ChatGPT

PV Curves and Voltage Stability

PV (Power-Voltage) curves plot the bus voltage in an electric power system against the delivered active power (load) at that bus, typically under constant power factor. The resulting graph resembles a "nose curve," where voltage stays high at low loading and then drops as loading increases [1]. In this plot the **nose (tip)** of the curve marks the maximum loadability: it corresponds to the highest power the system can deliver to that bus with acceptable voltages [2] [3]. In other words, the tip defines the upper limit of stable operation. As load increases from zero, the operating point moves along the upper (right-side) branch of the PV curve toward the nose, with voltage gradually falling. If loading continues beyond the nose, both voltage and power collapse sharply (the point of *voltage collapse*) and the solution becomes infeasible [4]. Engineers must maintain a safety margin on the left of the nose to avoid instability – once load exceeds the nose-point, voltage collapse is likely [4] [3].

Figure 1: A typical P–V curve (voltage vs. active power) at a bus. The "nose point" (peak) is the maximum deliverable power. Operating along the upper-right branch up to the nose is stable; beyond the nose (left branch) the system enters voltage collapse (unstable) [4] [3].

Key points (beginner): A PV curve is generated by incrementally increasing the real power load at a bus and solving the power flow each time. At low load the voltage is near nominal, but as power increases the voltage drops. The nose peak indicates the *maximum load* before voltage collapse. In practice, operators keep the system well left of the nose to maintain stability [2] [5].

Advanced insights: The shape of the PV curve is governed by the network's impedance and reactive power limits. If reactive support is insufficient, the collapse can occur *before* the theoretical nose point [4]. The lower-left part of the curve (past collapse) is not physically attainable in steady-state operation, as it would require unrealistically low voltages and high currents. Mathematically, PV curves are related to voltage stability margins and are used in *static voltage stability* analysis – for example, tracing the curve is a common method to "assess the distance to voltage instability" [5]. In fact, textbooks note that voltage stability assessment often "determines and characterizes the network's power transfer capability" via PV curves, with the nose point marking the limiting condition [3] [2].

Constructing a PV Curve: In practice, one scales system load and runs a power flow repeatedly. For instance, a standard procedure is: (1) solve the base-case load flow; (2) scale all loads (and possibly generation) by a factor; (3) run power flow and record the total real power and monitored bus voltage; (4) repeat until the solver fails to converge – at that point the load factor has reached the nose point [6] [5]. The recorded P–V points are then plotted. Figure 1 illustrates such a curve: the flat (upper-left) part is safe (stable), the peak is the nose (limit), and the downward turn is the collapse region [4] [3].

PV Curve Tutorial with Pandapower

We can compute and plot PV curves programmatically using **pandapower**, a Python library for power flow analysis. Pandapower includes many standard test cases (e.g. IEEE 39-bus, 118-bus) via the

`pandapower.networks` module [7] [8]. For example, `pn.case39()` and `pn.case118()` load the IEEE 39-bus and 118-bus systems respectively [7] [8].

Steps to calculate a PV curve:
- **Select a network and bus.** Load an IEEE test case (or user-defined grid) into a `pandapower` network. Choose a load bus index for analysis.
- **Record base load.** Save the original active (P) and reactive (Q) loads at all buses.
- **Incrementally increase loading.** In a loop, scale up the total system load by a factor (e.g. from 100% up to failure) in steps. At each step:
- Update `net.load['p_mw']` and `net.load['q_mvar']` by the scale factor of the base load.
- Run the power flow with `pp.runpp(net)` [9].
- Record the monitored bus voltage `net.res_bus.vm_pu[bus_index]` and the current total real load power.
- **Stop at collapse.** Continue until `runpp` fails to converge, which indicates the nose point was exceeded [6].
- **Plot Voltage vs. Power.** Using Matplotlib, plot the recorded voltages (y-axis) against total power (x-axis) to obtain the PV curve.

