https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


# Exploring Cloud Microphysics Modeling Concepts the Pythonic Way

Sylwester Arabas[1], Agnieszka Żaba[1], Daria Klimaszewska[1], Emma C. Ware[2], Clare E. Singer[3], Manhal Alhilali[4], Jason Barr[5], Daniel G. Partridge[6], Shin-ichiro Shima[4], and Robert Wood[5]

[1]Faculty of Physics and Applied Computer Science, AGH University of Krakow, Krakow, Poland
[2]Department of Land, Air, and Water Resources, University of California Davis, Davis, CA, USA
[3]University of Colorado, Boulder, CO, USA
[4]Graduate School of Information Science, University of Hyogo, Kobe, Japan
[5]Department of Atmospheric and Climate Science, University of Washington, Seattle, WA, USA
[6]Department of Mathematics and Statistics, University of Exeter, Exeter, UK

**Correspondence:** Sylwester Arabas (sylwester.arabas@agh.edu.pl) and Agnieszka Żaba (azaba@agh.edu.pl)

**Abstract.** We share with the community a concise Pythonic codebase for hands-on exploration of cloud microphysics modeling concepts. It includes an adiabatic air-parcel model simulating diffusional droplet growth – leading to condensation, activation, evaporation, and ripening; as well as a box model resolving collisional growth of particles using the Super-Droplet Method Monte-Carlo scheme. Leveraging Pythonic abstractions and programmatic unit handling, the code combines pseudo-code-level readability, auditability, and dimensional-correctness enforcement with high performance through just-in-time (JIT) compilation, which generates native machine code on the fly, avoiding bytecode interpretation overhead for compute-heavy routines. With the goal of illustrating how concise yet complete implementations support both teaching and research software engineering, the paper includes and narrates the entire code, plotting logic included. The code runs with a single click "in the cloud" – on Google Colab or other Jupyter hubs, and thus constitutes a suitable resource for self-study or for a short course in microphysics modeling. Presented technical solutions originate from the PySDM particle-based simulation package, but are applicable in a wider scope and independently of PySDM as exemplified here. The paper includes section-wise exploratory exercises designed to guide further application of the methods and concepts discussed.

## Introduction

### Python in Atmospheric Sciences

In Lin (2012), there was remarkably accurate foresight predicting the widespread adoption of Python in Earth sciences. Yet, as noted in the paper, embracement of Python a dozen years ago still posed challenges, including

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


the speed penalty, the immaturity of the scientific package ecosystem, and the limited availability of documentation, resources, and examples helping users (and nowadays, machines) to learn from. The utility of PYTHON for introducing students to scientific computing concepts might seem self-evident today, and its educational potential in atmospheric science, and in earth sciences in general, has been discussed in, e.g., Arms et al. (2020) and Trauth (2024), respectively.

In this paper, we present a practical example of how PYTHON, along with a set of mature packages, can be used to engineer a compact modeling framework – in what amounts to a short novella of executable pseudocode. The resulting implementation is readily usable in the cloud (e.g., using GOOGLE COLAB platform or using institutional Jupyter hubs), which offers a clear advantage in education. The crux of the present work is to exhibit that this is achievable without compromising performance or trading off maintainability-oriented abstractions rooted in research software engineering.

### Python as a glue language

The feasibility of this stems from the ability of PYTHON to serve as a "glue language" – a term coined in van Rossum (1998) pointing to PYTHON's role in integrating multiple technologies into a single application. In fact, this paradigm is not merely an added value, it is rather a necessity. In contrast to technologies such as JULIA (or R, IDL, MATLAB), PYTHON is designed with minimalism in mind – the language depends on external packages even to provide support for arrays (while NUMPY is the de-facto standard, multiple packages provide alternatives mimicking the same API). Analogously, just-in-time compilation is delegated to external packages. This is a double-edged sword: it can hinder cohesion across projects, while at the same time offering great flexibility. Users can pick from a range of packages to use for array handling (Meurer et al., 2023) or *just-in-time* (JIT) compilation, while developers can offer the PYTHON community an entirely new compilation method or new hardware support without interfering with language internals.

### Scope of this work: cloud microphysics

In this paper, we demonstrate how one can glue together NUMPY (Harris et al., 2020), NUMBA (Lam et al., 2015), PINT (Grecco, 2025) and SCIPY (Virtanen et al., 2020) with its interface to the LSODA (Hindmarsh, 1983; Petzold, 1983) ordinary differential equation (ODE) integrators, combined into a modeling framework that combines research-grade performance with code clarity and readability for learners.

The result, comprising of ca. 400 lines of expressive code, constitutes an implementation of a particle dynamics model that explicitly represents growth by both vapor diffusion and particle collisions, with particle-resolved (*moving-sectional*) size-spectrum representation. Particle-based approach to microphysics modeling is gaining significant adoption in the cloud modeling community (see Grabowski et al., 2019; Morrison et al., 2026, for recent perspectives). One of its virtues is the by-design high fidelity in capturing aerosol-cloud interactions, which makes particle-based

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


methods particularly suitable for studying aerosol indirect effects on climate, in contexts ranging from ordinary natural and anthropogenic emissions to intentional geoengineering.

Here, for explanatory purposes, we present the two key microphysical processes of diffusional and collisional particle growth in separate sections, though fully-fledged atmospheric models call both at every timestep. First, the diffusional dynamics are exemplified with an adiabatic-parcel setup capturing activation on aerosol and subsequent droplet growth and ripening. Second, collisional dynamics are exemplified with a box-model setup comparing the provided multi-threaded implementation of the SDM Monte-Carlo algorithm (Shima et al., 2009) with an analytic solution to the Smoluchowski coagulation equation. Both examples share utility code and a common particle-sampling logic at initialization contributing to the code minimalism.

Presented model developments complement the range of PYTHON implementations of algorithms applicable in aerosol research presented in Topping and Bane (2022). Throughout the paper, we emphasize object oriented and expressive *Pythonic* coding as advocated in Ham (2021). Unicode and short variable names are used to match mathematical notation. Similarly, as in Irving et al. (2021, see sections 0.2-0.4), the herein presentation: targets readers acquainted with PYTHON basics, provides an actual implementation addressing a real research question, and offers concise code suitable for use in participatory live-coding coursework.

## Active learning with Python

Along with other technologies such as JUPYTER notebooks and languages such as JULIA, PYTHON is on the forefront of an ongoing shift towards engaging students in active-learning curricula that bridge the gap between lectures and coding (Granger and Pérez, 2021; Petters, 2022). With the canonical modeling examples chosen herein, we construct the paper aiming at providing a ready-to-use teaching material, yet the discussed and employed design patterns, as well as the code itself are readily applicable in research.

## Structure and format of the paper

Leveraging the expressive Pythonic coding and widespread visualization tooling, aiming to highlight the level of code readability for learners, we embed the complete model implementation in the narrative below, including all code used to generate figures. Throughout the text, software names are typeset in SMALLCAPS (e.g., NUMPY), code identifiers are typeset with `monospace` font, while both PYTHON jargon and selected cloud-modeling terms are typeset in *italics* (e.g., `@JIT` is a *decorator*, *multiplicity* is an integer).

The remainder of this paper is structured into ten model-development sections (or instructional modules), each building on the preceding ones, and each including exercises provided as starting points for independent investigation:

- Section 1: aerosol size spectrum sampling,

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


- Section 2: unit-aware coding and plotting,
- Section 3: parcel model governing equations,
- Section 4: JIT compilation using Numba,
- Section 5: parcel ODE system definition,
- Section 6: test-time dimensional analysis using Pint,
- Section 7: initial condition at equilibrium,
- Section 8: ODE integration and result analysis,
- Section 9: Super-Droplet Method algorithm,
- Section 10: SDM against analytic solution.

The paper closes with a summary and instructions on how to execute the presented code in the cloud.

## 1 (Super-)Particle population sampling

The system modeled here – a cloud – is a colloidal suspension of liquid particles in moist air. Modeling the dynamics of its microstructure therefore requires the representation of the thermodynamic state variables and the size spectrum of particles. For the latter, we embrace a coarse-grained particle model, in which each computational *super particle* (or *super droplet*) represents a multiplicity of physical particles of identical properties, referred to as *particle attributes*. The key attributes for the present study are particle size or mass, and its integer multiplicity, while other attributes can describe particle composition and location in space. Such particle-based representation provides a mesoscopic middle ground between a bulk model and a direct numerical simulation of all individual particles. Mathematically, the representation is equivalent to the *moving-sectional* or *full-moving structure* with point-width sections (size classes, particles), where the movement refers to Lagrangian tracking in attribute space (see, e.g., Jacobson, 2012, section 14.5.2). Employment of the *point-particle/moving-sectional* spectrum representation has been commonly used in cloud microphysics modeling (see, e.g., Howell, 1949, and Rothenberg and Wang (2017), for a seminal classic and a recent Python development).

To initialize a super-particle state with *dry sizes* of aerosol particles (a subset of which will serve as cloud condensation nuclei, CCN), here we will sample from a lognormal probability density commonly used for representing size distributions originating from multiplicative processes shaping particulate pollutants in the atmosphere. To this end, we create a *dictionary* (key-value mapping) named `cfg_ccn` holding configuration of a CCN activation simulation with the `dist` key pointing to a representation of a lognormal distribution from `SciPy.stats`:

```
import scipy.stats as sp_st
import math
```

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


```
cfg_ccn = {
    'dist': sp_st.lognorm( # dry CCN spectrum
        s := math.log(σ_geom := 1.75),
        scale = (r_mode_m := 20e-9) * math.exp(s**2),
    ),
    'norm': (n_per_cc_stp := 8000) * (v_cc_stp := 100),
    'Δv_m3_stp': (v_cc_stp * (1/100)**3),
    'n_sd': 44,
}
```

(where the *walrus operator* `:=` *binds* a value to a name and evaluates to that value – similar to C's assignment operator, but limited to *binding identifiers* within *expressions*). We prescribe the dry-CCN spectrum as a lognormal distribution with geometric standard deviation 1.75 and modal radius 20 nm. The `dist` key stores the shape of this spectrum as a unit-normalized probability density function. To scale that shape to an actual aerosol population, we specify the `norm` key as mapping to a particle concentration normalization factor corresponding to 8000 particles per cubic centimeter populating a sample volume $\Delta v$ of 100 cubic centimeters of air at standard temperature and pressure. In contrast to the physical parameters, the $n_{\mathrm{sd}} = 44$ key defines the size-spectral resolution in terms of the number of *super particles* (i.e., discrete size classes) to be sampled in a simulation.

