What is PowerModels.jl?

- An open-source package for the **Julia** programming language.
- Built on the **JuMP** optimization modeling layer.
- Designed for **Steady-State Power Network Optimization**.
- **Core Purpose:** A common platform for the *computational evaluation* of new and emerging power network formulations and algorithms.

It's not just a solver; it's a **research laboratory**.

The Core Design Principle:

1. **Problem Specifications (The "What")**
    - *What problem* are you solving?
    - *Examples:* Power Flow (PF), Optimal Power Flow (OPF), Transmission Network Expansion Planning (TNEP).
2. **Formulation Details (The "How")**
    - *What mathematical model* are you using?
    - *Examples:* `ACPPowerModel` (full non-linear), `DCPPowerModel` (linear approximation), `SOCWRPowerModel` (convex relaxation).

2. Getting Started

"Hello, World!" Script (Part 1)

This improved script assumes packages are already installed.

```julia
# Step 1: Import the necessary libraries
using PowerModels
using Ipopt
using JuMP # Often needed for solver settings

# Step 2: Clear the terminal for clean output
print("\033c")
println("Starting PowerModels.jl AC-OPF Example...")

# Step 3: Define the solver
# Instantiate the optimizer. This is more flexible.
# (e.g., to suppress logs:
#  optimizer = optimizer_with_attributes(Ipopt.Optimizer, "print_level" => 0))
optimizer = Ipopt.Optimizer
```

Enhanced "Hello, World!" Script (Part 2)

```julia
# Step 4: Find the path to a test data file
pm_path = dirname(pathof(PowerModels))
case_file = joinpath(pm_path, "..", "test", "data", "matpower", "case3.m")
println("Running optimization on test file: ", case_file)

# Step 5: Run the optimization
# This is a "high-level" API call.
result = solve_ac_opf(case_file, optimizer)

# Step 6: Display the results
println("\n--- Optimization Results ---")

# 'print_summary' gives a human-readable table
println("Solution Summary:")
print_summary(result["solution"])

# Access specific data points from the result dictionary
println("\nSolver solve_time: ", result["solve_time"], " seconds")
println("Final objective value: ", result["objective"])
```

3. Handling Network Data

Supported Formats

The library provides robust support for common power system formats.

- **Matpower (`.m`)**:
  - Extensive support.
  - The de-facto standard in academic research.
- **PSS/E (`.raw`)**:
  - Supports PTI network files (v33 specification).
  - Widely used in industry.
  - Use `import_all=true` flag to load extra PSS/E data fields.

Code: Inspecting Network Data

Before you optimize, inspect your data. `PowerModels.jl` has built-in helper functions for this.

```julia
using PowerModels

pm_path = dirname(pathof(PowerModels))
case_file = joinpath(pm_path, "..", "test", "data", "matpower", "case14.m")

# 1. Parse the file into a dictionary
network_data = PowerModels.parse_file(case_file)

# 2. Get a high-level summary (counts of components)
println("--- Network Summary (case14) ---")
print_summary(network_data)

# 3. Use component_table for a clean, targeted view
println("\n--- Bus Voltage Limits ---")
print(PowerModels.component_table(network_data, "bus", ["vmin", "vmax"]))
```

4. Problem Specifications

(The "What")

what problem you are asking `PowerModels.jl` to build.

- **Power Flow (PF)**
  - **Function:** `build_pf`
  - **Purpose:** Solves the power flow equations (a root-finding problem, not an optimization). *Asks: "What are the voltages and flows?"*
- **Optimal Power Flow (OPF)**
  - **Function:** `build_opf`
  - **Purpose:** Minimizes a cost (e.g., generation) subject to network physics and engineering limits. *Asks: "What is the cheapest way to run the grid?"*
- **Optimal Power Balance (OPB)**
  - **Function:** `build_opb`
  - **Purpose:** A "copper-plate" model. Ignores all network constraints and just balances generation and load.

Advanced Problem Specifications

These are primarily used for research and planning.

