Appendix A: Supplementary Numerical Results for Chapter 2

This appendix summarizes the algorithm implemented in Python 3.12.3 (NumPy 1.26.4 and SciPy 1.11.4) to correct the ionization degree estimated by QEOS for partially ionized plasmas generated by hypervelocity impacts on Ubuntu Linux (kernel 6.14.0-35-generic, x86_64). Corresponding results are shown in Figures 2-2. Only the essential steps relevant to physical interpretation are described. All P4P simulation results presented in this dissertation were processed using this method.

```
import numpy as np
from scipy.optimize import root_scalar

# ============================================================
# C.1 Phase discrimination using bonding energy (Eq. 2-8)
# ============================================================

c103 = 10.0 / 3.0
c53  = 5.0 / 3.0
NA   = 6.022e23  # 1/mol
eVJ  = 1.602176634e-19  # J/eV
# Prefactor for electron number density (Eq. 2-3)
prefactor = (1.0 / (2.0 * np.pi**2)) * (2.0 * me / hbar**2 * eVJ)**1.5
# Units: [m^-3 eV^-3/2]


def _compute_params(rho0_gcc, A_gmol, Znc):
    """
    Helper for computing b and E0 (Takabe 1993; see text).
    rho0_gcc: reference density [g/cm^3]
    A_gmol  : atomic/molecular mass [g/mol]
    Znc     : effective charge number (non-core / average-atom input)
    """
    y = rho0_gcc / (Znc * A_gmol)                # normalized density
    za0 = 1.0 / (1.0 + np.sqrt(0.148 / y))
    slpre = Znc ** c103
    cbd1 = 0.1 * (y * za0) ** c53
    pe0rosld = slpre * cbd1 * 100.0             # [Pa] (as implemented in code)
    return y, za0, slpre, cbd1, pe0rosld

def compute_b(rho0_gcc, A_gmol, Znc, K_Pa):
    """Compute b parameter used in Eq. (2-8)."""
    y, za0, slpre, cbd1, pe0rosld = _compute_params(rho0_gcc, A_gmol, Znc)
```

```
    term1 = 2.5 * za0 * np.sqrt(0.148 / y)
    term2 = 3.0 * (1.0 - K_Pa / pe0rosld)
    return term1 + term2

def compute_E0(rho0_gcc, A_gmol, Znc, b_val):
    """Compute E0 parameter used in Eq. (2-8), returned in [eV/atom]."""
    y, za0, slpre, cbd1, _ = _compute_params(rho0_gcc, A_gmol, Znc)
    # Convert to pressure-like scale (kept consistent with original implementation)
    cbd1 = cbd1 * slpre * 1e-1 * 1e14          # [Pa]
    rho_atoms_m3 = (rho0_gcc * NA / A_gmol) * 1e6  # [atoms/m^3]  (g/cm^3 -> kg/m^3 f
actor embedded)
    E0_J_per_atom = 3.0 * cbd1 / (b_val * rho_atoms_m3)
    return E0_J_per_atom / eVJ                 # [eV/atom]

def bonding_energy_Eb(rho_gcc, rho0_gcc, E0_eV, b_val):
    """
    Eq. (2-8): Eb(rho) = E0 * exp( b * (1 - 1/(rho/rho0)) )
    rho_gcc may be array-like.
    """
    dam = rho_gcc / rho0_gcc
    return E0_eV * np.exp(b_val * (1.0 - 1.0 / dam))


# ============================================================
# C.2 Electron density integral and Fermi level solver (Eqs. 2-3, 2-5)
# ============================================================

def electron_density(mu_eV, Te_eV, Emin_eV=0.0, Emax_eV=100.0, N=1000, prefactor=Non
e):
    """
    Electron number density from finite-range integration of FD distribution.
    This corresponds to Eq. (2-3) with a lower energy cutoff Emin (Eq. 2-7).
    - mu_eV: chemical potential [eV]
    - Te_eV: electron temperature [eV]