A common choice in *super-particle* modeling is to employ quantile sampling yielding a population of *super particles* in which each represents an equal multiplicity of physical particles (see Ware et al., 2026, for discussion, and appendix A therein for visualizations of different sampling methods). The code listing below defines a function that uses SciPy's `ppf()` (percentile point) method to perform deterministic quantile sampling returning an x-y *tuple* of two NumPy vectors representing the sampled probability distribution (vertical axis (y): equal multiplicities as integers, horizontal axis (x): quantiles as floating-point values; both using 64-bit NumPy data types):

```
import numpy as np

def quantile_sample(*, dist, n_sd, norm):
    return dist.ppf(q=(np.arange(n_sd) + 0.5) / n_sd), \
           np.full(n_sd, norm / n_sd, dtype=np.int64)
```

Let us highlight that evenly weighted quantile sampling is just one of possible approaches to initialize the attributes and multiplicities, with uniform-in-attribute (e.g., size) or uniform-in-log-attribute schemes being common alternative choices (see, e.g., Ware et al., 2026, for discussion), and with random sampling commonly used in Monte-Carlo

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


simulations (see exercise below). Furthermore, both quantile and uniform-in-attribute schemes can be applied not only to number distributions, but also – for instance – to mass distributions, which is particularly relevant for resolving the large-size tails of distributions, which correspond to small numbers of particles, yet those with momentous importance for CCN activation (as in the case of so-called Giant CCN, see e.g., Dziekan et al., 2021). Finally, the sampling strategies are not limited to the quantile or uniform apices – in Matsushima et al. (2023) a method for constructing a continuum of initialization methods using optimal-transport formalism is presented and applied to super-droplet cloud simulations.

The `quantile_sample()` function can be used to supplement the simulation configuration with sampled *dry radii* $r_\mathrm{d}$ and *multiplicities* $\xi$:

```
cfg_ccn["r_d"], cfg_ccn["ξ"] = quantile_sample(
    dist=cfg_ccn['dist'],
    n_sd=cfg_ccn['n_sd'],
    norm=cfg_ccn['norm'],
)
```

The result of the sampling can be plotted (Figure 1) using the MATPLOTLIB plotting package (Hunter, 2007). The `matplotlib.pyplot.eventplot()` function plots the sampled values, while the `matplotlib.pyplot.hist()` function both computes and plots a histogram from these values (treated as weighted samples – with multiplicities $\xi$ serving as weights). Note: while the above quantile sampling yields uniform multiplicities, it is not assumed anywhere in the following code.

```
from matplotlib import pyplot

pyplot.eventplot(
    cfg_ccn['r_d'] / (µm := 1e-6),
    color='orange',
    label='size attribute values'
)
_, bin_edges, _ = pyplot.hist(
    cfg_ccn["r_d"] / µm,
    weights=cfg_ccn["ξ"],
    density=True,
    bins=8, label='binned samples'
)
pyplot.plot(
```

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


```
    x := np.linspace(*bin_edges[[0, -1]]),
    cfg_ccn['dist'].pdf(x * µm) / (µm**-1),
    'r--', label='lognormal pdf'
)
pyplot.ylabel("number density [1/µm]")
pyplot.xlabel("dry radius [µm]")
pyplot.legend();
```

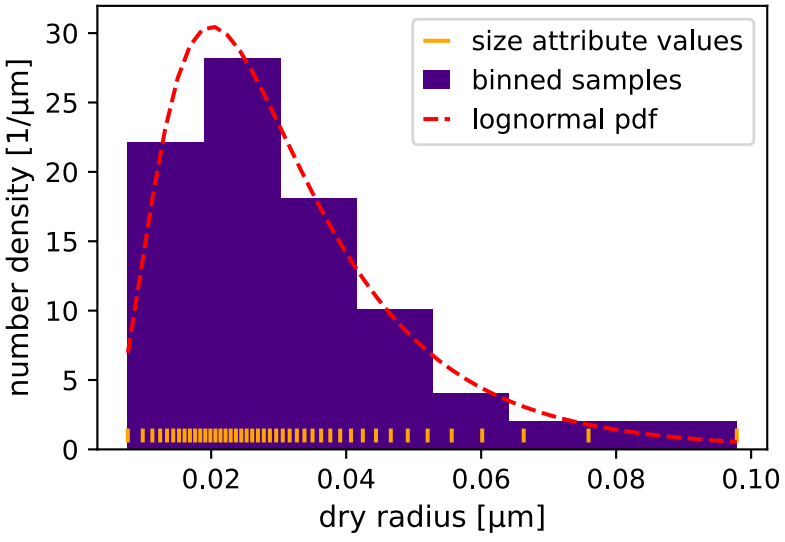


**Figure 1.** Histogram of super-particle size attribute values sampled from a lognormal probability density function (see Section 1).

**Exercises:**

- Replace the `(np.arange(n_sd) + 0.5) / n_sd` set of quantiles with randomly picked values, e.g., using `np.random.random(n_sd)` in the `quantile_sample()` function to employ randomized quantile sampling.
- In addition to the provided histogram of contribution of different sizes to the total number of particles, plot histogram depicting contribution to the total mass (see, e.g., chapter 8.1.2 in Seinfeld and Pandis, 2016).

## 2 Constants, Köhler curves & unit-aware coding

The above code snippets demonstrate that even in a few dozens of lines of basic modeling code, there are multiple opportunities for unit-related inconsistencies. Even though the least maintainable approach – relying on comments to track units – was not employed above, several potentials for error remain: (a) if not for the lucky coincidence

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


of cubic-centimeter units in calculation of the `norm`, an explicit conversion would need to be coded; (b) there is no mechanism in place alerting a developer of unphysical arithmetic operations (like adding Newtons to Joules); (c) the manually hardcoded units for axis labeling are in no way guaranteed to match the units conveyed by variable names.

The object-oriented abstractions of high-level programming languages (Python, Julia, C++ and modern FORTRAN included) allow to express the physical dimensionality of constants and variables in a programmatic way, eliminating all three of the above potential bug sources. One of the solutions available in the Python package ecosystem is Pint (Grecco, 2025; used, e.g., within the MetPy meteorological data analysis and visualization package May et al., 2022; and covered in the DeCaria and Petty, 2020, Python textbook with atmospheric-science focused examples (chapter 21.7 therein)). To simplify its use along with Numba and SciPy – which we are also going to leverage in Section 4 and Section 7, we propose to use a set of Pint-based utilities available in the PySDM cloud-modeling package (Bartman et al., 2022; de Jong et al., 2023). To start with, we will define a function returning a catalog of constants for use in the parcel model:

```
from collections import namedtuple
import scipy.constants as sp_c
import mendeleev as mv
from PySDM import physics

def constants():
    si = physics.si

    M_d = ( # molar mass or dry air
        0.78 * 2 * mv.N.atomic_weight * si.g / si.mol +
        0.21 * 2 * mv.O.atomic_weight * si.g / si.mol +
        0.01 * 1 * mv.Ar.atomic_weight * si.g / si.mol
    )
    M_v = ( # molar mass of water vapor
        1 * mv.O.atomic_weight * si.g / si.mol +
        2 * mv.H.atomic_weight * si.g / si.mol
    )
    T_0C = sp_c.zero_Celsius * si.K
    p_stp = 101.325 * si.kPa             # ICAO
    T_stp = 15 * si.K + T_0C             # ICAO
    R = sp_c.R * si.J / si.K / si.mol    # molar gas const

    co = namedtuple('Consts', (co := {
```

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


```
        'R_d': (R_d := R / M_d),
        'R_v': R / M_v,
        'T_0C': T_0C,
        'c_pd': 1 * si.kJ / si.kg / si.K, # heat capacity
        'l_v': 45 * si.kJ / si.mol / M_v, # latent heat
        'g': sp_c.g * si.m / si.s**2,     # grav. accel.
        'ρ_w': 1 * si.kg / si.liter,  # liq. water density
        'D_v': .25 * si.cm**2 / si.s, # vapor-in-air diff.
        'σ_w': 72 * si.mJ / si.m**2,  # surf tension coef.
        'ρ_stp': p_stp / R_d / T_stp, # ICAO
        'r_0': 1 * si.nm,             # for ln(r/r_0)
        'B80G0': 6.112 * si.hPa,      # Bolton 1980
        'B80G1': 17.67,               #   saturation vapor
        'B80G2': 243.5 * si.K,        #   pressure coeffs
    }))(**co)
    return co, si
```

where the returned value consists of a pair (two-element *tuple*) of: *namedtuple* of the constants (allowing access via `co.R_d`, `co.R_v`, `co.T_0C`, …), and an object `si` representing the SI unit registry employing the PINT package interface (for performance and compatibility reasons, usage of PINT itself is made optional here, as elaborated in Section 6). The technical reasons for using the *namedtuple* type relates to its inherent immutability which conveys that its contents are invariable. The MENDELEEV package is used to compute molar masses from elemental composition, rather than hardcoding these values, and making it clear that the values follow directly from the composition of dry air and water vapor. Also, the International Civil Aviation Organization (ICAO) standard atmosphere temperature and pressure, and several thermodynamic constants approximated to two significant digits are defined.