- **Optimal Transmission Switching (OTS)**
  - **Function:** `build_ots`
  - **Purpose:** An OPF where the "on/off" status of lines is a decision variable. *Asks: "Can we run the grid more cheaply by opening a line?"*
- **Transmission Network Expansion Planning (TNEP)**
  - **Function:** `build_tnep`
  - **Purpose:** A planning problem to decide *where to build new lines* from a set of candidates. *Asks: "Where should we build new lines to minimize total cost?"*

Problem Specification	Build Function	Description & Purpose
Power Flow (PF)	`build_pf`	Solves the non-linear power flow equations (root-finding).
Optimal Power Flow (OPF)	`build_opf`	Minimizes cost subject to network physics and limits.
Optimal Power Balance (OPB)	`build_opb`	"Copper-plate" (network-agnostic) economic dispatch.
Optimal Transmission Switching (OTS)	`build_ots`	OPF + line "on/off" status as variables.
Transmission Network Expansion Planning (TNEP)	`build_tnep`	Decides where to build new transmission lines.

5. Mathematical Formulations

(The "How")

Exact Model: `ACPPowerModel`

- **Description:** The full, non-linear, non-convex AC power flow model. Uses polar coordinates (voltage magnitude `vm` and angle `va`).
- **Pros:**
  - It is the **"ground truth"**.
  - Models the underlying AC physics exactly. Highest accuracy.
- **Cons:**
  - **Non-Convex.**
  - Solvers (like `Ipopt`) only guarantee finding a *local optimum*.
  - May fail to converge on complex problems.

Linear Approximation: `DCPPowerModel`

- **Description:** The standard "DC" approximation. A linear model based on several simplifying assumptions (e.g., flat 1.0 p.u. voltages, lossless lines).
- **Pros:**
    - **Convex** (it's a Linear Program).
    - Extremely fast and robust.
    - Guaranteed to find the global optimum *of the approximated problem*.
- **Cons:**
    - It's an **approximation**.
    - Ignores reactive power, voltage limits, and losses.
    - Only suitable for high-level indicative studies.

Convex Relaxation: `SOCWRPowerModel`

- **Description:** A modern, powerful Second-Order Cone (SOC) *relaxation* of the full AC problem.
- **Pros:**
    i. **Convex.** Can be solved to a *global optimum* reliably.
    ii. **More Accurate** than the `DCPPowerModel`.
    iii. **Provides a valid *lower bound*** on the true AC OPF objective.
    iv. Often "exact" (matches the AC solution) for radial networks.
- **Cons:**
    - It's a **relaxation**. The solution might be "too good to be true" and not physically feasible on the real AC grid.

Insight: The Diagnostic Tool

Convex relaxations are powerful diagnostic tools for debugging network data.

Scenario: Your `solve_ac_opf` run (using `ACPPowerModel`) fails and reports `INFEASIBLE`.

- **The Problem:** You don't know why. Is the network data *truly* infeasible (e.g., not enough generation)? Or did the non-convex solver just get stuck?

The Test: Rerun the *same* problem using `SOCWRPowerModel`.

- This is a *relaxation*, so its "feasible space" is larger (easier to solve).
- If this *easier* problem also reports `INFEASIBLE`, you have a **mathematical proof** that your original problem is *genuinely* infeasible.

The `SOCWRPowerModel` can *prove* infeasibility in a way the "exact" `ACPPowerModel` cannot.

Table: Formulation Comparison

Formulation	Model Type	Convexity	Speed	Accuracy	Typical Use Case
`ACPPowerModel`	Non-Linear	**Non-Convex**	Slow	**Exact**	Final "ground truth" analysis.
`DCPPowerModel`	Linear	**Convex**	Very Fast	**Approximate**	Quick checks, large-scale studies.
`SOCWRPowerModel`	Second-Order Cone	**Convex**	Fast	**Relaxation**	Getting a *lower bound*; proving infeasibility.

6. The Two APIs

High-Level vs. Low-Level

Workflow 1: The High-Level API

- **Functions:** `solve_ac_opf`, `solve_dc_pf`, etc.
- **Description:** These are *convenience wrappers* for the most common problem/formulation pairings.
- **Example:** `solve_ac_opf(file, solver)` is just a shorthand for the low-level call: `solve_model(file, ACPPowerModel, solver, build_opf)`
- **When to Use:**
    - Application users.
    - Running standard problems.
    - Quick validation checks.

Workflow 2: The Low-Level API

- **Functions:** `solve_model(...)` or `build_model(...)`
- **Description:** The true, research-grade API. It forces you to be explicit, implementing the core philosophy: `solve_model(data, <Formulation>, solver, <Problem Spec>)`
- **When to Use:**
  - **For research (i.e., most of the time).**
  - To compare different formulations (see next slide).
  - To run advanced problems (like `build_tnep` or `build_ots`) that have no high-level wrappers.
  - To build *custom* models.

Code: Comparing Formulations

This example uses the **low-level API** to run the *exact same* OPF problem against three different mathematical models.