    - Emin_eV: lower bound of integration [eV] (0 for total, V_barrier for free)
    - prefactor: constant factor converting integral to density (defined in main cod
e)
    """
    eps = np.linspace(Emin_eV, Emax_eV, N)
    dE  = eps[1] - eps[0]
    arg = (eps - mu_eV) / np.maximum(Te_eV, 1e-10)
    arg = np.clip(arg, -700, 700)
    f   = 1.0 / (1.0 + np.exp(arg))
    integral = np.trapz(np.sqrt(eps) * f, dx=dE)
```

```
    ne = prefactor * integral
    return np.clip(ne, 0.0, 1e32)

def solve_mu_from_ne(ne_target, Te_eV, prefactor, Emin_eV=0.0, Emax_eV=100.0, N=100
0):
    """
    Solve for chemical potential mu such that electron_density(mu, Te) = ne_target.
    Uses bracketing + Brent's method (robust).
    Returns (mu, ok_flag).
    """
    Te_eV = float(max(Te_eV, 1e-12))

    def residual(mu):
        return electron_density(mu, Te_eV, Emin_eV=Emin_eV, Emax_eV=Emax_eV, N=N, pre
factor=prefactor) - ne_target

    # Simple bracket around 0 eV; can be widened if needed
    lo, hi = -200.0, 200.0
    rlo, rhi = residual(lo), residual(hi)

    # Widen bracket if necessary
    for _ in range(60):
        if rlo < 0 and rhi > 0:
            break
        lo -= 50.0
        hi += 50.0
        rlo, rhi = residual(lo), residual(hi)
    else:
        return 0.0, 0  # failed to bracket

    sol = root_scalar(residual, bracket=[lo, hi], method="brentq", xtol=1e-6)
    return float(sol.root), 1


# ============================================================
# C.3 Ionization-degree correction (Eqs. 2-6, 2-7 and phase selection)
# ============================================================

def ionized_fraction(mu_eV, Te_eV, Vbarrier_eV, prefactor, Emax_eV=100.0, N=1000):
    """
    Fraction of 'truly free' electrons:
        frac = ne(E > Vbarrier) / ne(total)
    This corresponds to Eq. (2-7) divided by Eq. (2-3).
    """
```

```
    ne_total = electron_density(mu_eV, Te_eV, Emin_eV=0.0,       Emax_eV=Emax_eV, N=
N, prefactor=prefactor)
    ne_free  = electron_density(mu_eV, Te_eV, Emin_eV=Vbarrier_eV, Emax_eV=Emax_eV, N
=N, prefactor=prefactor)
    return np.where(ne_total > 1e-10, ne_free / ne_total, 0.0)

def correct_ionization_degree(Z_qeos, Ti_eV, Te_eV, rho_gcc, Eb_eV, ne_qeos_m3, Vbarr
ier_eV, prefactor):
    """
    Core correction logic described in Section 2.2.3:
    - If Ti < Eb(rho): condensed phase => Z=0
    - Else: solve mu from ne_qeos and apply barrier-cutoff fraction to Z_qeos
    """
    if Ti_eV <= Eb_eV:
        return 0.0, 0  # Z, ok_flag

    mu, ok = solve_mu_from_ne(ne_qeos_m3, Te_eV, prefactor=prefactor)
    if ok == 0:
        return 0.0, 0

    frac = ionized_fraction(mu, Te_eV, Vbarrier_eV, prefactor=prefactor)
    return Z_qeos * frac, 1


# ============================================================
# C.4 Application to simulation fields (excerpt)
# ============================================================

# Given arrays from P4P output at one time step:
#   rho[g/cc], Ti[eV], Te[eV], Z_qeos[-], ne_qeos[m^-3], material_id
# And material constants:
#   rho0[g/cc], (E0,b) for Eb, and Vbarrier[eV]

# Example flow (vectorized conceptually):
# 1) compute Eb(rho)
# 2) for cells with Ti > Eb: solve mu and compute frac
# 3) Z_corrected = Z_qeos * frac ; else Z=0
```