Such design pattern allows to address all three of the above-identified unit-related code flaws. First, handling of unit conversions using the `si` registry addresses the issue (a) by freeing developer from the need to explicitly do SI-prefix scaling. Second, usage of PINT renders the issue (b) of unphysical arithmetics impossible, because these will now *raise exceptions* if run within the PYSDM's *DimensionalAnalysis* context manager, as depicted below:

```
from PySDM.physics.dimensional_analysis import DimensionalAnalysis

with DimensionalAnalysis():
    co, si = constants()
    try:
        co.T_0C + co.g
```

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


```
    except Exception as ex:
        print(ex)
```

```
Cannot convert from 'kelvin' ([temperature]) to
'meter / second ** 2' ([length] / [time] ** 2)
```

(the attempt to sum temperature and gravitational acceleration is rejected and the *exception* includes error message pointing to the mismatched dimensionality of operands). Moreover, addressing (c), the physical dimensionality of abscissa and ordinate can now be programmatically determined by leveraging PINT's integration with MATPLOTLIB. We demonstrate it below with a quick-look plot of several Köhler curves describing the dependence on *dry and wet radii* of equilibrium vapor saturation ratio $\mathrm{RH}_{\mathrm{eq}}$ over an aqueous solution droplet. The $\kappa$-Köhler model used here parametrizes the effect of soluble material using a hygroscopicity parameter $\kappa$ (eq. 6 in Petters and Kreidenweis, 2007):

$$\mathrm{RH}_{\mathrm{eq}} = \frac{r_{\mathrm{w}}^3 - r_{\mathrm{d}}^3}{r_{\mathrm{w}}^3 - r_{\mathrm{d}}^3(1-\kappa)} \exp\left(\frac{2\sigma_{\mathrm{w}}}{R_{\mathrm{v}} T \rho_{\mathrm{w}} r_{\mathrm{w}}}\right) \tag{1}$$

where $r_{\mathrm{w}}$ is the so-called *wet radius*, i.e., the size of a water particle formed on dry soluble nuclei of size characterized by the *dry radius* $r_{\mathrm{d}}$. Eq. 1 can be implemented in PYTHON, e.g., this way:

```
eqs = {'RH_eq': lambda co, κ, r_w, r_d, T: (
    r_w**3 - r_d**3
) / (
    r_w**3 - r_d**3 * (1 - κ)
) * np.exp(2 * co.σ_w / co.R_v / T / co.ρ_w / r_w)}
```

where the anonymous function (i.e., a *lambda*) is put under the key `RH_eq` into a newly created *dictionary* `eqs`, which will be used to group all governing equations – enabling enumerating over the functions, and passing them collectively to other routines. The *positional argument* `co` is expected to point to a constants catalog, and this pattern will be used for subsequent equation implementations.

In the code generating Figure 2, it is depicted how PINT's integration with MATPLOTLIB makes the axes units automatically labeled based on the dimensionality of arguments passed to the plotting function (`semilogx` here):

```
cfg_ccn['κ'] = 0.7

with DimensionalAnalysis():
```

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


```
    co, si = constants()
    si.setup_matplotlib()

cmn = {'co': co, 'κ': cfg_ccn['κ'], 'T': 300 * si.K}
    step = cfg_ccn['n_sd'] // 4
    for r_d in cfg_ccn['r_d'][::step] * si.m:
        pyplot.semilogx(
            r_w := np.logspace(-1, 1) * si.um,
            eqs['RH_eq'](r_w=r_w, r_d=r_d, **cmn),
            label=fr"$r_\text{{d}}$={r_d.to(si.nm):.3g~}",
        )
    pyplot.axhline(1, label="RH=100%", color='k')
    pyplot.legend()
```

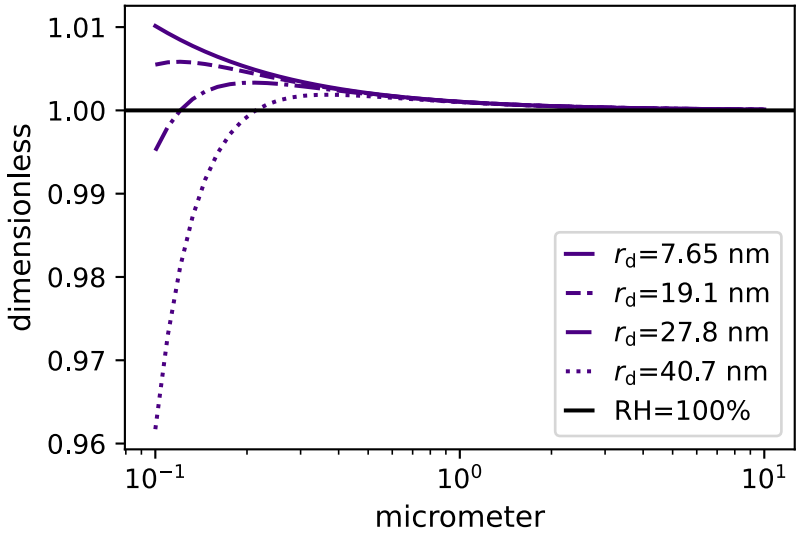


**Figure 2.** Köhler curves for four selected dry radii, with plot axes automatically labeled with units of the variables passed to the plotting function.

Note that the automatic labeling occurs only if the plotting code, and the instantiation of the constants catalog is enclosed within the `DimensionalAnalysis` *context*. Technically, this is achieved using the PYTHON's built-in *reimport* functionality which is used by `DimensionalAnalysis` to temporarily replace the default *float*-typed SI-prefix catalog `PySDM.physics.si` with unit-aware objects constructed using the PINT package.

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


**Exercises:**

- Extend the estimation of the molar mass of dry air to take into account the carbon dioxide content, explore how it has changed along the recent decades (see, e.g., Gatley et al., 2008).
- The Köhler curve defined by (1) features the Raoult and the Kelvin terms, corresponding to solute and curvature effects, respectively; explore how the equilibrium saturation depends on the droplet electric charge, how strongly a droplet needs to be charged for it to enhance activation (see, e.g., Harrison and Ambaum, 2008).

## 3 Adiabatic parcel model governing equations

The parcel model used herein follows the minimal formulation used in Arabas and Shima (2017, eq. 13 therein), extending its application from monodisperse to polydisperse particle population. The model takes a form of a system of $2+n_{\mathrm{sd}}$ ordinary differential equations describing the rate of change of: the dry-air partial pressure $p_{\mathrm{d}}$, the ambient temperature $T$, and $n_{\mathrm{sd}}$ size attributes $x^{[i]} = \ln \frac{r_{\mathrm{w}}^{[i]}}{r_0}$ (one per each super-particle $i$):

$$\frac{d}{dt}\begin{bmatrix} p_{\mathrm{d}} \\ T \\ \ln \frac{r_{\mathrm{w}}^{[1]}}{r_0} \\ \vdots \\ \ln \frac{r_{\mathrm{w}}^{[n_{\mathrm{sd}}]}}{r_0} \end{bmatrix} = \begin{bmatrix} -w\rho_{\mathrm{d}} g \\ \left(\dot{p}_{\mathrm{d}}/\rho_{\mathrm{d}} - \dot{q}_{\mathrm{v}} l_{\mathrm{v}}\right)/c_{\mathrm{pd}} \\ \underbrace{\frac{1}{r_{\mathrm{w}}^{[i]}} \frac{D_{\mathrm{v}}}{\rho_{\mathrm{w}}} \left(\rho_{\mathrm{v}} - \rho_{\circ}\right)}_{dr_{\mathrm{w}}^{[i]}/dt} \Big/ \frac{dr_{\mathrm{w}}^{[i]}}{dx} \end{bmatrix} \tag{2}$$

```
eqs['dp_d__dt'] = lambda co, w, ρ_d: -w * ρ_d * co.g

eqs['dT__dt']   = lambda co, dp_d__dt, dq_v__dt, ρ_d: \
  (dp_d__dt / ρ_d - dq_v__dt * co.l_v) / co.c_pd

eqs['dr_w__dt'] = lambda co, r_w, ρ_v, ρ_o: \
  co.D_v / co.ρ_w * (ρ_v - ρ_o) / r_w
```

where the equation for $\dot{p}_{\mathrm{d}}$ conveys hydrostatic pressure adjustments upon vertical motion with velocity $w$ of air with dry-air density $\rho_{\mathrm{d}}$ subject to gravitational acceleration $g$; the equation for $\dot{T}$ conveys the heat budget of the parcel with a pressure-derivative term embodying cooling due to adiabatic expansion, and a vapor mixing-ratio derivative $\dot{q}_{\mathrm{v}}$ term embodying latent heat release by condensing vapor (with $l_{\mathrm{v}}$ denoting the specific heat of vaporization, and $c_{\mathrm{pd}}$ denoting specific heat capacity of dry air). The equation for $\dot{r}_{\mathrm{w}}$ models the diffusion of vapor around a droplet with its mass flux being proportional to the vapor diffusivity in air $D_{\mathrm{v}}$ and to the difference of ambient ($\rho_{\mathrm{v}}$) and drop-surface ($\rho_{\circ}$) vapor densities.

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


Noting the positive definiteness of $r_\mathrm{w} > 0$, we formulate the system in terms of $x = \ln(r_\mathrm{w}/r_0)$. This transformation embeds the positivity constraint and rescales the dynamics near $r_\mathrm{w} \to 0$, thereby enhancing the robustness of the time integration, for which intrinsic numerical difficulties have long been documented (e.g., Árnason and Brown, 1971).

```
eqs['x']        = lambda co, r_w: np.log(r_w / co.r_0)
eqs['r_w']      = lambda co, x: co.r_0 * np.exp(x)
eqs['dr_w__dx'] = lambda r_w: r_w
```

The dry-air and vapor densities are proportional to the respective partial pressures $p_c$ of component $c$ with proportionality defined by the ideal gas law ($R_c$ denoting specific gas constant):

$$\rho_c = p_c/(R_c T) \tag{3}$$

```
eqs['ρ_d'] = lambda co, p_d, T: p_d / co.R_d / T
eqs['ρ_v'] = lambda co, p_v, T: p_v / co.R_v / T
```

The vapor density at the drop surface is assumed to follow from phase equilibrium, and is thus computed using saturation vapor pressure $p_\mathrm{vs}$ and the Köhler curve for given values of dry and wet radii:

$$\rho_\circ = \rho_\mathrm{v}\left(RH_\mathrm{eq}(\kappa, r_\mathrm{w}, r_\mathrm{d}, T)\cdot p_\mathrm{vs}(T), T\right) \tag{4}$$

with the equilibrium vapor pressure formulated as in Bolton (1980, eq. 10 therein):

```
eqs['p_vs'] = lambda co, T: co.B80G0 * np.exp(
    (co.B80G1 * (T - co.T_0C)) / ((T - co.T_0C) + co.B80G2)
)
```

also used to define the ambient relative humidity:

$$\mathrm{RH} = p_\mathrm{v}/p_\mathrm{vs}(T) = \rho_\mathrm{d} q_\mathrm{v}/\rho_\mathrm{vs} \tag{5}$$

```
eqs['RH'] = lambda q_v, ρ_vs, ρ_d: ρ_d * q_v / ρ_vs
```

which depends on the water-vapor mixing ratio $q_\mathrm{v} = q_\mathrm{t} - q_\mathrm{l}$ defined using the constant total-water mixing ratio $q_\mathrm{t}$, and the time-dependent liquid-water mixing ratio $q_\mathrm{l}$, the latter computed as the ratio of the dot product (@ operator) of multiplicities and volumes to the mass $m_\mathrm{d}$ of dry air:

