples

Constructing sulfate aerosol with a specified lognormal distribution -

```
>>> aerosol1 = AerosolSpecies('(NH4)2SO4', Lognorm(mu=0.05, sigma=2.0, N=300.),
...                           bins=200, kappa=0.6)
```

Constructing a monodisperse sodium chloride distribution -

```
>>> aerosol2 = AerosolSpecies('NaCl', {'r_drys': [0.25, ], 'Nis': [1000.0, ]},
...                           kappa=0.2)
```

Warning: Throws a `ValueError` if an unknown type of `distribution` is passed to the constructor, or if *bins* isn't present when `distribution` is an instance of `Lognorm`.

Attributes

nr [float] Number of sizes tracked for this aerosol.

r_drys [array of floats of length `nr`] Dry radii of each representative size tracked for this aerosol, m.

rs [array of floats of length `nr + 1`] Edges of bins in discretized aerosol distribution representation, m.

Nis [array of floats of length `nr`] Number concentration of aerosol of each representative size, m**-3.

total_N [float] Total number concentration of aerosol in this species, cm**-3.

__init__(*self*, *species*, *distribution*, *kappa*, *rho=None*, *mw=None*, *bins=None*, *r_min=None*, *r_max=None*)
Initialize self. See help(type(self)) for accurate signature.

stats(*self*)
Compute useful statistics about this aerosol's size distribution.

Returns

dict Inherits the values from the `distribution`, and if `rho` was provided, adds some statistics about the mass and mass-weighted properties.

Raises

ValueError If the stored `distribution` does not implement a `stats()` function.

The following are utility functions which might be useful in studying and manipulating aerosol distributions for use in the *model* or activation routines.

dist_to_conc(dist, r_min, r_max[, rule])	Converts a swath of a size distribution function to an actual number concentration.

pyrcel.aerosol.dist_to_conc

pyrcel.aerosol.**dist_to_conc**(*dist*, *r_min*, *r_max*, *rule='trapezoid'*)
Converts a swath of a size distribution function to an actual number concentration.

Aerosol size distributions are typically reported by normalizing the number density by the size of the aerosol. However, it's sometimes more convenient to simply have a histogram of representing several aerosol size ranges (bins) and the actual number concentration one should expect in those bins. To accomplish this, one only needs to integrate the size distribution function over the range spanned by the bin.

Parameters

dist [object implementing a `pdf()` method] the representation of the size distribution

r_min, r_max [float] the lower and upper bounds of the size bin, in the native units of `dist`

rule [{'trapezoid', 'simpson', 'other'} (default='trapezoid')] rule used to integrate the size distribution

Returns

float The number concentration of aerosol particles the given bin.

Examples

```
>>> dist = Lognorm(mu=0.015, sigma=1.6, N=850.0)
>>> r_min, r_max = 0.00326456461236 0.00335634401598
>>> dist_to_conc(dist, r_min, r_max)
0.114256210943
```

1.6.5 Distributions

Collection of classes for representing aerosol size distributions.

Most commonly, one would use the *Lognorm* distribution. However, for the sake of completeness, other canonical distributions will be included here, with the notion that this package could be extended to describe droplet size distributions or other collections of objects.

BaseDistribution	Interface for distributions, to ensure that they contain a pdf method.
Gamma	Gamma size distribution
Lognorm(mu, sigma[, N, base])	Lognormal size distribution.
MultiModeLognorm(mus, sigmas, Ns[, base])	Multimode lognormal distribution class.

pyrcel.distributions.BaseDistribution

class pyrcel.distributions.**BaseDistribution**
Interface for distributions, to ensure that they contain a pdf method.

__init__(*self*, */*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

abstract cdf(*self*, *x*)
Cumulative density function

abstract pdf(*self*, *x*)
Probability density function.

abstract property stats

pyrcel.distributions.Gamma

class pyrcel.distributions.**Gamma**
Gamma size distribution

__init__(*self*, */*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

pyrcel.distributions.Lognorm

class pyrcel.distributions.**Lognorm**(*mu*, *sigma*, *N=1.0*, *base=2.718281828459045*)
Lognormal size distribution.