Table A.1 Summary of numerical simulation results for hypervelocity impact cases oresented in Chapter 2. Corresponding results are shown in Figures 2-3 - 2-8 and 2-10 - 2-12.

Impact Velocity	23.0 km s^{-1}	27.3 km s^{-1}	33.6 km s^{-1}	44.0 km s^{-1}
Projectile radius [nm]	48.3	41.8	34.5	26.5
Projectile mass [kg]	3.69×10^{-18}	2.38×10^{-18}	1.34×10^{-18}	2.40×10^{-18}
Peak Ion yield [C]	3.48×10^{-14}	7.93×10^{-14}	1.49×10^{-13}	2.32×10^{-13}
Impact pressure [TPa]	2.32	3.05	4.34	6.82
Impact ion temperature [eV]	11.7	15.5	19.6	37.1
Impact electron temperature [eV]	8.9	11.3	15.5	22.9
Impact density [g cm^{-3}]	47.7	50.2	53.6	58.3
Jetting region ion temperature [eV]	1.09×10^{2}	1.24×10^{2}	1.44×10^{2}	1.89×10^{2}

Appendix B: Supplementary Numerical Results for Chapter 3

Table B.1 Summary of simulation results for hypervelocity impact cases analyzed in Chapter 3. Corresponding results are shown in Figures 3-5 – 3-10, and Table 3-3.

Impact Velocity [km s^{-1}]	5.0	7.5	10.0	14.0	19.5
Projectile radius [μm]	3.6	2.0	1.4	0.8	0.285
Projectile mass [kg]	1.78×10^{-13}	3.05×10^{-14}	1.05×10^{-14}	1.95×10^{-15}	8.82×10^{-17}
Peak ion yield [C]	7.24×10^{-15}	1.40×10^{-14}	2.76×10^{-14}	6.61×10^{-15}	2.82×10^{-14}
Total ion yield [C]	1.00×10^{-14}	1.91×10^{-14}	6.19×10^{-14}	1.73×10^{-14}	4.57×10^{-14}
Impact pressure [GPa]	64.7	136.1	136.8	237.0	379.8
Impact ion temperature [eV]	3.52	3.21	7.04	9.77	13.99
Impact electron temperature [eV]	1.05	0.92	1.58	2.88	5.27
Impact density [g cm^{-3}]	14.26	15.46	16.69	18.55	21.18

Table B.2 Summary of simulation results for laser irradiation cases analyzed in Chapter 3. Corresponding results are shown in Figures 3-11 – 3-13, and Table 3-3.

Laser intensity [MW cm^{-2}]	1150	975
Delay time [μs]	4.4	3.2
Peak ion yield [C]	1.10×10^{-5}	8.35×10^{-6}
Maximum pressure [GPa]	0.834	0.865
Maximum ion temperature [eV]	25.66	24.58
Maximum electron temperature [eV]	8.48	8.14
Maximum density [g cm^{-3}]	1.04	1.04
Maximum pressure (Delay time selected) [GPa]	0.171	0.146
Maximum ion temperature (Delay time selected) [eV]	25.66	24.58
Maximum electron temperature (Delay time selected) [eV]	8.48	8.14
Maximum density (Delay time selected) [g cm^{-3}]	3.12×10^{-3}	2.61×10^{-3}

Appendix C: Supplementary Numerical Data for Chapter 4

Table C.1 Values of the main parameters used in the P4P simulations in Chapter 4. Corresponding results are shown in Figures 5-8, 5-9, and Table 5-1.

Parameters	Values for impact	Values for laser ablation
Impact velocity	12.0 km s^{-1}	-
Projectile diameter	0.7 μm	-
Laser intensity	-	52.8 / 7.3 / 2.4 TW cm^{-2}
Pulse width	-	5 ns
Pulse peak time	-	7.5 ns
Δx	5.4 nm	0.25 μm
Δy	4.0 nm	0.5 μm
Cell per projectile radius	130 / 175	-
High resolution zone width	1.2 μm	100 μm
High resolution zone Height	5.4 μm	62.5 μm
High resolution target depth	1.1 μm	25 μm
Extension width	10 μm	100 μm
Extension top	7 μm	40 μm
Extension bottom	2.6 μm	30 μm
End time	175 ps	15 ns
Vacuum level	5 x 10^{-5} Pa	5 x 10^{-5} Pa

Table C.2 Summary of simulation results for hypervelocity impact cases analyzed in Chapter 4. Corresponding results are shown in Figures 4-7, 4-8 in Section 4.2.3.

Impact Velocity [km s^{-1}]	12.0
Projectile diameter [μm]	0.7
Projectile mass [kg]	1.31×10^{-15}
Peak ion yield [C]	2.69×10^{-14}
Total ion yield [C]	8.31×10^{-14}
Impact pressure [GPa]	174.26
Impact ion temperature [eV]	9.77
Impact electron temperature [eV]	2.97
Impact density [g cm^{-3}]	17.11

Table C.3 Summary of simulation results for laser irradiation cases analyzed in Chapter 4. Corresponding results are shown in Figures 4-7, 4-8 in Section 4.2.3.