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


```
eqs['q_l'] = lambda co, se, r_w: \
  4/3 * np.pi * (co.ρ_w * se.ξ @ r_w**3) / se.m_d
```

The rate of change of the vapor mixing ratio is:

```
eqs['dq_v__dt'] = lambda co, se, r_w, dr_w__dt: \
  -4 * np.pi * (se.ξ * r_w**2 @ dr_w__dt) / se.m_d * co.ρ_w
```

Finally, an approximate formula for the critical radius of a droplet is defined as the location of the maximum of the Köhler curve (leading-order terms approximation):

$$\mathrm{RH}_{\mathrm{eq}} \approx 1 + \overbrace{\frac{2\sigma_{\mathrm{w}}}{R_{\mathrm{v}} T \rho_{\mathrm{w}}}}^{A} \frac{1}{r_{\mathrm{w}}} - \frac{\kappa r_{\mathrm{d}}^3}{r_{\mathrm{w}}^3} \quad \rightsquigarrow \quad r_{\mathrm{c}} = \sqrt{\frac{3\kappa r_{\mathrm{d}}^3}{A}} \tag{6}$$

```
eqs['r_c'] = lambda co, κ, r_d, T: \
  (3/2 * κ * r_d**3 * co.R_v * T * co.ρ_w / co.σ_w)**.5
```

Values of $r_{\mathrm{c}}$ will be used in Section 7 to define the initial condition, and in Section 8 for diagnosing concentration of activated droplets.

**Exercise:**

- Recast the governing thermodynamic equations so that the model state features the saturation ratio (see, e.g., eq. (3.1) in Squires, 1952, note: supersaturation is defined there as the dew-point excess and has the dimension of temperature).

## 4 JIT compilation

Next, the functions implementing the governing equations of the model, defined above, are marked for compilation into optimized machine code using Numba JIT compiler (`jit` and `JIT` functions have *decorator* semantics – expecting a function as argument, returning a function):

```
from numba import jit
from functools import partial
JIT = partial(jit, error_model='numpy',
              fastmath=True, boundscheck=False)
```

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


where the *keyword arguments* control compilation options, including ignoring divide-by-zero errors (default is to *raise* an *exception*); opting for non-strict floating-point transformations by assuming no NaNs, infinities, or signed-zero distinctions; and omitting checks for out-of-bounds array access (which is default in NUMBA).

We then define a new *namedtuple* type `Eqs`, which groups all equations into a single immutable container with `.member` access semantics. Using the `JIT` compiler, we create two instances of `Eqs`: `eqp` with the original PYTHON functions, and `eqj` with their JIT-compiled versions:

```
Eqs = namedtuple("Eqs", eqs.keys())
eqp = Eqs(**eqs)
eqj = Eqs(**{k: JIT(v) for k, v in eqs.items()})
```

The two instances are created in order to enable using the same PYTHON code either with unit-aware constants catalog within the `DimensionalAnalysis()` *context manager* or ignoring the unit abstractions and executing compiled code. The goal is to cover all codebase with dimensional analysis checks for testing purposes, while executing the same code – without any runtime overhead from units checking – during simulations. Analogous solutions for compiling a single codebase with or without dimensional analysis is achievable in C++ using templates and BOOST.UNITS library (see Arabas et al., 2015, sec. 5.3).

Using the NUMBA reduces execution times by transitioning from interpreted dynamically-typed mode of operation to statically-typed JIT-compiled mode (and by providing multi-threading capabilities). The performance gains will be quantified in Section 10.

**Exercise:**

- Explore other Python just-in-time compilers, e.g. JAX, and the corresponding solutions for handling physical units, e.g., UNXT (Starkman et al., 2025).

## 5 Definition of the ODE system

To interface with SCIPY's initial-value-problem solvers, the right-hand-side of (2) should be defined as a *callable* with signature `dy_dt(t, y, ...)`, returning an array-like object of the same shape as the array-like state vector `y` (with the non-syntactic ellipsis standing here for an arbitrary number of additional parameters).

The following *namedtuple* defines indices for accessing elements of the state vector `y`:

```
ix = namedtuple("Indices", (ix := {
    'x':  slice(0, cfg_ccn['n_sd']),
```

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


```
    'p_d': cfg_ccn['n_sd'],
    'T':  cfg_ccn['n_sd'] + 1,
    'size': cfg_ccn['n_sd'] + 2,
}))(**ix)
```

Here, the use of a *slice* ensures that no data copying occurs when addressing `y[ix.x]`; instead, views on the existing data are used (as in the case of addressing an array with `[a:b]`, which is equivalent to `[slice(a,b)]`). The ODE can then be implemented as follows:

```
def ode_rhs(_, y, dy__dt, eq: Eqs, co, se):
    ρ_d  = eq.ρ_d(co, ρ_d=y[ix.ρ_d], T=y[ix.T])
    ρ_vs = eq.ρ_v(co, ρ_v=eq.ρ_vs(co, y[ix.T]), T=y[ix.T])
    r_w  = eq.r_w(co, x=y[ix.x])

    dr_w__dt = eq.dr_w__dt(
        co, r_w=r_w,
        ρ_v=ρ_vs * eq.RH(
            ρ_vs=ρ_vs, ρ_d=ρ_d,
            q_v=se.q_t - eq.q_l(co, se, r_w=r_w)
        ),
        ρ_o=ρ_vs * eq.RH_eq(
            co, κ=se.κ, r_w=r_w, r_d=se.r_d, T=y[ix.T]
        )
    )
    dq_v__dt = eq.dq_v__dt(
        co, se, r_w=r_w, dr_w__dt=dr_w__dt
    )

    dy__dt[ix.ρ_d] = eq.dρ_d__dt(co, se.w, ρ_d=ρ_d)
    dy__dt[ix.x]   = dr_w__dt / eq.dr_w__dx(r_w=r_w)
    dy__dt[ix.T]   = eq.dT__dt(
        co, dρ_d__dt=dy__dt[ix.ρ_d],
        dq_v__dt=dq_v__dt, ρ_d=ρ_d
    )
    return dy__dt
```

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


where in the argument list: `_` denotes that there is no explicit dependence on time, `y` is the state vector, `dy__dt` is a vector to be returned after filling it with computed derivative values, `eq` is a *namedtuple* with equation implementations (JIT-compiled or not), `co` is the constants catalog (with units or not), `se` is a *namedtuple* with parameters defining the setup, which can be created with:

```
def setup(co, si, cfg):
    return namedtuple("Setup", (se := {
        'p_d0':  100 * si.kPa,        # initial dry pres.
        't_max': 150 * si.s,          # temporal extent
        'q_t':   200 * si.dg / si.kg, # total water
        'w':     250 * si.cm / si.s,  # updraft speed
        'T_0':   300 * si.K,          # initial temp.
        'κ':   cfg['κ'],              # hygroscopicity
        'm_d': cfg['Δv_m3_stp'] * si.m**3 * co.ρ_stp,
        'r_d': cfg["r_d"] * si.m,
        'ξ':   cfg["ξ"],
    }))(**se)
```

where parameter values are chosen arbitrarily to represent a pertinent CCN activation case.

**Exercise:**

- Implement an option to handle time-varying updraft and an updraft profile resulting in an activation-deactivation cycle (e.g., as in Jensen and Nugent, 2017).

## 6 Dimensional analysis and unit testing

The design pattern used in the preceding section, whereby all physical constants are injected from the top-level scope, paves the way for dimensional analysis of the codebase. The `DimensionalAnalysis` *context manager* used in Section 2 to enable automatic plot axes labeling can now be leveraged for testing the dimensionality of all constants and arithmetic operations, which is a prerequisite for physical correctness of the implementation. To this end, we construct a customary *Arrange-Act-Assert* test calling the `ode_rhs` function in the *Act* block:

```
with DimensionalAnalysis():
    # Arrange
    co, si = constants()
    se = setup(co, si, cfg_ccn)
```

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


```
    y = [np.nan] * ix.size
    y[ix.T] *= si.K
    y[ix.p_d] *= si.Pa

    # Act
    rhs = ode_rhs(None, y, [np.nan] * ix.size, eqp, co, se)

    # Assert
    assert rhs[ix.T].check("[temperature] / [time]")
    assert rhs[ix.p_d].check("[pressure] / [time]")
    assert all(x.check("1 / [time]") for x in rhs[ix.x])
```

In the `Arrange` block above, a unit-aware constants catalog `co` is instantiated, along with a unit-aware setup `se`, and unit-aware vector `y` (here represented with a *list* instead of a NumPy array because of type heterogeneity – different state vector elements have different physical units). In the `Act` block, these objects are passed to `ode_rhs` along with the non-compiled equation catalog `eqp` and a NaN-filled `dy__dt` *list* of the same size as `y`. Calling the `ode_rhs` function with these unit-aware arguments implies that the Pint package will *raise* an *exception* on any unphysical addition or subtraction, any case of passing a dimensional quantity as an argument to logarithm, exponential or any other special function. Furthermore, the dimensionality of the results of all computations can be asserted using the `.check()` method as depicted above. Besides the dimensional analysis, the codebase is effectively also checked here for out-of-bounds access to `y` and `dy__dt`, which helps in debugging segmentation faults that can occur inside Numba JIT-compiled code.

It is worth highlighting that outside of the `DimensionalAnalysis` *context*, all calculations are done on plain *floats*, making the codebase compilable with Numba, interoperable with SciPy solvers, and free of any simulation-time overhead.

**Exercise:**

- Implement unit tests for individual equation-implementing functions in the equation catalog.