For example, pseudocode in Python:

```
import pandapower as pp
import pandapower.networks as pn
import numpy as np
import matplotlib.pyplot as plt

# Load an IEEE test grid
net = pn.case39()
bus_idx = 10              # bus number to monitor

# Save base loads
base_p = net.load['p_mw'].values.copy()
base_q = net.load['q_mvar'].values.copy()

P_list, V_list = [], []
for scale in np.linspace(1.0, 2.0, 41):  # scale load from 100% to 200%
    net.load['p_mw'] = base_p * scale
    net.load['q_mvar'] = base_q * scale
    try:
        pp.runpp(net)
    except pp.LoadflowNotConverged:
        break   # collapse (nose) reached
    totalP = net.res_load['p_mw'].sum()
    V_bus = net.res_bus.at[bus_idx, 'vm_pu']
    P_list.append(totalP)
    V_list.append(V_bus)
```

```
plt.plot(P_list, V_list, marker='o')
plt.xlabel("Total Load P [MW]")
plt.ylabel(f"Voltage at Bus {bus_idx} [p.u.]")
plt.title("PV Curve at Bus {}".format(bus_idx))
plt.grid(True)
plt.show()
```

This code performs a series of power flows with increasing load and plots the resulting voltage. Note that *near the nose point*, smaller increments (or adaptive stepping) are often needed to accurately locate the peak; in practice one observes that if `runpp` fails to converge, the last successful point was just before the collapse [6].

Interpretation of results: The generated PV curve shows how much the voltage falls as load grows. From this curve we can identify: (a) the **nose point** (maximum delivered P and its voltage), (b) the **voltage margin** (difference between base loading and the nose loading), and (c) the shape of decline (which indicates reactive power sensitivity). For example, after plotting, one might compute the nose-point values and margin and then note: "Bus 10 collapses at 1.42 pu loading (total P=350 MW) with V≈0.93 pu; base case was 250 MW, so the margin is 100 MW." These insights complement the visual plot.

Supporting IEEE test systems: Pandapower's library makes it easy to try standard cases. For instance, loading `pn.case118()` gives the IEEE 118-bus network [8]. Open datasets like the Illinois or Washington state repositories provide the raw data that pandapower uses for these cases (as noted in the documentation) [10] [11]. Users can also import custom grid models (e.g., from Matpower or CSV data) to perform PV analysis on any system.

LLM-Powered Conversational PV Curve System

We envision an interactive **conversational tool** (using a Deepseek LLM) that lets a user request PV curve analyses in natural language. For example, a user might type "Plot the PV curve at bus 10 of the IEEE 39-bus system with 5% load steps," and the system would produce the corresponding plot and commentary. The system architecture must support **retrieval-augmented generation (RAG)** and **multi-step conversational planning (MCP)** so that it can interpret flexible queries, fetch relevant domain knowledge, perform computations (via pandapower), and guide the conversation.

Architecture (LLM + RAG + Tool Integration)

The system consists of the following components:

- **User Interface (Chat):** Receives free-text queries describing the desired analysis.
- **LLM Agent (Deepseek):** The conversational AI core that parses queries, plans actions, asks clarifying questions, and formulates answers. It is orchestrated using a multi-turn agent framework (e.g. LangChain or Guidance) that supports tool use.
- **Retrieval Module:** A vector-based knowledge store (e.g. a Haystack or LlamaIndex index) containing relevant documents and data (e.g. PV curve theory, grid descriptions, pandapower docs). On each

query, this RAG component retrieves pertinent facts to augment the LLM's context. (RAG ensures the model answers with up-to-date, factual information [12].)

- **Parameter Extraction:** A step where the LLM or a subprocess identifies key parameters (bus index, step size, network selection, base loading) from the user query (or asks follow-up questions if missing). The frameworks can use prompting or tool-calling to parse entities.
- **Computation Tool (Pandapower API):** A backend service or function that, given a network model and load-scaling instructions, runs the power flow loop and returns the PV data (voltage vs. load). This is exposed as a "tool" that the LLM can invoke (e.g. via LangChain's tool API or an HTTP function).
- **Plot Generation:** The numerical PV data is plotted (matplotlib or Plotly) and the image is sent back to the user. The LLM references this image when explaining results.
- **Memory/Context:** A conversation memory (e.g. Haystack chat memory or LangChain memory) that stores previous Q&A, so follow-up queries (like "How about bus 20?") are handled coherently.