An instance of *Lognorm* contains a construction of a lognormal distribution and the utilities necessary for computing statistical functions associated with that distribution. The parameters of the constructor are invariant with respect to what length and concentration unit you choose; that is, if you use meters for mu and cm**-3 for N, then you should keep these in mind when evaluating the *pdf()* and *cdf()* functions and when interpreting moments.

Parameters

mu [float] Median/geometric mean radius, length unit.

sigma [float] Geometric standard deviation, unitless.

N [float, optional (default=1.0)] Total number concentration, concentration unit.

base [float, optional (default=np.e)] Base of logarithm in lognormal distribution.

Attributes

median, mean [float] Pre-computed statistical quantities

Methods

pdf(x)	Evaluate distribution at a particular value
cdf(x)	Evaluate cumulative distribution at a particular value.
moment(k)	Compute the *k*-th moment of the lognormal distribution.

__init__ (*self*, *mu*, *sigma*, *N=1.0*, *base=2.718281828459045*)
Initialize self. See help(type(self)) for accurate signature.

cdf (*self*, *x*)
Cumulative density function

$$\text{CDF} = \frac{N}{2}\left(1.0 + \text{erf}(\frac{\log x/\mu}{\sqrt{2}\log\sigma})\right)$$

Parameters

x [float] Ordinate value to evaluate CDF at

Returns

value of CDF at ordinate

invcdf (*self*, *y*)
Inverse of cumulative density function.

Parameters

y [float] CDF value, between (0, 1)

Returns

value of ordinate corresponding to given CDF evaluation

moment (*self*, *k*)
Compute the k-th moment of the lognormal distribution

$$F(k) = N\mu^k \exp\left(\frac{k^2}{2}\ln^2\sigma\right)$$

Parameters

k [int] Moment to evaluate

Returns

moment of distribution

pdf (*self*, *x*)
Probability density function

$$\text{PDF} = \frac{N}{\sqrt{2\pi}\log\sigma x}\exp\left(-\frac{\log x/\mu^2}{2\log^2\sigma}\right)$$

Parameters

x [float] Ordinate value to evaluate CDF at

Returns

value of CDF at ordinate

stats (*self*)
Compute useful statistics for a lognormal distribution

Returns

dict Dictionary containing the stats `mean_radius`, `total_diameter`, `total_surface_area`, `total_volume`, `mean_surface_area`, `mean_volume`, and `effective_radius`

pyrcel.distributions.MultiModeLognorm

class pyrcel.distributions.**MultiModeLognorm**(*mus*, *sigmas*, *Ns*, *base=2.718281828459045*)
Multimode lognormal distribution class.

Container for multiple Lognorm classes representing a full aerosol size distribution.

__init__ (*self*, *mus*, *sigmas*, *Ns*, *base=2.718281828459045*)
Initialize self. See help(type(self)) for accurate signature.

cdf (*self*, *x*)
Cumulative density function

pdf (*self*, *x*)
Probability density function.

stats (*self*)
Compute useful statistics for a multi-mode lognormal distribution

TODO: Implement multi-mode lognorm stats

The following dictionaries containing (multi) Lognormal aerosol size distributions have also been saved for convenience:

1. `FN2005_single_modes`: Fountoukis, C., and A. Nenes (2005), Continued development of a cloud droplet formation parameterization for global climate models, J. Geophys. Res., 110, D11212, doi:10.1029/2004JD005591
2. `NS2003_single_modes`: Nenes, A., and J. H. Seinfeld (2003), Parameterization of cloud droplet formation in global climate models, J. Geophys. Res., 108, 4415, doi:10.1029/2002JD002911, D14.
3. `whitby_distributions`: Whitby, K. T. (1978), The physical characteristics of sulfur aerosols, Atmos. Environ., 12(1-3), 135–159, doi:10.1016/0004-6981(78)90196-8.
4. `jaenicke_distributions`: Jaenicke, R. (1993), Tropospheric Aerosols, in *Aerosol-Cloud-Climate Interactions*, P. V. Hobbs, ed., Academic Press, San Diego, CA, pp. 1-31.