Code: Comparing Formulations (Part 1)

```julia
using PowerModels, Ipopt, JuMP

print("\033c")
println("Starting Formulation Comparison on case5.m...")

# --- Setup ---
pm_path = dirname(pathof(PowerModels))
case_file = joinpath(pm_path, "..", "test", "data", "matpower", "case5.m")
nl_solver = optimizer_with_attributes(Ipopt.Optimizer, "print_level" => 0)

# --- 1. Solve AC-OPF (The "Ground Truth") ---
# We use the generic solve_model, passing the Formulation (ACPPowerModel)
# and the Problem (build_opf) explicitly.
result_ac = solve_model(case_file, ACPPowerModel, nl_solver, build_opf)

println("AC-OPF (Exact, Non-Convex)")
println("  Status: ", result_ac["termination_status"])
println("  Objective: ", result_ac["objective"])
```

Code: Comparing Formulations (Part 2)

```julia
# --- 2. Solve DC-OPF (The Linear Approximation) ---
result_dc = solve_model(case_file, DCPPowerModel, nl_solver, build_opf)

println("\nDC-OPF (Linear Approximation)")
println("  Status: ", result_dc["termination_status"])
println("  Objective: ", result_dc["objective"])

# --- 3. Solve SOC-OPF (The Convex Relaxation) ---
result_soc = solve_model(case_file, SOCWRPowerModel, nl_solver, build_opf)

println("\nSOC-OPF (Convex Relaxation)")
println("  Status: ", result_soc["termination_status"])
println("  Objective: ", result_soc["objective"])

# --- 4. Analysis ---
println("\n--- Comparison Insight ---")
println("The SOC objective (", result_soc["objective"], ")")
println("is a provable *lower bound* for the AC objective (", result_ac["objective"], ").")
```

7. Interpreting and Exporting Results

The `result` Dictionary Structure

Every `solve_...` call returns a `result` dictionary. This structure is essential for extracting your solution.

Key	Data Type	Description
`"optimizer"`	String	The name of the solver class used (e.g., "Ipopt.Optimizer").
`"termination_status"`	Enum	The *solver's* final status (e.g., `OPTIMAL`, `INFEASIBLE`).
`"primal_status"`	Enum	Status of the primal solution (e.g., `FEASIBLE_POINT`).
`"solve_time"`	Float	The time (in seconds) the solver reported.

Key	Data Type	Description
`"objective"`	Float	The final objective function value (e.g., total cost).
`"objective_lb"`	Float	The best-known lower bound on the objective (if available).
`"solution"`	Dict	A **nested dictionary** containing all solution data.

Code: Accessing Solution Data

The solution values are inside `result["solution"]`. This dictionary *mirrors* the structure of your input data.

```julia
println("--- Accessing Solution Data ---")

# 1. Get simple top-level data
obj = result["objective"]
time = result["solve_time"]
println("Objective: $obj, Solve Time: $time")

# 2. Get data for a specific component
# Access bus "2" voltage angle ('va')
bus_2_va = result["solution"]["bus"]["2"]["va"]
println("Bus 2 Voltage Angle: $bus_2_va radians")

# Access generator "1" active power output ('pg')
gen_1_pg = result["solution"]["gen"]["1"]["pg"]
println("Gen 1 Active Power: $gen_1_pg p.u.")
```

Code: Accessing Solution Data (Cont.)

```julia
# 3. Get ALL bus voltage angles using a comprehension
all_vas = Dict(id => data["va"]
               for (id, data) in result["solution"]["bus"])
println("\nAll Bus Voltage Angles:")
display(all_vas)

# 4. Use the built-in summary tool
println("\n--- Full Solution Summary ---")
print_summary(result["solution"])

# 5. Update the original data with the solution
# This writes the solution values (vm, va, etc.)
# back into your input data dictionary.
# network_data = PowerModels.parse_file(case_file)
# PowerModels.update_data!(network_data, result["solution"])
```

9. Conclusions

Conclusions

1. **Core Philosophy:** `PowerModels.jl` is a research platform built on the **decoupling** of **Problems** ("what") from **Formulations** ("how").
2. **Two APIs:**
   - **High-Level (`solve_...`):** For simple, standard applications.
   - **Low-Level (`solve_model`):** For research, comparing formulations, and advanced problems.
3. **Formulations Matter:** The choice of formulation (`ACP`, `DC`, `SOC`) is a trade-off between **Accuracy**, **Speed**, and **Convexity**.
4. **Diagnostic Power:** Convex relaxations (`SOCWRPowerModel`) are powerful diagnostic tools for *proving* network infeasibility.