## 7 Solving for equilibrium in initial condition

Solution of the stated initial-value problem requires formulating an initial condition. Here, we pick one that yields vanishing derivatives for a parcel at rest. This implies that the initial thermodynamic state must be subsaturated (see Arabas and Shima, 2017, for discussion), and the following condition must be satisfied for each super-particle $i$:

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


$$\frac{dr_{\mathrm{w}}^{[i]}}{dt} = 0 \;\rightsquigarrow\; \mathrm{RH} - \mathrm{RH}_{\mathrm{eq}}(r_{\mathrm{w}}^{[i]}, r_{\mathrm{d}}^{[i]}, ...) = 0 \tag{7}$$

which can be solved for all super-particles using SciPy's `optimize.elementwise.find_root` routine:

```
from scipy.optimize import elementwise as sp_o

def initial_condition(eq: Eqs, co, se):
    cmn = {'co': co, 'T': se.T_0}
    RH = eq.RH(
        q_v=se.q_t,
        ρ_d=eq.ρ_d(p_d=se.p_d0, **cmn),
        ρ_vs=eq.ρ_v(p_v=eq.p_vs(**cmn), **cmn),
    )
    cmn |= {'κ': se.κ}
    root = sp_o.find_root(
        lambda x, r_d: RH - eq.RH_eq(r_w=x, r_d=r_d, **cmn),
        (se.r_d, eq.r_c(r_d=se.r_d, **cmn)),
        args=(se.r_d,)
    )
    assert all(root.success)

    y0 = np.empty(ix.size)
    y0[ix.p_d] = se.p_d0
    y0[ix.T] = se.T_0
    y0[ix.x] = eq.x(co, r_w=root.x)
    return y0, RH
```

where `initial_condition()` performs the optimization for a given setup passed as `se`, and returns a state vector `y0` and the value of initial relative humidity (computed assuming $q_{\mathrm{v}} \approx q_{\mathrm{t}}$). Figure 3 supplements the *dry-radii* histogram from Figure 1 with a histogram of the *wet radii* depicting the shift to larger sizes due to uptake of ambient vapor.

```
co, si = constants()
se = setup(co, si, cfg_ccn)
y0, RH = initial_condition(eqj, co, se)

cmn = {'bins': 8, 'weights': se.ξ, 'density': True}
```

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


```
pyplot.hist(se.r_d / µm, \
            **cmn, alpha=.25, label="dry")
pyplot.hist(eqp.r_w(co, x=y0[ix.x]) / µm, \
            **cmn, alpha=.5, label=f"wet ({RH=:.2g})")
pyplot.ylabel("number density [1/µm]")
pyplot.xlabel("radius [µm]")
pyplot.legend();
```

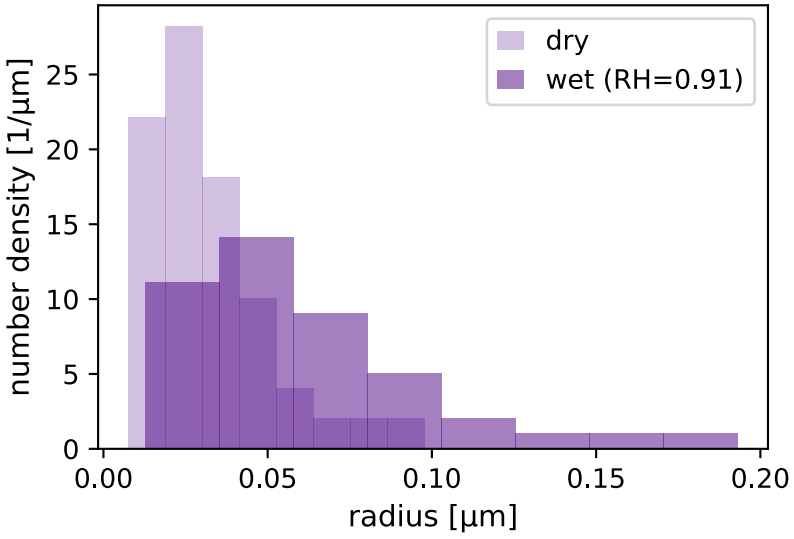


**Figure 3.** Initial condition: dry/wet radii equilibrium, see Section 7.

Since SciPy root-finding routines are not readily usable with Pint objects, the equilibration and plotting of Figure 3 were done outside the `DimensionalAnalysis` *context*.

**Exercise:**

– Implement a routine that solves for the equilibrium dry radii taking wet radii as input.

## 8 Integration using LSODA & result analysis

The numerical integration is carried out using the LSODA solver (Hindmarsh, 1983), which features adaptive timestepping, automatic stiffness detection and dynamic switching between Adams and BDF integration methods (Petzold, 1983). The adaptive timestepping helps in robustly fulfilling the stability criteria in the aerosol activation

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


problem (see, e.g., Árnason and Brown, 1971). We use the SciPy `solve_ivp` interface to LSODA (which internally calls the original FORTRAN implementation), passing to it: a JIT-compiled instance of `ode_rhs`, a *tuple* representing the temporal range to integrate over, an initial state `y0`, a *tuple* of additional arguments for `ode_rhs`, and a relative tolerance for LSODA. A wall-clock time estimate is obtained by surrounding the solver invocation with two calls to `time.perf_counter()`:

```
from time import perf_counter
import scipy.integrate as sp_i

t_start = perf_counter()
sol = sp_i.solve_ivp(
    JIT(ode_rhs),
    (0, se.t_max),
    y0,
    args=(np.empty(ix.size), eqj, co, se),
    method='LSODA', rtol=1e-4,
)
print(f"time: {(t := perf_counter() - t_start):.2g} s")
assert sol.success, sol.message
```

```
time: 1.6 s
```

The printed time includes both the numerical integration and the just-in-time compilation. This is because Numba defaults to lazy-evaluation semantics, meaning that compilation occurs only on the first call to any given function marked with `JIT`. Given the above, and given that a large part of computations take place within pre-compiled SciPy/LSODA routines, there is little to be commented on the execution time besides stating that it is modest. In Section 10, we provide a performance assessment for the collisional growth solver, discerning integration and compilation time, and deriving a speedup factor with respect to a run without JIT-compilation and without multi-threading.

In subsequent snippet, the `DimensionalAnalysis` *context manager* is used in construction of a multi-dataset Figure 4 depicting the integration results, and leveraging the automated plot axis labeling. Twin axes are used: saturation (RH) is plotted against the bottom axis; the wet radii $r_\mathrm{w}$ of the 25% of super-particles with the largest sizes are plotted against the top axis (with different line styles and colors when unactivated and activated), along with the mean radius for all droplets that fulfill the $r_\mathrm{w}^{[i]} > r_\mathrm{c}^{[i]}$ criterion.

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


```
with DimensionalAnalysis():
    co, si = constants()
    se = setup(co, si, cfg_ccn)

    z = sol.t * si.s * se.w
    p_d = sol.y[ix.p_d] * si.Pa
    T = sol.y[ix.T] * si.K
    r_w = eqp.r_w(co, x=sol.y[ix.x])
    RH = eqp.RH(
        q_v=se.q_t - eqp.q_l(co, se, r_w=r_w),
        ρ_vs=eqp.ρ_v(co, p_v=eqp.p_vs(co, T=T), T=T),
        ρ_d=eqp.ρ_d(co, p_d=p_d, T=T),
    )
    r_c = eqp.r_c(co, se.κ, se.r_d[:,None], T[None,:])

    si.setup_matplotlib()
    prim = pyplot.gca()
    prim.plot(RH, z, label="RH", lw=3/2)
    prim.axvline(100, label="100%", lw=.5)
    prim.legend(loc='lower center')
    prim.xaxis.set_units(si.percent)
    prim.yaxis.set_units(si.m)

    twin = prim.twiny()
    lines = {}
    for i in range(3 * r_w.shape[0] // 4, r_w.shape[0]):
        for cond, ls, color in [
            (r_w < r_c, '--', 'r'),
            (r_w >= r_c, '-', 'k')
        ]:
            lines[color], = twin.plot(
                np.where(cond[i], r_w[i], np.nan),
                z, ls=ls, lw=1/3, color=color
            )
    lines['r'].set_label(r'wet radii $<r_\text{c}$')
    lines['k'].set_label(r'wet radii $\,≥\,r_\text{c}$')

    twin.plot(
```

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


```
        np.sum(m := np.where(r_w > r_c, r_w, 0), axis=0)
        / np.maximum(m := np.count_nonzero(m, axis=0), 1)
        * np.where(m, 1, np.nan),
        z,
        label=r"mean ($≥\,r_\text{c}$)", lw=2, ls=':'
    )
    twin.legend(loc='lower right')
    twin.xaxis.set_units(si.um)

    n_a = (r_w[:,-1]>r_c[:,-1]) @ se.ξ / se.m_d * co.ρ_stp
    err = np.amax(se.ξ) / se.m_d * co.ρ_stp
    print("n_a:", f"{n_a.to(u := si.cm**-3):.4g~} ± {err.to(u):.3g~}")
```

```
n_a: 1091 / cm ** 3 ± 182 / cm ** 3
```

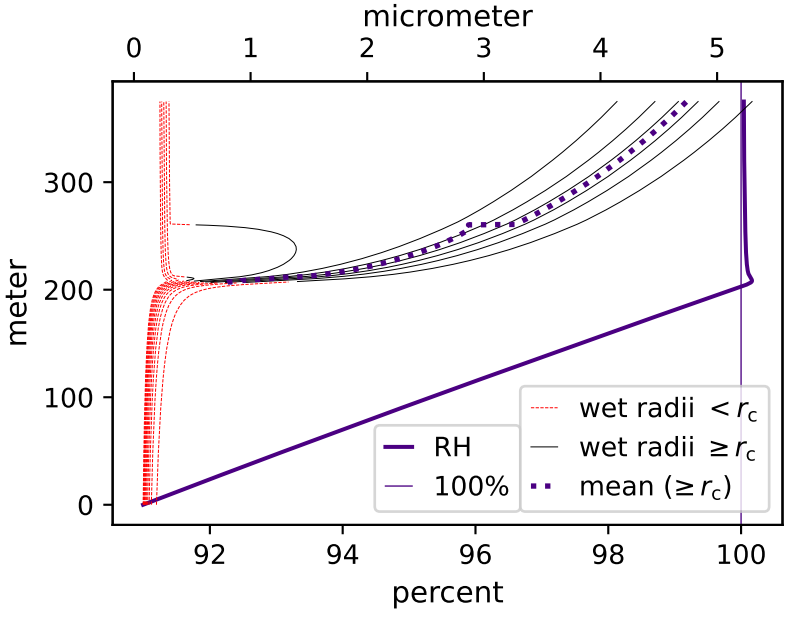


**Figure 4.** Saturation (bottom axis) and size-spectral profiles (top axis) as a function of height (vertical axis) for a vertically displaced air parcel, see Section 8.