1.6.6 Activation

Collection of activation parameterizations.

lognormal_activation(smax, mu, sigma, N, kappa)	Compute the activated number/fraction from a lognormal mode
binned_activation(Smax, T, rs, aerosol[, approx])	Compute the activation statistics of a given aerosol, its transient size distribution, and updraft characteristics.
multi_mode_activation(Smax, T, aerosols, rss)	Compute the activation statistics of a multi-mode, binned_activation aerosol population.
arg2000(V, T, P[, aerosols, accom, mus, ...])	Computes droplet activation using a psuedo-analytical scheme.
mbn2014(V, T, P[, aerosols, accom, mus, ...])	Computes droplet activation using an iterative scheme.
shipwayabel2010(V, T, P, aerosol)	Activation scheme following Shipway and Abel, 2010 (doi:10.1016/j.atmosres.2009.10.005).
ming2006(V, T, P, aerosol)	Ming activation scheme.

pyrcel.activation.lognormal_activation

pyrcel.activation.**lognormal_activation**(*smax*, *mu*, *sigma*, *N*, *kappa*, *sgi=None*, *T=None*, *approx=True*)

Compute the activated number/fraction from a lognormal mode

Parameters

smax [float] Maximum parcel supersaturation

mu, sigma, N [floats] Lognormal mode parameters; `mu` should be in meters

kappa [float] Hygroscopicity of material in aerosol mode

sgi :float, optional Modal critical supersaturation; if not provided, this method will go ahead and compute them, but a temperature `T` must also be passed

T [float, optional] Parcel temperature; only necessary if no `sgi` was passed

approx [boolean, optional (default=False)] If computing modal critical supersaturations, use the approximated Kohler theory

Returns

N_act, act_frac [floats] Activated number concentration and fraction for the given mode

pyrcel.activation.binned_activation

pyrcel.activation.**binned_activation**(*Smax*, *T*, *rs*, *aerosol*, *approx=False*)

Compute the activation statistics of a given aerosol, its transient size distribution, and updraft characteristics. Following Nenes et al, 2001 also compute the kinetic limitation statistics for the aerosol.

Parameters

Smax [float] Environmental maximum supersaturation.

T [float] Environmental temperature.

rs [array of floats] Wet radii of aerosol/droplet population.

aerosol [`AerosolSpecies`] The characterization of the dry aerosol.

approx [boolean] Approximate Kohler theory rather than include detailed calculation (default False)

Returns

eq, kn: floats Activated fractions

alpha [float] N_kn / N_eq

phi [float] N_unact / N_kn

pyrcel.activation.multi_mode_activation

pyrcel.activation.**multi_mode_activation**(*Smax*, *T*, *aerosols*, *rss*)

Compute the activation statistics of a multi-mode, binned_activation aerosol population.

Parameters

Smax [float] Environmental maximum supersaturation.

T [float] Environmental temperature.

aerosol [array of `AerosolSpecies`] The characterizations of the dry aerosols.

rss [array of arrays of floats] Wet radii corresponding to each aerosol/droplet population.

Returns

eqs, kns [lists of floats] The activated fractions of each aerosol population.

pyrcel.activation.arg2000

pyrcel.activation.**arg2000**(*V*, *T*, *P*, *aerosols=[]*, *accom=1.0*, *mus=[]*, *sigmas=[]*, *Ns=[]*, *kappas=[]*, *min_smax=False*)

Computes droplet activation using a psuedo-analytical scheme.

This method implements the psuedo-analytical scheme of [ARG2000] to calculate droplet activation an an adiabatically ascending parcel. It includes the extension to multiple lognormal modes, and the correction for non-unity condensation coefficient [GHAN2011].