Laser intensity [TW cm^{-2}]	52.8
Peak ion yield [C]	1.19×10^{-2}
Maximum pressure [GPa]	258.84
Maximum ion temperature [eV]	222.44
Maximum electron temperature [eV]	80.73
Maximum density [g cm^{-3}]	17.11

Appendix D: Core Python Implementation for the PI Modeling in Chapter 5

Appendix D.1 Model Assumptions and Scope

The photoionization (PI) model presented in this appendix serves as a first-order theoretical framework to interpret the experimental results discussed in Section 5.4.2. The model assumes a single-photon photoionization process, in which ionization occurs through direct absorption of a single VUV photon by a neutral molecule. Gas transport within the ionization region is treated under steady-state molecular flow conditions, and temporal variations in gas density are neglected. Ion-ion recombination, space-charge effects, and secondary ionization processes are not included in the present formulation. In addition, surface adsorption and delayed desorption of polar molecules are not explicitly modelled; the implications of these effects are discussed separately in Section 5.4.2.

Appendix D.2 Photon Flux and Ionization Rate Calculation

The photon flux within the PI region is calculated by propagating the spectral photon flux emitted from the VUV lamp through the chamber geometry while accounting for geometric dilution and wavelength-dependent attenuation. The local spectral photon flux is obtained by applying an inverse-square law to the source intensity and an exponential attenuation factor that includes both molecular absorption and Rayleigh scattering along the photon path. Using this local photon flux, the photoionization rate per molecule is computed by integrating the product of the photon flux and the photoionization cross section over wavelength. The resulting ionization rate is then multiplied by the local neutral gas number density to obtain the ion generation rate per unit volume, which is subsequently integrated over the effective PI volume to yield the total ion production rate. This algorithm is implemented in Python 3.12.3 (NumPy 1.26.4 and SciPy 1.11.4) on Ubuntu Linux (kernel 6.14.0-35-generic, x86_64).

```
import numpy as np

# ============================================================
# Appendix D: Photoionization (PI) Modeling Core Implementation
# Correspondence to Eqs. (5-1)-(5-4) in the main text
# ============================================================
```

```
# ----------------------------
# D.0 Inputs required (from your workspace)
# ----------------------------
# Arrays (1D, same length Npos = number of PI cells / evaluation points):
#   r_diff          : distance parameter used in your PI mesh (e.g., along z or beam
axis) [mm]
#   cos_theta       : cosine factor for molecular-flow jet (>=0 downstream) [-]
#   X_int, Y_int, Z_int : PI region cell coordinates [mm]
#   n_jet_cm3       : neutral gas number density at each PI cell [cm^-3]  (THIS is "
A" option)
#
# Lamp spectrum & cross sections:
#   energy_range    : photon energy grid [eV] (same indexing as extracted arrays)
#   extracted_vuv   : spectral radiant power [uW/nm] on energy_range grid
#   extracted_cross_section : list/array of cross sections in Mb (1 Mb = 1e-18 cm^2)
#       expected indices (as in your code):
#         [5]=o-xylene absorption? , [6]=acetone absorption?, [7]=ethanol absorptio
n?, [8]=background (N2) absorption
#         [0],[1],[2] etc can be PI cross sections if you stored them that way
#
# QMS / SIMION:
#   N_ion_QMS       : measured ion detection rates [ions/s] for [o-xylene, acetone,
ethanol]