The print statement at the end outputs an estimate of the volume concentration (at standard temperature and pressure) of the activated droplets. The uncertainty concerns numerical spectrum representation only, and

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


is estimated to be equal to the concentration of droplets corresponding to the super-particle with the largest multiplicity (for the given setup, all super-particles have uniform multiplicities).

The figure is constructed in analogy to figures 8-10, 12 & 14 from Howell (1949) – a seminal numerical cloud microphysics work predating digital computers. The saturation curve features a characteristic maximum at cloud base, which is where the onset of rapid growth leads to CCN activation. The saturation then approaches the RH=100% asymptote, and some of the super-particles are undergoing what is referred to as ripening (in analogy to Ostwald ripening), whereby larger drops effectively grow at the expense of the vapor evaporated from smaller droplets (see, e.g., Wood et al., 2002). The ripening leads to deactivation, which is readily observable in the wet-radii characteristic, as well as in the mean radius profile, where it is associated with a sharp change in slope. Occurrence of ripening is contingent on high aerosol loading.

To sum up, in under 300 lines of pure-Python code (an order of magnitude less than, e.g., the PyrCel codebase), that covers setup, simulation, visualization, and a dimensional-analysis test, we have just achieved a reasonably complete cloud-parcel model with moving-sectional/particle-resolved aerosol-cloud microphysics, capable of resolving condensation, activation, evaporation, ripening and deactivation, and completing a typical simulation within 1.6 s.

**Exercises:**

- Explore performance of other ODE solvers available in SciPy (different settings of the 'method' argument), note the difference between solvers suited for stiff and non-stiff systems.
- Corroborate data from multiple runs of the parcel model (e.g., fraction of activated droplets, maximal supersaturation) with output from parameterization of the activation process used in climate models (see, e.g., Ghan et al., 2011).

## 9 Multi-threaded JIT-compiled SDM code

In Shima et al. (2009), the Super-Droplet Method (SDM) Monte-Carlo scheme was proposed for representing collisional growth in super-particle cloud microphysics models. The SDM algorithm is a probabilistic alternative to the deterministic model of the process embodied in the Smoluchowski coagulation equations. Adopting their notation and terminology, each step of SDM alters the size and multiplicity attributes in a population of droplets in a way to represent coagulation of the particles occurring with probability specified by a *coagulation kernel* function.

SDM, by design, is not subject to the "curse of dimensionality" that hampers application of other methods when multiple particle attributes need to be resolved in a simulation (see, e.g., discussion in Grabowski et al., 2019). The algorithm is embarrassingly parallel, has linear complexity, constant (i.e. not growing) state vector size, and no numerical diffusion.

The listing below constitutes a fully functional implementation of the algorithm coded with the assumption of non-zero multiplicities, and handling evolution of a single extensive attribute $m$ (plus multiplicity $\xi$). In addition to using

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


Numba's JIT compilation, the implementation also relies on Numba's multi-threading facility – note the `prange` iteration instead of an ordinary `range`. Variables within the `sdm()` function are named to match the notation in Shima et al. (2009, points 1-5 in section 5 therein). Arguments are annotated with *type hints*, which are part of Python syntax allowing external tools to check type consistency, but are not enforced by the interpreter.

```
from collections import abc
from numba import prange

@JIT(parallel=True)
def sdm(
    n_steps: int,
    *,
    ξ: abc.MutableSequence[int],
    m: abc.MutableSequence[float],
    kern: abc.Callable[[float, float], float],
    Δt: float,
    Δv: float,
):
    n_part = len(ξ)
    n_pair = n_part // 2
    perm = np.arange(n_part)
    for _ in range(n_steps):                                    ⓪
        np.random.shuffle(perm)                                 ①
        pairs = perm[: 2 * n_pair].reshape(-1, 2)               ②
        p_ratio = n_part * (n_part - 1) / 2 / n_pair            ③
        for α in prange(n_pair):                                ④
            j, k = pairs[α]                                     ⑤
            p_jk = kern(m[j], m[k]) * Δt / Δv
            if ξ[j] < ξ[k]:
                j, k = k, j
            p_α = ξ[j] * p_ratio * p_jk                         ⑥
            φ = np.random.random()                              ⑦
            γ = p_α // 1 + ((p_α - p_α // 1) > φ)               ⑧
            if γ != 0:
                γ = min(γ, (ξ[j] / ξ[k]) // 1)                  ⑨
                if ξ[j] != γ * ξ[k]:
                    ξ[j] -= γ * ξ[k]
                    m[k] += γ * m[j]
```

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


```
            else:
                ξ[j] = ξ[k] // 2
                ξ[k] -= ξ[j]
                m[k] += γ * m[j]
                m[j] = m[k]
```

Describing the functionality of the lines marked with circled numbers: ⓪ a single call to the `sdm()` function results in advancing the system state by `n_steps` timesteps of length `Δt`; each step involves ① permuting the super particles to ② select a random set of `n_pair` non-overlapping pairs among all `n_part` super particles in the considered volume `Δv`; ③ computing the `p_ratio` of all possible particle unordered pairs to the number of considered candidate pairs `n_pair`; ④ enumerating across all candidate pairs with `α` numbering the pair, and ⑤ `j` & `k` numbering the super droplets within a pair; ⑥ evaluating the probability `p_α` of collision within the super-droplet pair `α` as a product of the coagulation kernel, the timestep-to-cell-volume ratio `Δt/Δv`, the `p_ratio` and `ξ_j` where `j` is chosen to point to the super droplet of higher multiplicity in the pair; ⑦ shuffling a random number `φ` from a uniform distribution (note that NUMBA features thread-safe random-number generation with each thread producing independent streams of random numbers); ⑧ evaluating the `γ` factor, which for `p_α<1` is a Boolean flag resulting from comparison of the probability of coagulation with the random number from the $[0,1)$ range, while in the case of `p_α≥1`, `γ` is augmented by the number of "certain" coagulations to represent repeated collision events within a single step; ⑨ depending on the value of the `ξ[j]/ξ[k]` ratio, the number of repeated events may need to be truncated incurring a collision-event deficit (see Ware et al., 2026, for discussion); finally, updating particle attribute vectors by setting new values to attributes `ξ[j]`, `m[k]`, and if `ξ[j]` is equal to `γ·ξ[k]`, also to `ξ[k]` and `m[j]`.

**Exercise:**

- Explore the scalability of the `sdm()` function by analyzing its execution time as a function of the size of the state vector `n_sd` and the number of threads used for computations (which can be altered using `numba.set_num_threads()`).

## 10 Verification against analytic solution

For the particle coagulation problem, an analytic solution to the Smoluchowski coagulation equation for additive kernel and exponential initial spectrum is known (Safranov, 1962; Golovin, 1963). In Shima et al. (2009), it is used for verification of the method. The following snippets reproduce Fig. 2a from the paper, presented here as Figure 5 (note the different value of `n_sd`).

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


```
si = physics.si

cfg_sdm = {
    'Δv': (Δv := 1e6 * si.m**3),
    'Δt': 1 * si.s,
    'n0': (n0 := 2**23 / si.m**3),
    'norm': (norm := n0 * Δv),
    'dist': sp_st.expon(loc=0, scale=1 / norm),
    'n_sd': 2**16,
    'b': (b := 1.5e3 / si.s),
    'kernel': JIT(lambda m1, m2: b * (m1 + m2)),
}
```

The Safranov-Golovin solution (see eq. 20 in Safranov, 1962; and eq. 20 in Golovin, 1963) can be implemented using the SciPy's `special.ive` implementation of the exponentially scaled modified Bessel function (note that the equations in the referenced papers use the numerically challenging unscaled Bessel function, here the exponentially scaled variant is used which introduces additional terms in the exponents):

```
from scipy import special as sp_sp

def saf_gol_eq20(mass, time, cfg):
    x0 = 1 / cfg['norm']
    tau = 1 - np.exp(-cfg['n0'] * x0 * cfg['b'] * time)
    sqrt_tau = np.sqrt(tau)
    return (
        (1 - tau) / (mass * sqrt_tau)
        * sp_sp.ive(1, 2 * mass / x0 * sqrt_tau)
        * np.exp(-(1 + tau - 2 * sqrt_tau) * mass / x0)
    )
```

To reproduce the figure, solution is sought at times of 1200 s, 2400 s and 3600 s by calling the `sdm()` function in a loop in which the first iteration is used merely for storing the initial condition (and for JIT compilation of `sdm()`). Prior to the execution, the Numba random number generator seed is set, which requires calling the `np.random.seed()` function within jit-compiled code. Execution times are measured using `timeit()` and printed:

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


```
from timeit import timeit

cmn = {
    'kern': cfg_sdm['kernel'],
    'Δt': cfg_sdm['Δt'],
    'Δv': cfg_sdm['Δv'],
}
cmn['m'], cmn['ξ'] = quantile_sample(
    dist=cfg_sdm['dist'],
    n_sd=cfg_sdm['n_sd'],
    norm=cfg_sdm['norm']
)

jit(lambda: np.random.seed(44))()

output, times, prev = {}, [], 0
for step in (0, 1200, 2400, 3600):
    calc = lambda: sdm(step - prev, **cmn)
    times += [timeit(calc, number=1)]
    print(f"{prev=:-4d} … {step=:-4d}: {times[-1]:.1f} s")
    output[step] = {k: cmn[k].copy() for k in ('m', 'ξ')}
    prev = step
assert all(cmn['ξ'] > 0)
```

```
prev=   0 … step=   0: 0.8 s
prev=   0 … step=1200: 1.1 s
prev=1200 … step=2400: 1.1 s
prev=2400 … step=3600: 1.1 s
```

where the time of the first iteration (performing zero steps) corresponds to JIT-compilation overhead.

In order to gauge the speedup with respect to execution without NUMBA, the `.py_func` property of the `sdm()` JIT-decorated function is invoked to advance the simulation state by another 1200 s, discarding results for simplicity.

```
t_py = timeit(lambda: sdm.py_func(1200, **cmn), number=1)
print(f"time w/o Numba: {t_py:.0f} s")
print(f"speedup: ×{t_py / min(times[1:]):.0f}")
```

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


```
time w/o Numba: 106 s
speedup: ×95
```

The measured speedup (estimated excluding the compilation time at first call, hence `times[1:]`) is substantial and clearly supports employment of JIT compilation (and multi-threading, which alone can provide at most an $n$-fold speedup, where $n$ is the number of threads).