To deal with multiple aerosol modes, the scheme includes an expression trained on the mode std deviations, σ_i

ight]}

This effectively combines the supersaturation maximum for each mode into a single value representing competition between modes. An alternative approach, which assumes the mode which produces the smallest predict Smax sets a first-order control on the activation, is also available

Parameters

V, T, P [floats]

Updraft speed (m/s), parcel temperature (K) and pressure (Pa)

aerosols [list of `AerosolSpecies`] List of the aerosol population in the parcel; can be omitted if `mus`, `sigmas`, `Ns`, and `kappas` are present. If both supplied, will use `aerosols`.

accom [float, optional (default=:const:*constants.ac*)] Condensation/uptake accomodation coefficient

mus, sigmas, Ns, kappas [lists of floats] Lists of aerosol population parameters; must be present if `aerosols` is not passed, but `aerosols` overrides if both are present.

min_smax [boolean, optional] If *True*, will use alternative formulation for parameterizing competition described above.

Returns

smax, N_acts, act_fracs [lists of floats]

Maximum parcel supersaturation and the number concentration/activated fractions for each mode

pyrcel.activation.mbn2014

pyrcel.activation.**mbn2014**(*V*, *T*, *P*, *aerosols=[]*, *accom=1.0*, *mus=[]*, *sigmas=[]*, *Ns=[]*, *kappas=[]*, *xmin=1e-05*, *xmax=0.1*, *tol=1e-06*, *max_iters=100*)

Computes droplet activation using an iterative scheme.

This method implements the iterative activation scheme under development by the Nenes' group at Georgia Tech. It encompasses modifications made over a sequence of several papers in the literature, culminating in [MBN2014]. The implementation here overrides some of the default physical constants and thermodynamic calculations to ensure consistency with a reference implementation.

Parameters

V, T, P [floats] Updraft speed (m/s), parcel temperature (K) and pressure (Pa)

aerosols [list of `AerosolSpecies`] List of the aerosol population in the parcel; can be omitted if `mus`, `sigmas`, `Ns`, and `kappas` are present. If both supplied, will use `aerosols`.

accom [float, optional (default=:const:*constants.ac*)] Condensation/uptake accomodation coefficient

mus, sigmas, Ns, kappas [lists of floats] Lists of aerosol population parameters; must be present if `aerosols` is not passed, but `aerosols` overrides if both are present

xmin, xmax [floats, opional] Minimum and maximum supersaturation for bisection

tol [float, optional] Convergence tolerance threshold for supersaturation, in decimal units

max_iters [int, optional] Maximum number of bisections before exiting convergence

Returns

smax, N_acts, act_fracs [lists of floats] Maximum parcel supersaturation and the number concentration/activated fractions for each mode

pyrcel.activation.shipwayabel2010

pyrcel.activation.**shipwayabel2010**(*V*, *T*, *P*, *aerosol*)

Activation scheme following Shipway and Abel, 2010 (doi:10.1016/j.atmosres.2009.10.005).

pyrcel.activation.ming2006

pyrcel.activation.**ming2006**(*V*, *T*, *P*, *aerosol*)

Ming activation scheme.

NOTE - right now, the variable names correspond to the FORTRAN implementation of the routine. Will change in the future.

1.6.7 Constants

Commonly used constants in microphysics and aerosol thermodynamics equations as well as important model parameters.