#   means, means_a, means_e : SIMION transmission (%) for each radial zone (length 6)
#   IR_rad          : PI-region radius or ionization-region diameter reference [mm]
#   x_center        : x position of PI region axis [mm]
#
# Geometry reference (from your original implementation):
#   r_mid           : reference offset in the inverse-square factor [mm] (same lengt
h as r_diff or scalar)
#
# NOTE:
#   If r_mid is scalar -> OK. If array -> must match length of r_diff.

# ----------------------------
# D.1 Constants & parameters (Table D.1 recommended in the thesis)
# ----------------------------
h  = 6.62607015e-34      # J s
c  = 2.99792458e8        # m/s
kB = 1.380649e-23        # J/K
T  = 300.0               # K

S_TMP = 520.0            # L/s  (turbomolecular pump)
```

```
P_ch  = 5e-3             # Pa   (chamber pressure)
Q_out = S_TMP * P_ch * 1e-3  # Pa m^3/s  (since 1 L = 1e-3 m^3)

v_mol = 400.0            # m/s  (typical thermal speed scale)
P_bg  = 5e-3             # Pa   (background pressure offset)
dx_m  = 0.1e-3           # m    (path integration step; 0.1 mm)
r_min = 0.5e-3           # m    (minimum radius to avoid divergence near the orifice)

# Mesh cell volume used for integrating R_ion(r) (you used 0.03 cm per side)
dx_cm = dy_cm = dz_cm = 0.03
cell_volume_cm3 = dx_cm * dy_cm * dz_cm

Mb_to_cm2 = 1e-18        # 1 Mb = 1e-18 cm^2

# ----------------------------
# D.2 Helper functions (Eqs. 5-3, 5-4)
# ----------------------------
def energy_to_wavelength_nm(E_eV):
    return 1240.0 / E_eV

def vuv_spectral_photon_flux(extracted_vuv_uW_nm, energy_range_eV):
    """
    Convert spectral radiant power [uW/nm] to photon flux [photons/(cm^2 s nm)]
    (up to geometry factor applied later).
    """
    lam_m  = (1240.0 / energy_range_eV) * 1e-9
    lam_nm = (1240.0 / energy_range_eV)

    # [uW/nm] -> [W/nm]
    P_W_nm = extracted_vuv_uW_nm * 1e-6

    # photons/(m^2 s nm):  P / (hc/λ) = P * λ / (hc)
    I0_m2 = P_W_nm * lam_m / (h * c)
    I0_cm2 = I0_m2 * 1e-4
    return lam_nm, I0_cm2

def molecular_flow_pressure(Q_out, cosi, s_m, v_mol=400.0, P_bg=5e-3, r_min=0.5e-3):
    """
    P(s) = Q_out cos(theta) / (pi v s^2) + P_bg
    """
    s_eff = np.maximum(s_m, r_min)
    return (Q_out * cosi / (np.pi * v_mol * s_eff**2)) + P_bg

def optical_depth(sigma_tot_cm2, n_cm3, ds_cm):
```

```
    """
    tau(λ) = ∫ sigma_tot(λ) * n(s) ds  (discretized)
    """
    Ncol = np.sum(n_cm3) * ds_cm   # [cm^-2]
    return sigma_tot_cm2 * Ncol    # dimensionless

def local_photon_flux(I0_E, r_mm, tau):
    """
    Eq. (5-3): Phi(r, λ) = I0(λ) * (r_ref/r)^2 * exp(-tau)
    Your original: (549.4/(x+r_mid))^2 * I0 * exp(-tau) * 0.01
    Here we keep the same structure.
    """
    r_ref_mm = 549.4
    geom = (r_ref_mm / r_mm)**2
    return geom * I0_E * np.exp(-tau) * 0.01  # keep your mesh scaling

# ----------------------------
# D.3 Photoionization rate & ion generation (Eqs. 5-1, 5-2)
# ----------------------------
def gamma_photoionization(Phi_cm2_s_nm, sigma_PI_cm2, lam_nm):
    """
    Eq. (5-2): Gamma(r) = ∫ sigma_PI(λ) * Phi(r, λ) dλ
    """
    integrand = sigma_PI_cm2[None, :] * Phi_cm2_s_nm
    return np.trapz(integrand, lam_nm, axis=1)  # [1/s] for each cell

def ion_generation_rate_density(n_gas_cm3, Gamma_s):
    """
    R_ion(r) = n_gas(r) * Gamma(r)   [ions/(cm^3 s)]
    """
    return n_gas_cm3 * Gamma_s

# ----------------------------
# D.4 Mixture absorption cross section used in your attenuation model
# (this reproduces your sigma_E_0 construction)
# ----------------------------
def build_sigma_abs_mix_cm2(