The following code generates Figure 5, which juxtaposes SDM simulation results with the analytic curves, both cast in terms of the contribution to the total mass – in analogy to Fig. 2a in Shima et al. (2009).

```python
x_of_m = lambda mass: np.log(mass) / 3
m_of_x = lambda x: np.exp(3 * x)
x_range, x_bins = (-11, -5), 44
x_sdm, y_sdm, x_anl, y_anl = {}, {}, {}, {}

for step, data in reversed(output.items()):
    y_sdm[step], x_sdm[step], _ = pyplot.hist(
        x=x_of_m(data['m']),
        weights=data['ξ'] / cfg_sdm['norm'] * data['m'],
        bins=x_bins, range=x_range, density=True,
        label=f't={step * cfg_sdm['Δt']:.0f} s',
        alpha=.1 + .5 * step / 3600,
    )
    y_fun = lambda m, x: m * saf_gol_eq20(
        time=1e-10 if step == 0 else step * cfg_sdm['Δt'],
        mass=m, cfg=cfg_sdm,
    ) * cfg_sdm['norm'] * np.diff(m_of_x(x)) / np.diff(x)
    x_anl[step] = x_sdm[step][1:] - np.diff(x_sdm[step])/2
    y_anl[step] = y_fun(m_of_x(x_anl[step]), x_sdm[step])
    pyplot.step(
        x=x_anl[step], y=y_anl[step],
        label='discretised analytic' if step == 0 else '',
        color='k', ls='-', where='mid'
    )
pyplot.ylabel('pdf(x) $\\cdot$ m(x)')
pyplot.xlabel('$x = \\ln(\\sqrt[3]{\\text{m/m}_0})$')
pyplot.legend();
```

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


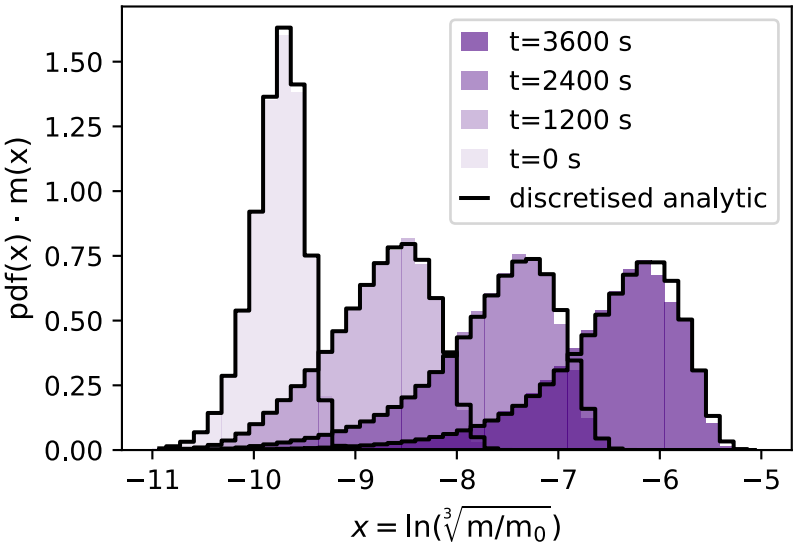


**Figure 5.** Size spectrum at four different times: analytic solution (dashed lines) and SDM results, see Section 10.

The match against the analytic solution can be quantified and programmatically *asserted* with:

```
for step in output:
    np.testing.assert_allclose(
        actual=y_sdm[step], desired=y_anl[step],
        atol=0.1, rtol=0.1
    )
```

where the `atol` and `rtol` values correspond to absolute and relative tolerances, respectively (i.e., $|\texttt{actual} - \texttt{desired}| \leq \texttt{atol} + \texttt{rtol} \cdot |\texttt{desired}|$).

**Exercise:**

- Using SCIKIT-LEARN package, train a compact neural-network surrogate for a prescribed collision kernel, export the fitted weights, and replace only the kernel evaluator in sdm() with a Numba-compatible forward pass of the trained network; compare the kernel error, SDM spectra, and runtime against the reference evaluator.

## Summary

This work is meant to help in teaching and learning both: cloud microphysics modeling and research software engineering in PYTHON. Among the covered modeling concepts, there are: *particle-resolved/moving-sectional*

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


aerosol size-spectrum representation, aerosol *hygroscopicity*, cloud thermodynamics, classification of particles into unactivated (haze) and activated droplets, size-spectral statistics, and *coagulation kernel*. Among the model engineering concepts: dimensional analysis including its usage for software testing and plot axis labelling automation, just-in-time compilation, data structures relevant for computational numerics, multi-threaded concurrency, and a range of applications for canonical algorithms packaged in NumPy, SciPy (probability distribution sampling, root finding, special functions, ODE integration, random number generation). By presenting complete implementations of two cloud modeling "classics", we offer open-source and annotated materials that are fit for self-study or coursework. Importantly, the presented code does not trade off performance, maintainability abstractions or physical sophistication, and can be readily used in or extended towards research applications in cloud microphysics or other fields. While the paper is focused on cloud microphysics modeling, the technical solutions presented here apply equally well to other subdomains of computational research. Quoting van Rossum (1998): *Python code looks like executable pseudocode*, which we have aimed to highlight here, and which greatly helps to avoid cluttering physics-focused logic with *boilerplate code* contributing to its maintainability and auditability.

Several of the technological solutions described in this work were developed in the course of building the PySDM cloud modeling package (Bartman et al., 2022; de Jong et al., 2023). In addition to implementations of the SDM algorithm, and an adiabatic parcel framework akin to the one used here, the package consists of representation of numerous other processes, and include examples reproducing results from literature, which are maintained with the project. Both the presented implementation and PySDM are engineered in pure Python and depend only on packages that offer compilation-free installation. Moreover, to the authors' knowledge, there is novelty in the way Pint, NumPy and SciPy/LSODA are coupled here, offering both zero-overhead execution for simulations, and unit-aware execution for testing or plotting, using the very same codebase. This is in contrast to other Python aerosol-cloud modeling packages, e.g., pyrcel (Rothenberg and Wang, 2017) or PyPartMC (D'Aquino et al., 2024) whose codebase is not compatible with the dimensional analysis workflow described here.

It is important to acknowledge here that analogous unit-aware implementations are feasible with other technologies. The Julia language features a built-in just-in-time compiler (based on the same LLVM backend as Numba), and its package ecosystem includes Unitful.jl and DifferentialEquations.jl – packages providing even richer functionalities than their Python counterparts Pint and SciPy.odeint; moreover Julia has strong support among Jupyter tools and services (including Colab). C++ language features templating syntax enabling efficient representation of physical units what has been leveraged in the Boost.Units package, while the Boost.odeint package offers flexible object-oriented differential equations solvers. Still, the sheer popularity of Python, fueled in recent years by its widespread use in the machine-learning ecosystem, makes it particularly suitable for development of active learning materials addressed towards a general quantitative audience, not necessarily math- or CS- focused- which enables to start without introducing prerequisites as would likely be needed in the case of Julia and C++, respectively. Something that sets Python apart is that tools for binding Python code from

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


other languages (e.g., calling PYTHON from C++ or JULIA, and vice versa – calling C++ or JULIA from PYTHON) are readily available, and in the case of MATLAB and IDL are included by default within their distributions. Consequently, the concepts presented here extend beyond cloud microphysics and are not limited to PYTHON.

## Code availability

The paper source in a form of a QUARTO/JUPYTER notebook is enclosed as electronic supplement. Entire simulation code is included in the paper as listings (as well as in the notebook).

## Author contributions

The paper idea originated from preparation of materials for, and from discussions during two recent workshops: "Mini-workshop on particle-based modeling" (September 2025, University of Hyogo, Kobe, Japan) organized by SS and "Hacking aerosol-cloud μ-physics modeling concepts in Python" (January 2026, University of Washington, Seattle, WA) organized by RW. The code presented in the paper was developed by SA, EW, AŻ and DK. It adapts solutions engineered in the PySDM package and tutorials, to which CS, MA, DP and JB contributed, along with numerous PySDM project contributors. The paper draft was written by SA, all co-authors contributed to refactoring and annotating the code, and to the final text.

## Acknowledgments

SA and EW acknowledge mobility support from the European Commission's Erasmus+ program. SA acknowledges University of Hyogo support for participation in the workshop in Japan. AŻ acknowledges support from the Polish National Science Centre (grant no. 2020/39/D/ST10/01220). DP acknowledges funding from NERC CLOSURE project (grant no. NE/W001713/1).

.

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


## References

Arabas, S. and Shima, S.: On the CCN (de)activation nonlinearities, Nonlin. Proc. Geophys., 24, https://doi.org/10.5194/npg-24-535-2017, 2017.

Arabas, S., Jaruga, A., Pawlowska, H., and Grabowski, W. W.: libcloudph++ 1.0: a single-moment bulk, double-moment bulk, and particle-based warm-rain microphysics library in C++, Geosci. Model Dev., 8, https://doi.org/10.5194/gmd-8-1677-2015, 2015.

Arms, S., Chastang, J., Grover, M., Thielen, J., Wilson, M., and Dirks, D.: Introducing Students to Scientific Python for Atmospheric Science, Bull. Amer. Meteorol. Soc., 101, https://doi.org/10.1175/BAMS-D-20-0069.1, 2020.

Árnason, G. and Brown, P. S., J.: Growth of Cloud Droplets by Condensation: A Problem in Computational Stability, J. Atmos. Sci., 28, https://doi.org/10.1175/1520-0469(1971)028<0072:GOCDBC>2.0.CO;2, 1971.

Bartman, P., Bulenok, O., Górski, K., Jaruga, A., Łazarski, G., Olesik, M. A., Piasecki, B., Singer, C. E., Talar, A., and Arabas, S.: PySDM v1: particle-based cloud modelling package for warm-rain microphysics and aqueous chemistry, J. Open Source Softw., 7, https://doi.org/10.21105/joss.03219, 2022.

Bolton, D.: The Computation of Equivalent Potential Temperature, Mon. Weather Rev., 108, https://doi.org/10.1175/1520-0493(1980)108<1046:TCOEPT>2.0.CO;2, 1980.