Conclusions

5. **Data-Centric:** The entire workflow is based on Julia `Dict`s, making data parsing, modification (for scenarios), and result extraction simple and robust.

8. Advanced Examples

Advanced Example 1: TNEP

This example shows how to use the **low-level API** to solve an advanced problem (`build_tnep`) that has no high-level `solve_...` wrapper.

We will solve a *relaxed* TNEP problem using `DCPPowerModel` for speed. A true TNEP is a Mixed-Integer problem and would require a solver like Juniper or Gurobi.

Code: TNEP Example

```julia
using PowerModels, Ipopt, JuMP

println("--- Running TNEP Example ---")
solver = optimizer_with_attributes(Ipopt.Optimizer, "print_level" => 0)

# Find the TNEP case file (which contains "ne_branch" data)
pm_path = dirname(pathof(PowerModels))
case_file = joinpath(pm_path, "..", "test", "data", "matpower", "case6_tnep.m")

# We MUST use the low-level API:
# Pass the data, the Formulation, the Solver, and the Problem
result = solve_model(case_file, DCPPowerModel, solver, build_tnep)

println("\n--- TNEP Results ---")
println("Total Cost (Op + Inv): ", result["objective"])

# Inspect the "build" decisions for candidate "ne_branch"
println("\nCandidate Branch Build-Out (Relaxed 0.0-1.0):")
for (id, branch) in result["solution"]["ne_branch"]
    println("  Branch $id: build_decision = ", branch["built"])
end
```

Advanced Example 2: Scenario Analysis

This example shows the power of the **dictionary-centric workflow**.

We will:

1. Load and solve a base case.
2. **Programmatically modify the data** (increase all loads by 20%).
3. Re-solve the new scenario case.
4. Compare the results.

Code: Scenario Analysis (Part 1)

```julia
using PowerModels, Ipopt, JuMP

println("--- Running Load-Increase Scenario ---")
pm_path = dirname(pathof(PowerModels))
case_file = joinpath(pm_path, "..", "test", "data", "matpower", "case14.m")
solver = optimizer_with_attributes(Ipopt.Optimizer, "print_level" => 0)

# 1. Parse and solve the base case
network_data = PowerModels.parse_file(case_file)
base_result = solve_ac_opf(network_data, solver)
println("Base Case Objective: ", base_result["objective"])

# 2. Create a NEW data dictionary for the scenario
# We must use deepcopy to avoid changing the original
scenario_data = deepcopy(network_data)
```

Code: Scenario Analysis (Part 2)

```julia
# 3. Modify the data programmatically
load_increase_factor = 1.20
for (id, load) in scenario_data["load"]
    load["pd"] = load["pd"] * load_increase_factor
    load["qd"] = load["qd"] * load_increase_factor
end
println("Modified all loads by $load_increase_factor...")

# 4. Solve the new scenario
scenario_result = solve_ac_opf(scenario_data, solver)

println("\nScenario Case Objective: ", scenario_result["objective"])
println("Termination Status: ", scenario_result["termination_status"])

if scenario_result["termination_status"]!= LOCALLY_SOLVED
    println("\n!!! WARNING: 20% load increase may be infeasible!!!!")
end
```

Advanced Example 3: Custom Models

This is the most advanced use case, illustrating the platform's extensibility for research.

You can write your *own* `build_...` function that:

1. Calls a standard `PowerModels.build_...` function (like `build_opf`).
2. Adds your **new, custom constraints** on top of it.

This allows you to add novel research constraints (e.g., new security rules, different generator models) into the standard OPF problem.

Code: Custom Model (Conceptual)

This pattern shows how a researcher would add a custom constraint.

```julia
# 1. Define your custom build function
function build_my_custom_opf(pm::AbstractPowerModel)
    # 2. Build the standard OPF problem first
    PowerModels.build_opf(pm)

    # 3. Get references to JuMP variables
    gen_power = var(pm, :pg)

    # 4. Add your new, custom constraint
    # (e.g., enforce pg[1] + pg[2] <= 5.0)
    @constraint(pm.model, gen_power[1] + gen_power[2] <= 5.0)
    println("Custom constraint added!")
end

# 5. Solve using your custom function
# result = solve_model(case_file, ACPPowerModel, solver,
#                      build_my_custom_opf)
```