Hi, I am working on this project that is about the paper AC/DC efficient calculation up to hundreds of MHz. Here is what im doing: 1. build board in PowerSI tool, extract its stackup.csv, .spd file for board configuration and touchstone .snp files after the PDN simulation for the board. 2. input these 3 files, plus 1 layer_type.csv file in vs code input_AH.py. 3. parse this input data to consctruct a board inside python 4. use this data to calculate res matrix 5. using res matrix, calculate z efficiently 6. use simulated .snp files to compare PowerSI and python calculates for IC port Z11. here are my code files. can you make sense of what is going on? use the paper in the context to to make sense of the CIM/BEM/Node Volate method/Merging. here is also part of input_AH.py # Board 4.1 layer_type_path = r"input files\board 4.1\b4_1_layer_type.csv" stackup_path = r"input files\board 4.1\b4_1_stackup.csv" input_path = r"input files\board 4.1\b4_1.spd" touchstone_path = r"input files\board 4.1\b4_1.S3P" and here are what input files look like. can you analyze these code deeply and tell me what each module of the project does.

Awesome — I can see the whole pipeline and it maps very neatly to the "boundary-integration (BEM) + node-voltage + merging" strategy from the paper. Here's a clean, module-by-module walkthrough, with how each piece lines up with CIM/BEM/Node-Voltage/Merging.

What each module does

1) `main.py` — Orchestrator

- Wires together everything: imports your PDN model (`PDN`), the DC plane solver (`main_res`), the SPD/stackup parsers, and scikit-rf for Touchstone comparison. main
- `Board` is a tiny container used while parsing, then you actually operate on the `PDN` object. main
- `gen_brd_data(...)` is the main builder:
    a. Parses the SPD (board shapes, vias, start/stop layers, etc.) with `parse_spd`, and handles the presence/absence of buried vias. main
    b. Stores the boundary shapes (polygons/boxes collapsed if identical). main
    c. Assigns/normalizes via arrays, start/stop layer indices, and de-dupes via locations so you don't get accidental overlaps. main main main

- Later, you call the fast Z builder on the `PDN` (`brd.calc_z_fast(...)`) and then compare to Touchstone (interpolated to your sim frequencies). The compare path includes helpers to short one port by removing a row/column in Z^{-1} (standard port-short trick). main main

- Touchstone plotting flows through `compare_shorted_z` (magnitude in dBΩ over frequency) and an `ic_port` auto-detector that reads the comment lines and defaults sensibly if missing. main main

2) `pdn_io/spd_parser.py` — PowerSI .spd parser

- Extracts board **boundary shapes** (polygons or boxes), normalizes them, and collapses duplicates (so multiple identical shape definitions become a single shape). These returned shapes feed your later per-cavity area logic. spd_parser spd_parser spd_parser

- Builds **node info** with canonical names and attributes (PWR/GND, x/y in mm, Signal## layer) — this is the basis for assembling via arrays and IC/decap labeling. spd_parser spd_parser

- The `parse_spd(...)` top-level returns everything your builder expects (shapes, IC/decap vias & types, start/stop layers, etc.), which `gen_brd_data(...)` unpacks and installs into the `PDN`. main

3) `pdn_io/stackup_parser.py` — Stackup & layer typing

- Normalizes the **layer-type CSV** (e.g., allowing “PWR”/“GND” strings), extracts `Signal##` indices, and returns a tidy DataFrame. stackup_parser stackup_parser

- Builds a **0/1 mask** over signal layers indicating PWR/GND assignment, which is used when you later anchor/merge nodes per layer. stackup_parser

- Convenience API also loads **cavity thickness** `die_t`, **per-cavity εr list**, and **copper thickness** `d_r` from the stackup CSV (exactly what `main.py` needs). stackup_parser

4) `RES1_AH.py` — DC plane resistance via boundary integration + node-voltage

- Utility `seg_bd_node(...)` discretizes the plane boundary into segments; `segment_port(...)` segments a circular “port” boundary — these create the BEM panelization you need. RES1_AH RES1_AH

- `planesresistance(...)` is a classic **BEM for DC sheet resistance**: it assembles the U and H matrices from segment interactions, builds the interaction operator $U^{-1}H$, and (after segment-merging) inverts to get a

nodal resistance matrix; then converts to an **rb** matrix with entries that act like effective resistances between vias on that layer. This is your “CIM/BEM” DC plane piece. RES1_AH RES1_AH RES1_AH RES1_AH

- In the **node-voltage assembly**, you build incidence A (node-branch connectivity), a diagonal branch impedance **zb**, then compute **Yn = A·inv(zb)·Aᵀ**, remove the reference node, and invert to get **Z** (classic nodal formulation). The code currently returns `zn2[0,0]` as a scalar DC result. RES1_AH RES1_AH
- Horizontal (in-plane) branches on each layer receive entries from the BEM-computed **rb**, so the plane’s sheet resistance is folded into `zb` for those branches. RES1_AH

Note: in your current flow, `main.py` calls `main_res(...)` and passes its output to `calc_z_fast(res_matrix=...)`, but `calc_z_fast` doesn’t actually consume `res_matrix` anywhere inside — see the signature vs. the Yb construction below. (So DC rb isn’t yet coupled into the AC sweep.) code_pdn_AH code_pdn_AH