Symbol	Variable	Value	Units	Description
g	`g`	9.8	m s**-2	gravitational constant
C_p	`Cp`	1004.0	J/kg	specific heat of dry air at constant pressure
ρ_w	`rho_w`	1000.0	kg m**-3	density of water at STP
R_d	`Rd`	287.0	J/kg/K	gas constant for dry air
R_v	`Rv`	461.5	J/kg/K	gas constant for water vapor
R	`R`	8.314	J/mol/K	universal gas constant
M_w	`Mw`	0.018	kg/mol	molecular weight of water
M_a	`Ma`	0.0289	kg/mol	molecular weight of dry air
D_v	`Dv`	3e-5	m**2/s	diffusivity of water vapor in air
L_v	`L`	2.25e6	J/kg/K	latent heat of vaporization of water
α_c	`ac`	1.0	unitless	condensation coefficient
K_a	`Ka`	0.02	J/m/s/K	thermal conductivity of air
a_T	`at`	0.96	unitless	thermal accommodation coefficient
ϵ	`epsilon`	0.622	unitless	ratio of M_w/M_a

Additionally, a reference table containing the 1976 US Standard Atmosphere is implemented in the constant `std_atm`, which is a pandas DataFrame with the fields

- `alt`, altitude in km
- `sigma`, ratio of density to sea-level density
- `delta`, ratio of pressure to sea-level pressure
- `theta`, ratio of temperature to sea-level temperature
- `temp`, temperature in K
- `press`, pressure in Pa
- `dens`, air density in kg/m**3
- `k.visc`, air kinematic viscosity
- `ratio`, ratio of speed of sound to kinematic viscosity in m**-1

Using default pandas functons, you can interpolate to any reference pressure or height level.

Current version: 1.3.1.dev-2da4611

Documentation last compiled: Aug 25, 2019

BIBLIOGRAPHY

[Nenes2001] Nenes, A., Ghan, S., Abdul-Razzak, H., Chuang, P. Y. & Seinfeld, J. H. Kinetic limitations on cloud droplet formation and impact on cloud albedo. Tellus 53, 133–149 (2001).

[SP2006] Seinfeld, J. H. & Pandis, S. N. Atmospheric Chemistry and Physics: From Air Pollution to Climate Change. Atmos. Chem. Phys. 2nd, 1203 (Wiley, 2006).

[Rothenberg2016] Daniel Rothenberg and Chien Wang, 2016: Metamodeling of Droplet Activation for Global Climate Models. *J. Atmos. Sci.*, **73**, 1255–1272. doi: http://dx.doi.org/10.1175/JAS-D-15-0223.1

[PK2007] Petters, M. D. & Kreidenweis, S. M. A single parameter representation of hygroscopic growth and cloud condensation nucleus activity. Atmos. Chem. Phys. 7, 1961–1971 (2007).

[Ghan2011] Ghan, S. J. et al. Droplet nucleation: Physically-based parameterizations and comparative evaluation. J. Adv. Model. Earth Syst. 3, M10001 (2011).

[SP2006] Seinfeld, John H, and Spyros N Pandis. Atmospheric Chemistry and Physics: From Air Pollution to Climate Change. Vol. 2nd. Wiley, 2006.

[SP2006] Seinfeld, John H, and Spyros N Pandis. Atmospheric Chemistry and Physics: From Air Pollution to Climate Change. Vol. 2nd. Wiley, 2006.

[Petty2006] Petty, Grant Williams. A First Course in Atmospheric Radiation. Sundog Publishing, 2006. Print.

[Bolton1980] Bolton, David. "The Computation of Equivalent Potential Temperature". Monthly Weather Review 108.8 (1980): 1046-1053

[RY1989] Rogers, R. R., and M. K. Yau. A Short Course in Cloud Physics. Burlington, MA: Butterworth Heinemann, 1989.

[PK2007] Petters, M. D., and S. M. Kreidenweis. "A Single Parameter Representation of Hygroscopic Growth and Cloud Condensation Nucleus Activity." Atmospheric Chemistry and Physics 7.8 (2007): 1961-1971

[ARG2000] Abdul-Razzak, H., and S. J. Ghan (2000), A parameterization of aerosol activation: 2. Multiple aerosol types, J. Geophys. Res., 105(D5), 6837-6844, doi:10.1029/1999JD901161.