    g, P_back, P_sat, extracted_cross_section, energy_range_eV,
    idx_xyl=5, idx_ace=6, idx_eth=7, idx_bg=8,
    bg_scale=0.2
):
    """
    Reproduce your "sigma_E_0" mixture for attenuation:
      - g=0: o-xylene line + background
```

```
      - g=1 or 2: acetone + ethanol + background
    Cross sections are assumed stored in Mb on the energy grid.
    Returns sigma_abs_mix [cm^2] on the wavelength/energy grid.
    """
    frac_x = P_sat[0] / P_back[g]
    frac_a = P_sat[1] / P_back[g]
    frac_e = P_sat[2] / P_back[g]

    if g == 0:
        sigma_mix = frac_x * extracted_cross_section[idx_xyl] + (1.0 - frac_x) * bg_s
cale * extracted_cross_section[idx_bg]
    else:
        sigma_mix = (
            frac_a * extracted_cross_section[idx_ace] +
            frac_e * extracted_cross_section[idx_eth] +
            (1.0 - (frac_a + frac_e)) * bg_scale * extracted_cross_section[idx_bg]
        )

    # Ensure same ordering as your later usage (you often used [::-1])
    # Here we assume you will pass arrays already in correct order.
    return sigma_mix * Mb_to_cm2

# ----------------------------
# D.5 SIMION transmission (Fig. 5-10) as zone-wise piecewise constant
# ----------------------------
edges = np.array([0.0, 0.1, 0.3, 0.5, 0.7, 0.9, 1.0])  # zone boundaries in r/R

def transmission_piecewise(r_norm, zone_values_percent):
    """
    Map r/R to transmission efficiency (0-1).
    zone_values_percent: length 6, in %
    """
    zone_values = np.array(zone_values_percent, dtype=float) * 1e-2
    # digitize gives bin index 1..len(edges); convert to 0..5
    idx = np.digitize(r_norm, edges, right=False) - 1
    idx = np.clip(idx, 0, len(zone_values) - 1)
    return zone_values[idx]

# ----------------------------
# D.6 Main evaluation for 3 species
# ----------------------------
def compute_PI_model_rates(
    # geometry
    r_diff_mm, r_mid, cos_theta, X_int, Y_int, Z_int, x_center, IR_rad,
```

```
    # gas density field (A option)
    n_jet_cm3,
    # lamp & cross sections
    extracted_vuv, energy_range_eV, extracted_cross_section,
    # PI cross sections for species (Mb)
    sigma_PI_species_Mb_list,
    # mixture params used in your code
    P_back, P_sat,
    # SIMION zones
    zone_vals_list_percent,  # [means, means_a, means_e] each length 6 (%)
    # include Rayleigh if desired (set to 0 array otherwise)
    sigma_ray_cm2=None
):
    # wavelength grid & lamp photon flux
    lam_nm, I0_E = vuv_spectral_photon_flux(extracted_vuv, energy_range_eV)

    Npos = len(r_diff_mm)
    r_diff_mm = np.asarray(r_diff_mm)
    cos_theta = np.asarray(cos_theta)

    # distance used in inverse-square factor: (x + r_mid) in your original
    if np.isscalar(r_mid):
        r_mm = r_diff_mm + float(r_mid)
    else:
        r_mm = r_diff_mm + np.asarray(r_mid)

    # normalized radius for SIMION mapping
    r_norm = np.sqrt((X_int - x_center)**2 + (Z_int)**2) / (IR_rad / 2.0)
    r_norm = np.clip(r_norm, 0.0, 1.0)

    total_ion_rate = []
    total_ion_rate_in = []
    detection_eff = []
    detection_eff_in = []

    # per species: 0=o-xylene, 1=acetone, 2=ethanol
    for g in (0, 1, 2):
        # --- attenuation cross section (absorption mixture) ---
        sigma_abs_mix_cm2 = build_sigma_abs_mix_cm2(
            g=g, P_back=P_back, P_sat=P_sat,
            extracted_cross_section=extracted_cross_section,
            energy_range_eV=energy_range_eV
        )
```

```
        # Rayleigh scattering term (optional; usually small)
        if sigma_ray_cm2 is None:
            sigma_tot_cm2 = sigma_abs_mix_cm2
        else:
            sigma_tot_cm2 = sigma_abs_mix_cm2 + sigma_ray_cm2