A high-level flow is:

1. **Receive Query:** User asks a question, e.g. "Show PV curve for bus 5 in 118-bus system."
2. **Information Retrieval (RAG):** The system retrieves documents about PV curves and the 118-bus system, to ground the LLM's understanding (for example, key facts about IEEE 118 and voltage stability [12]).
3. **Parse & Plan:** The LLM (using e.g. a ReAct or Pre-Act style prompt) extracts parameters. If any are missing (e.g. user did not specify step size or base load), the agent generates a clarifying question ("What load increment do you want?"). This planning of conversation steps is guided by multi-step reasoning techniques [13].
4. **Execute Computation:** Once all inputs are gathered, the agent invokes the pandapower PV tool with those inputs. This may involve calling a python function (or service) that runs the loop as in the tutorial above.
5. **Formulate Answer:** The LLM receives the PV data and plot. It generates a response that embeds the plot and explains the results (for example, "The PV curve for bus 5 shows a nose at 290 MW with $V \approx 0.95$ pu, so the margin is 30 MW above base" plus interpretation). It cites any theoretical context retrieved earlier.
6. **Ask Follow-ups:** Finally, the system might proactively offer next options ("Would you like to analyze another bus or plot a Q-V curve?"), leveraging conversation memory.

This agentic architecture (combining LLM reasoning, tools, and retrieval) is an instance of *agentic RAG*: the AI not only retrieves context but also takes actions (running simulations) based on reasoning [14] [12]. In other words, the LLM uses contextual knowledge to *decide what to do*, and has *tools* (pandapower) to execute those decisions [15].

Example pseudocode (workflow):

```
def handle_user_query(query):
    # Step 1: Retrieve relevant context (theory, docs) via RAG
    context_snippets = retriever.get_relevant_docs(query)

    # Step 2: LLM extracts or asks for parameters using context
    params = llm_agent.extract_params(query, context=context_snippets)
```

```
    while not params.complete():
        question = llm_agent.ask_for_missing(params)
        user_reply = get_user_response(question)
        params.update(llm_agent.parse_params(user_reply))

    # Step 3: Run Pandapower PV analysis as a tool
    net = load_network(params.grid_model)    # e.g. IEEE 39 or custom model
    pv_result = run_pandapower_pv(net, params.bus_index, params.step_size)

    # Step 4: LLM generates final answer including plot and insights
    answer = llm_agent.generate_answer(params, pv_result, context_snippets)
    display_plot(pv_result.plot)
    return answer
```

In practice, frameworks like **LangChain** or **Guidance** can implement this flow. For example, LangChain can define a custom "run_pv_curve" tool (wrapping pandapower) and an agent that plans to call it. Haystack v2 likewise offers **ConversationalRetrievalQA** pipelines with memory for chatbots. LlamaIndex (GPT Index) could index large manuals (e.g. power system textbooks) to augment RAG. We would also include prompt engineering: the LLM must be instructed (via prompt or fine-tuning) to carry out the multi-step plan, ask clarifications, and interpret results in technical terms. Recent research (e.g. *Pre-Act*) shows that explicitly generating intermediate planning steps ("multi-step execution plan") leads to better performance on complex tasks [13]. We would leverage similar ideas: the agent generates a step-by-step plan (question user, retrieve, compute, answer) before executing each step.

Frameworks & Tools: Open-source libraries facilitate each component. For RAG, we can use **Haystack** or **LlamaIndex** with a vector DB (e.g. Chroma) to store documents (like pandapower docs, IEEE system data, stability notes). For agentic prompting, **LangChain Agents** or **Guidance** allow the LLM to use external tools with memory. **Pandapower** is the backend solver. For plotting, Matplotlib/Plotly is used; the LLM can refer to the saved image file. For conversations, a chat frontend (like Streamlit or a chatbot UI) handles turn-taking.

Conversational Workflow and Follow-ups

Parameter Extraction: When the user query arrives, the LLM first identifies needed inputs: e.g. network model (IEEE 39-bus vs custom), bus index, load step size. If any detail is missing or ambiguous, the system asks a clarifying question. This can be done by prompting the LLM with a schema or using a small specialized parser. For example, if the user says "PV curve at bus 10", the agent might check if "step size" or "increase range" was given, and if not, ask "How much to increase the load each step?"

Validation & Memory: The system stores confirmed parameters and previous answers in memory. If the user later asks "Now do bus 15," the agent keeps prior context (e.g. that the grid is IEEE 39-bus and step=5%) and only changes the bus parameter. This conversational memory ensures continuity.

Backend Computation: Once all inputs are available, the LLM triggers the pandapower computation. Internally, this may be an API call (for security and modularity) or a direct function call. The LLM can

supervise or adjust the steps; for instance, if the first run fails to converge, the agent might retry with finer increments (a simple form of self-repair).