5) `code_pdn_AH.py` — The PDN object and fast AC Z(ω) builder

- `class PDN` is your working model: it stores shapes (`outer/inner_bd_node`, `sxy`), areas, vias (IC/decap/buried), stackup, εr list, etc. code_pdn_AH code_pdn_AH
- While building, you compute **per-cavity area** (single shape → same area per cavity; multiple shapes → overlap of consecutive shapes) and **C_pul = εr·ε0·Area/height** (used later as inter-layer capacitors). code_pdn_AH code_pdn_AH
- `calc_lpul_bem(...)` computes the **partial inductance matrix** with a BEM-style formulation (D, G, Gh, E = I−D, etc.). This is the **AC/BEM** piece that gives you L between vias around the plane boundary. code_pdn_AH code_pdn_AH
- `org_merge_pdn(...)` (and the earlier logic that returns `branch_merge_list`) builds the **branch list**, identifies “anchored” layers via ports, and collects **branch-merge** groups to compress the system (merging branches across cavities or same-via branches). That’s your **Merging** step to keep the problem small. code_pdn_AH code_pdn_AH
- `merge_L_big(...)` then merges rows/cols of the L matrix according to that plan, producing a reduced **L_new_inv** and a **new↔old** node map for later reduction. code_pdn_AH code_pdn_AH code_pdn_AH
- In `PDN.calc_z_fast(...)` you assemble the **branch admittance matrix Yb(ω)** for the AC sweep:

- **Inductive branches** use $Y_L = \frac{1}{j\omega} L^{-1}$ from `L_new_inv`. code_pdn_AH
- **Cavity capacitors** are added as **extra branches** between the (possibly anchored) nodes of adjacent planes with $Y_C = j\omega C$ using εr·ε0·Area/thickness. code_pdn_AH code_pdn_AH
- Then you build the **reduced incidence A** and compute **Yn(ω) = A·Yb·Aᵀ** (vectorized over frequency via `einsum`) → invert (with ref removal) to get port Z(ω). code_pdn_AH code_pdn_AH
- There's rich debug output for anchored layers, added capacitor branches, node mapping, and connectivity checks if `verbose=True`. code_pdn_AH code_pdn_AH code_pdn_AH code_pdn_AH

How this matches the paper's method (CIM/BEM/Node-Voltage/Merging)

- **CIM/BEM for DC planes**
  Your `planesresistance(...)` is a boundary-integral BEM that computes the plane's **conductive interaction** via U/H kernels, then reduces to an effective via-to-via resistance matrix **rb** on each layer. Those **rb** entries populate the diagonal of `zb` for horizontal branches in the DC node-voltage solve (A·inv(zb)·Aᵀ), exactly the "CIM/BEM + nodal reduction" idea the paper leans on for efficient DC extraction. RES1_AH RES1_AH RES1_AH RES1_AH
- **BEM partial-inductance for AC**
  `calc_lpul_bem(...)` builds the mutual partial-inductance matrix among vias using the boundary segments (D, Gh, E=I−D, etc.). That provides the **L** used to form $Y_L(\omega) = \frac{1}{j\omega} L^{-1}$. code_pdn_AH code_pdn_AH
- **Node-voltage formulation**
  Everywhere, you stick to $Y_n = A Y_b A^T$ and ref-node elimination; DC does it with `zb` (resistive branches + plane rb), and AC does it with `Yb(ω)` that combines inductive and capacitive branches. RES1_AH RES1_AH code_pdn_AH
- **Merging to keep it small**
  `org_merge_pdn(...)` marks which branches to merge (e.g., coincident vias across cavities or layer anchored nodes), and `merge_L_big(...)` actually compresses the L matrix + branch list, drastically shrinking

the solve. That's the paper's "connectivity-aware merging" to scale to large via counts. code_pdn_AH code_pdn_AH

End-to-end flow (your 6 steps mapped)

1. **Build in PowerSI, export CSV/SPD/S-params** → the SPD gives shapes & nodes; stackup CSV gives dielectrics and Cu thickness; Touchstone (`.S3P`) gives measured/simulated ports. main
2. **Point `input_AH.py` to files** → `main.py` imports those paths and passes them into `gen_brd_data(...)`. main
3. **Parse in Python** → SPD parser constructs shapes + via arrays; stackup parser normalizes PWR/GND and returns `die_t`, `er_list`, `d_r`, `mask`. main stackup_parser
4. **Compute "res matrix" (DC)** → `main_res(...)` runs BEM on planes to get `rb`, builds A/zb, and returns the DC nodal impedance (currently the code returns a scalar `zn2[0,0]`). RES1_AH RES1_AH RES1_AH
5. **Efficient Z(ω)** → `PDN.calc_z_fast(...)` builds L via BEM, merges, adds inter-layer **capacitor branches** from εr·ε0·Area/d, forms **Yb(ω)**, then $Y_n(\omega) = A \cdot Y_b \cdot A^T$ and inverts to get Z(ω). (Note: `res_matrix` isn't used yet.) code_pdn_AH code_pdn_AH code_pdn_AH
6. **Compare to Touchstone** → scikit-rf read + interpolation, short IC port in both Z's, plot |Z| dBΩ for apples-to-apples verification. main main

---

Two small observations you'll likely want to address

- **Hook DC plane loss into AC**: you already compute the DC plane interaction (`rb`), but `calc_z_fast(...)` doesn't include resistive terms in **Yb(ω)**. If you want loss up to hundreds of MHz, add a resistive branch component (e.g., series R per inductive branch or shunt sheet loss per cavity) rather than leaving Y purely reactive. code_pdn_AH
- **`res_matrix` is computed but unused**: `main.py` passes `res_matrix=res_matrix`, but `calc_z_fast` ignores it. If you intended to blend DC/low-freq sheet resistance into the sweep, that's where to consume it. code_pdn_AH

If you want, I can sketch the minimal edits to (a) thread `rb`/conductor loss into `Yb(w)` and (b) tidy the port-short comparison to directly overlay Z_{11} from Python vs. Touchstone.