        # --- compute Phi(r, λ) for each PI cell ---
        Phi = np.zeros((Npos, len(lam_nm)))

        ds_cm = dx_m * 1e2
        for i in range(Npos):
            cosi = np.clip(cos_theta[i], 0.0, 1.0)

            # path length from orifice to that cell (your s_all definition)
            s_all_m = np.sqrt((X_int[i] - x_center)**2 + (Y_int[i])**2 + (Z_int[i] +
15.0)**2) * 1e-3
            s = np.arange(dx_m, s_all_m, dx_m)

            P_s = molecular_flow_pressure(Q_out, cosi, s, v_mol=v_mol, P_bg=P_bg, r_m
in=r_min)
            n_cm3_path = (P_s / (kB * T)) * 1e-6

            tau = optical_depth(sigma_tot_cm2, n_cm3_path, ds_cm)
            Phi[i, :] = local_photon_flux(I0_E, r_mm[i], tau)

        # --- PI cross section for that species ---
        sigma_PI_cm2 = (np.asarray(sigma_PI_species_Mb_list[g]) * Mb_to_cm2)

        # --- Eq. (5-2): Gamma(r) ---
        Gamma = gamma_photoionization(Phi, sigma_PI_cm2, lam_nm)  # [1/s]

        # --- Eq. (5-1): R_ion(r) using A option density field ---
        # NOTE: n_jet_cm3 is spatial distribution at each cell
        R_ion = ion_generation_rate_density(n_jet_cm3, Gamma)     # [ions/(cm^3 s)]

        # cell-wise ion generation [ions/s]
        dN_dt_cell = R_ion * cell_volume_cm3

        # total generation rate (no SIMION correction)
        R_total = np.sum(dN_dt_cell)
        total_ion_rate.append(R_total)

        # apply SIMION transmission (zone-wise)
        trans = transmission_piecewise(r_norm, zone_vals_list_percent[g])  # 0-1
```

```
        dN_dt_cell_in = dN_dt_cell * trans
        R_total_in = np.sum(dN_dt_cell_in)
        total_ion_rate_in.append(R_total_in)

    # detection efficiency vs experimental QMS rate (if available)
    # N_ion_QMS must be defined outside if you want these metrics
    return np.array(total_ion_rate), np.array(total_ion_rate_in)

# ============================================================
# D.7 Example execution (minimal)
# ============================================================

# Example PI cross sections for [o-xylene, acetone, ethanol] on your energy grid (M
b):
# sigma_PI_species_Mb_list = [sigma_PI_xyl_Mb, sigma_PI_ace_Mb, sigma_PI_eth_Mb]
#
# Example zone transmission lists (percent) from SIMION:
# zone_vals_list_percent = [means, means_a, means_e]

# Example mixture parameters (as used in your code):
# P_back = np.array([1e5, 2e5, 2e5])
# P_sat  = np.array([1e2, 700e2, 400e2])

# total_rate, total_rate_in = compute_PI_model_rates(
#     r_diff_mm=r_diff,
#     r_mid=r_mid,
#     cos_theta=cos_theta,
#     X_int=X_int, Y_int=Y_int, Z_int=Z_int,
#     x_center=x_center, IR_rad=IR_rad,
#     n_jet_cm3=n_jet_cm3,
#     extracted_vuv=extracted_vuv,
#     energy_range_eV=energy_range,
#     extracted_cross_section=extracted_cross_section,
#     sigma_PI_species_Mb_list=sigma_PI_species_Mb_list,
#     P_back=P_back, P_sat=P_sat,
#     zone_vals_list_percent=[means, means_a, means_e],
#     sigma_ray_cm2=None
# )
#
# print("samples = [o-Xylene, Acetone, Ethanol]")
# print("-------------------------------------")
# print(f"total_ion_rate      = {total_rate} [ions/s]")
# print(f"total_ion_rate_in   = {total_rate_in} [ions/s]  (SIMION-weighted)")
```

Table D.1 Parameters used in the photoionization (PI) modeling constructed in section 5.4.2. Corresponding results are shown in Figures 5-8, 5-9, and Table 5-1.

Parameter	Value
Turbomolecular pump speed, S_{TMP}	520 L s^{-1}
Chamber pressure, P_{ch}	5 x 10^{-3} Pa
Spatial step along photon path, dx	0.1 mm
PI mesh cell volume	0.03 cm^3
Molecular flow velocity, v_{mol}	400 m s^{-1}