Output Generation: After computation, the LLM crafts an answer. It will embed the resulting PV plot (as an image) and provide textual analysis: e.g., "The PV curve at bus 10 peaks at 1.4 pu total power (nose point) and collapses beyond that [4]. The operating margin from the original load (~1.0 pu) to collapse is about 0.4 pu, indicating moderate stability. Increasing reactive compensation would shift the nose right." It should explain terms in plain language for beginners, but can also give quantitative metrics for engineers. All claims (e.g. "nose at 1.4 pu") come from the actual data, and any theoretical statements ("nose is limit") can be supported by context (we retrieved that info via RAG [12] [13]).

Follow-up Options: Finally, the assistant would prompt the user with next steps: "Would you like to analyze another bus (0-38), a different grid (e.g. IEEE 118-bus), or adjust load growth rate?" This uses a conversation-level plan: after delivering the answer, propose related actions. For example, if the user says "Also show Q-V curve," the agent would know to fix P and vary Q, running a reactive power sweep. In each case, the RAG+MCP flow repeats.

This **multi-step conversational planning** (asking for inputs, computing, then offering choices) leverages the LLM's reasoning abilities. Recent work shows that when an agent outlines and follows a plan, complex tasks become manageable [13]. Here, the LLM effectively acts as an orchestrator: it reasons about what the user wants, consults its augmented knowledge, decides to call the PV-curve "tool," and refines the answer step by step [14] [15].

Supporting Datasets and Resources

- **IEEE Test Cases:** Standard grid data (39-bus, 118-bus, etc.) are available in pandapower and other libraries (often converted from MATPOWER). References like the Illinois Power System Test Case Archive list these networks [10] [11].
- **Data Repositories:** Synthetic or regional grid datasets (e.g. Illinois 200-bus) can be included by importing files into pandapower.
- **Libraries:** Pandapower's GitHub and docs (e.g. the *Power System Test Cases* page) provide code snippets and case definitions [7] [8]. LangChain and Haystack have extensive tutorials (e.g. LangChain's RAG tutorial).
- **Educational Material:** Online textbooks and notes on voltage stability (like *Power System Stability* by Kundur) or lecture slides can be ingested into the RAG index for explanation of terms.

By combining domain knowledge (via RAG) with LLM-driven orchestration (MCP), this system can serve both novices and experts. Beginners get clear explanations and visuals ("This curve shows how voltage drops as load goes up; the highest safe load is at the nose [3]"), while experienced engineers can drill into quantitative results and subtleties ("Reactive limits caused collapse before full thermal limit – see Q-V curve next"). The use of open-source tools and datasets ensures transparency and reproducibility.

References: PV curve theory and voltage stability are well documented in power systems literature [1] [3]. Pandapower's documentation and examples (e.g. networks list) show how to load test grids [7] [8]. Modern LLM/agent frameworks (e.g. LangChain, Haystack) are designed for exactly this kind of RAG + tool usage. For example, IBM's RAG pattern overview explains augmenting LLMs with external knowledge [12], and recent research (Pre-Act) highlights multi-step planning in conversational agents [13]. Finally, the

concept of *agentic RAG* – where the AI agent reasons and uses tools – has been discussed in recent literature [14] [15], matching our system's design.

---

[1] [2] [4] Power-voltage curve - Wikipedia
https://en.wikipedia.org/wiki/Power-voltage_curve

[3] [6] Lab 3 - Voltage Stability Accessment
http://www.ece.ualberta.ca/~terheide/ECE433-lab/lab3.html

[5] ECE 310
https://www.powerworld.com/files/D14VoltageStability.pdf

[7] [8] [10] [11] Power System Test Cases — pandapower 2.0.0 documentation
https://pandapower.readthedocs.io/en/v2.0.0/networks/power_system_test_cases.html

[9] Power Flow — pandapower 2.2.2 documentation
https://pandapower.readthedocs.io/en/v2.2.2/powerflow/ac.html

[12] Retrieval Augmented Generation
https://www.ibm.com/architectures/patterns/genai-rag

[13] Pre-Act: Multi-Step Planning and Reasoning Improves Acting in LLM Agents
https://arxiv.org/pdf/2505.09970

[14] [15] Agentic RAG: Architecture, Use Cases, and Limitations
https://www.vellum.ai/blog/agentic-rag