D'Aquino, Z., Arabas, S., Curtis, J. H., Vaishnav, A., Riemer, N., and West, M.: PyPartMC: A Pythonic interface to a particle-resolved, Monte Carlo aerosol simulation framework, SoftwareX, 25, https://doi.org/10.1016/j.softx.2023.101613, 2024.

de Jong, E. K., Singer, C. E., Azimi, S., Bartman, P., Bulenok, O., Derlatka, K., Dula, I., Jaruga, A., Mackay, J. B., Ward, R. X., and Arabas, S.: New developments in PySDM and PySDM-examples v2: collisional breakup, immersion freezing, dry aerosol initialization, and adaptive time-stepping, J. Open Source Softw., 8, https://doi.org/10.21105/joss.04968, 2023.

DeCaria, A. J. and Petty, G. W.: Python Programming and Visualization for Scientists (2nd Ed.), Sundog Publishing LLC, Madison, WI, https://web.archive.org/web/20251017164724/https://sundogpublishing.com/products/python-programming-and-visualization-for-scientists-2nd-ed, 2020.

Dziekan, P., Jensen, J. B., Grabowski, W. W., and Pawlowska, H.: Impact of Giant Sea Salt Aerosol Particles on Precipitation in Marine Cumuli and Stratocumuli: Lagrangian Cloud Model Simulations, J. Atmos. Sci., 78, https://doi.org/10.1175/JAS-D-21-0041.1, 2021.

Gatley, D. P., Herrmann, S., and Kretzschmar, H.-J.: A Twenty-First Century Molar Mass for Dry Air, HVACR Res., 14, https://doi.org/10.1080/10789669.2008.10391032, 2008.

Ghan, S. J., Abdul-Razzak, H., Nenes, A., Ming, Y., Liu, X., Ovchinnikov, M., Shipway, B., Meskhidze, N., Xu, J., and Shi, X.: Droplet nucleation: Physically-based parameterizations and comparative evaluation, J. Adv. Model. Earth Sys., 3, https://doi.org/10.1029/2011MS000074, 2011.

Golovin, A.: The solution of the coagulation equation for raindrops. Taking condensation into account, Bull. Acad. Sci. SSSR Geophys. Ser., 148, http://mathnet.ru/dan27630, (in Russian), 1963.

Grabowski, W. W., Morrison, H., Shima, S.-I., Abade, G. C., Dziekan, P., and Pawlowska, H.: Modeling of Cloud Microphysics: Can We Do Better?, Bull. Amer. Meteorol. Soc., 100, https://doi.org/10.1175/BAMS-D-18-0005.1, 2019.

Granger, B. E. and Pérez, F.: Jupyter: Thinking and Storytelling With Code and Data, Computing in Science & Engineering, 23, https://doi.org/10.1109/MCSE.2021.3059263, 2021.

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


Grecco, H. E.: Pint: makes units easy, https://web.archive.org/web/20260225082608/https://pint.readthedocs.io/en/0.25.2/, 2025.

Ham, D.: Object-oriented Programming in Python for Mathematicians, https://web.archive.org/web/20250906063850/https://object-oriented-python.github.io/, 2021.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E.: Array programming with NumPy, Nature, 585, https://doi.org/10.1038/s41586-020-2649-2, 2020.

Harrison, R. G. and Ambaum, M. H. P.: Enhancement of cloud formation by droplet charging, Proc. Royal Soc. A, 464, https://doi.org/10.1098/rspa.2008.0009, 2008.

Hindmarsh, A. C.: ODEPACK, A Systematized Collection of ODE Solvers, in: Scientific Computing, edited by Stepleman, R. S., https://web.archive.org/web/20260301140706/https://academicweb.nd.edu/~powers/ame.60636/hindmarsh1983.pdf, 1983.

Howell, W. E.: The growth of cloud drops in uniformly cooled air, J. Atmos. Sci., 6, https://doi.org/10.1175/1520-0469(1949)006<0134:TGOCDI>2.0.CO;2, 1949.

Hunter, J. D.: Matplotlib: A 2D graphics environment, Comput. Sci. Eng., 9, https://doi.org/10.1109/MCSE.2007.55, 2007.

Irving, D., Hertweck, K., Johnston, L., Ostblom, J., Wickham, C., and Wilson, G.: Research Software Engineering with Python: Building software that makes research possible, Chapman and Hall/CRC, Boca Raton, FL, https://web.archive.org/web/20260113031519/https://third-bit.com/py-rse/, 2021.

Jacobson, M. Z.: Fundamentals of Atmospheric Modeling, Cambridge University Press, https://doi.org/10.1017/CBO9781139165389, 2012.

Jensen, J. B. and Nugent, A. D.: Condensational Growth of Drops Formed on Giant Sea-Salt Aerosol Particles, J. Atmos. Sci., 74, https://doi.org/10.1175/JAS-D-15-0370.1, 2017.

Lam, S. K., Pitrou, A., and Seibert, S.: Numba: a LLVM-based Python JIT compiler, in: LLVM '15: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, https://doi.org/10.1145/2833157.2833162, 2015.

Lin, J. W.-B.: Why Python Is the Next Wave in Earth Sciences Computing, Bull. Amer. Meteorol. Soc., 93, https://doi.org/10.1175/BAMS-d-12-00148.1, 2012.

Matsushima, T., Nishizawa, S., and Shima, S.: Overcoming computational challenges to realize meter- to submeter-scale resolution in cloud simulations using the super-droplet method, Geosci. Model Dev., 16, https://doi.org/10.5194/gmd-16-6211-2023, 2023.

May, R. M., Goebbert, K. H., Thielen, J. E., Leeman, J. R., Camron, M. D., Bruick, Z., Bruning, E. C., Manser, R. P., Arms, S. C., and Marsh, P. T.: MetPy: A Meteorological Python Library for Data Analysis and Visualization, Bull. Amer. Meteorol. Soc., 103, https://doi.org/10.1175/BAMS-D-21-0125.1, 2022.

Meurer, A., Reines, A., Gommers, R., Fang, Y.-L. L., Kirkham, J., Barber, M., Hoyer, S., Müller, A., Zha, S., Shanabrook, S., Gacha, S. J., Lezcano-Casado, M., Fan, T. J., Reddy, T., Passos, A., Kwon, H., Oliphant, T., and Consortium for Python Data API Standards: Python Array API Standard: Toward Array Interoperability in the Scientific Python Ecosystem, SciPy Proceedings, https://doi.org/10.25080/gerudo-f2bc6f59-001, 2023.

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


Morrison, H., Chandrakar, K. K., Rehme, M., Bryan, G. H., Chen, S., Grabowski, W. W., Lawson, R. P., McFarquhar, G. M., Shaw, R. A., and Xue, L.: A virtual cloud physics laboratory, Bull. Amer. Meteorol. Soc., 107, https://doi.org/10.1175/BAMS-D-25-0026.1, 2026.

Petters, M. D.: Modules for Active Atmospheric Learning: Encoding Student-Centered, Open-Access Science Education, Bull. Amer. Meteorol. Soc., 103, https://doi.org/10.1175/BAMS-D-20-0072.A, 2022.

Petters, M. D. and Kreidenweis, S. M.: A single parameter representation of hygroscopic growth and cloud condensation nucleus activity, Atmos. Chem. Phys., 7, https://doi.org/10.5194/acp-7-1961-2007, 2007.

Petzold, L.: Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations, SIAM J. Sci. Stat. Comput., 4, https://doi.org/10.1137/0904010, 1983.

Rothenberg, D. and Wang, C.: An aerosol activation metamodel of v1.2.0 of the pyrcel cloud parcel model: development and offline assessment for use in an aerosol–climate model, Geosci. Model Dev., 10, https://doi.org/10.5194/gmd-10-1817-2017, 2017.

Safranov, V. S.: A particular solution of the coagulation equation, Bull. Acad. Sci. SSSR Geophys. Ser., 147, https://mathnet.ru/dan27172, in Russian, 1962.

Seinfeld, J. H. and Pandis, S. N.: Atmospheric Chemistry and Physics: From Air Pollution to Climate Change, John Wiley & Sons, third Edition, 2016.

Shima, S., Kusano, K., Kawano, A., Sugiyama, T., and Kawahara, S.: The super-droplet method for the numerical simulation of clouds and precipitation: a particle-based and probabilistic microphysics model coupled with a non-hydrostatic model, Q. J. Royal Meteorol. Soc., https://doi.org/10.1002/qj.441, 2009.

Squires, P.: The growth of cloud drops by condensation, Aust. J. Sci. Res. A Phys. Sci., 5, https://doi.org/10.1071/CH9520059, 1952.

Starkman, N., Price-Whelan, A. M., and Nibauer, J.: unxt: A Python package for unit-aware computing with JAX, J. Open Source Softw., 10, https://doi.org/10.21105/joss.07771, 2025.

Topping, D. and Bane, M., eds.: Introduction to Aerosol Modelling: From Theory to Code, Wiley, https://doi.org/10.1002/9781119625728, 2022.

Trauth, M. H.: Python Recipes for Earth Sciences, Springer, https://doi.org/10.1007/978-3-031-56906-7, second Edition, 2024.

van Rossum, G.: Glue It All Together With Python, in: OMG-DARPA-MCC Workshop on Compositional Software Architecture, Monterey, California, January 6-8, 1998, https://web.archive.org/web/19980614041513/http://www.objs.com/workshops/ws9801/papers/paper070.html, 1998.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and SciPy 1.0 Contributors: SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python, Nat. Methods, 17, https://doi.org/10.1038/s41592-019-0686-2, 2020.

Ware, E., Bartman-Szwarc, P., Igel, A. L., and Arabas, S.: Addressing the Collision-Coalescence Deficit in the Super-Droplet Monte Carlo Scheme with Adaptive Time-Stepping, J. Adv. Model. Earth Sys. (accepted); arXiv pre-print, https://doi.org/10.48550/arXiv.2509.05536, 2026.

https://doi.org/10.5194/egusphere-2026-3906
Preprint. Discussion started: 21 July 2026


Wood, R., Irons, S., and Jonas, P. R.: How Important Is the Spectral Ripening Effect in Stratiform Boundary Layer Clouds? Studies Using Simple Trajectory Analysis, J. Atmos. Sci., https://doi.org/10.1175/1520-0469(2002)059<2681:HIITSR>2.0.CO;2, 2002.