[GHAN2011] Ghan, S. J. et al (2011) Droplet Nucleation: Physically-based Parameterization and Comparative Evaluation, J. Adv. Model. Earth Syst., 3, doi:10.1029/2011MS000074

[MBN2014] Morales Betancourt, R. and Nenes, A.: Droplet activation parameterization: the population splitting concept revisited, Geosci. Model Dev. Discuss., 7, 2903-2932, doi:10.5194/gmdd-7-2903-2014, 2014.

PYTHON MODULE INDEX

p

pyrcel.activation, 33
pyrcel.aerosol, 28
pyrcel.constants, 36
pyrcel.distributions, 30
pyrcel.driver, 20
pyrcel.thermo, 22

INDEX

Symbols

__init__() (*pyrcel.ParcelModel method*), 20
__init__() (*pyrcel.aerosol.AerosolSpecies method*), 29
__init__() (*pyrcel.distributions.BaseDistribution method*), 30
__init__() (*pyrcel.distributions.Gamma method*), 30
__init__() (*pyrcel.distributions.Lognorm method*), 31
__init__() (*pyrcel.distributions.MultiModeLognorm method*), 32

A

AerosolSpecies (*class in pyrcel.aerosol*), 28
arg2000() (*in module pyrcel.activation*), 34

B

BaseDistribution (*class in pyrcel.distributions*), 30
binned_activation() (*in module pyrcel.activation*), 33

C

cdf() (*pyrcel.distributions.BaseDistribution method*), 30
cdf() (*pyrcel.distributions.Lognorm method*), 31
cdf() (*pyrcel.distributions.MultiModeLognorm method*), 32
critical_curve() (*in module pyrcel.thermo*), 27

D

dist_to_conc() (*in module pyrcel.aerosol*), 29
dv() (*in module pyrcel.thermo*), 23

E

es() (*in module pyrcel.thermo*), 25

G

Gamma (*class in pyrcel.distributions*), 30

I

invcdf() (*pyrcel.distributions.Lognorm method*), 31
iterate_runs() (*in module pyrcel.driver*), 21

K

ka() (*in module pyrcel.thermo*), 23
kohler_crit() (*in module pyrcel.thermo*), 27

L

Lognorm (*class in pyrcel.distributions*), 30
lognormal_activation() (*in module pyrcel.activation*), 33

M

mbn2014() (*in module pyrcel.activation*), 35
mmg2006() (*in module pyrcel.activation*), 35
moment() (*pyrcel.distributions.Lognorm method*), 31
multi_mode_activation() (*in module pyrcel.activation*), 34
MultiModeLognorm (*class in pyrcel.distributions*), 32

P

parcel_ode_sys() (*pyrcel.parcel method*), 18
ParcelModel (*class in pyrcel*), 15, 19
pdf() (*pyrcel.distributions.BaseDistribution method*), 30
pdf() (*pyrcel.distributions.Lognorm method*), 32
pdf() (*pyrcel.distributions.MultiModeLognorm method*), 32
pyrcel.activation (*module*), 33
pyrcel.aerosol (*module*), 28
pyrcel.constants (*module*), 36
pyrcel.distributions (*module*), 30
pyrcel.driver (*module*), 20
pyrcel.thermo (*module*), 22

R

rho_air() (*in module pyrcel.thermo*), 24
run() (*pyrcel.ParcelModel method*), 16
run_model() (*in module pyrcel.driver*), 21

S

Seq() (*in module pyrcel.thermo*), 26
Seq_approx() (*in module pyrcel.thermo*), 26
set_initial_conditions() (*pyrcel.ParcelModel method*), 17
shipwayabel2010() (*in module pyrcel.activation*), 35
sigma_w() (*in module pyrcel.thermo*), 25
stats() (*pyrcel.aerosol.AerosolSpecies method*), 29
stats() (*pyrcel.distributions.BaseDistribution property*), 30
stats() (*pyrcel.distributions.Lognorm method*), 32
stats() (*pyrcel.distributions.MultiModeLognorm method*), 32