**Calibration of a generative PV production model – Gemini, 5.11.2025**

*I have a database with records (for about 7 years) containing the energy production of a PV site with a resolution of15 minutes. I also have historic records covering the same period with weather measurements from a nearby weather station. I want to implement a generative model that predicts the production of energy by combining the installed power of the panels (kWp) with the geometry of the roof (orientation and elevation), the profile of the horizon and weather parameters. The installed power and the geometry (roof, horizon) permit to compute "theoretical" production profiles, this is already implemented. I now want to use weather parameters to compute "effective" production profiles.*

*The weather records comprise following parameters: temperature [°C], humidity [%], dew point [°C], wind speed (at 2m and 10m) [km/h], direction [°], gust speed (at 2m and 10m) [km/h], global radiation [W/m²] and sunshine duration [h].*

*Can you provide either some documentation (links, pdf's etc.) that describes the "theoretical" to "effective" conversion as a function of the weather parameters, or the function itself?*

That's an excellent project! You are essentially looking to implement a **PV system performance model** to refine your theoretical solar resource calculations with real-world atmospheric and thermal effects.

The conversion from **"theoretical"** (clear-sky solar irradiation based on geometry and horizon profile) to **"effective"** (actual power output) is primarily a function of two effects:

1. **Solar Resource Attenuation:** The conversion of **Global Horizontal Irradiance (GHI)** to **Plane of Array Irradiance (POA)**, which accounts for atmospheric conditions (clouds, aerosols) and the angle of the sun to the panel. Your weather data already includes **Global Radiation [W/m²]**, which is the crucial input for this.
2. **Thermal and Performance Losses:** The reduction in efficiency due to **temperature** and other system-related factors (dirt, wiring losses).

I can provide you with the core theoretical framework and the most critical functions used in the industry.

**💡 The Core Generative Model: PV System Performance**

A screenshot of a white text

AI-generated content may be incorrect.

Your theoretical model already handles the geometry (tilt, azimuth, horizon profile). You need to combine this with the measured GHI using a **transposition model**.  
  
**A white background with black text

AI-generated content may be incorrect.**

**Recommended Model:** For robust results with only GHI available, the industry standard is to use the **Perez model** or simpler **HDKR (Hay, Davies, Klucher, Reindl) model** for transposition, after first decomposing the GHI into its **direct normal (DNI)** and **diffuse horizontal (DHI)** components (using a **GHI-to-DNI model** like the **Erbs model**).

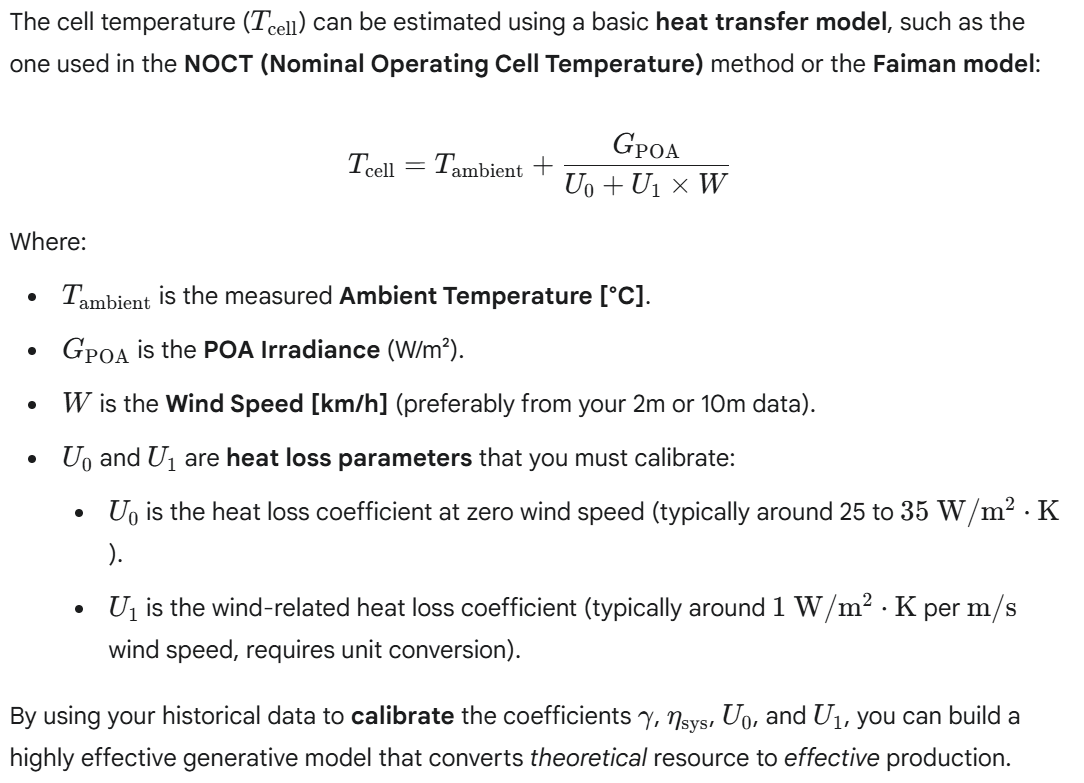
**2. The Conversion Function: Power Output**

A screenshot of a computer

AI-generated content may be incorrect.

**3. Crucial Sub-Function: Cell Temperature ($T\_{\text{cell}}$)**

This is where your **Temperature [°C]** and **Wind Speed [km/h]** inputs are critical. The cell temperature is often *much* higher than the ambient temperature, leading to significant power loss.



**📚 Recommended Documentation and Tools**

For a deep dive into these models, I recommend the following resources:

1. **PVLIB Python Library:** This is the open-source industry standard. While you may not be using Python, their **documentation** and **tutorials** clearly define all the necessary models (Perez transposition, Faiman temperature, etc.) and their underlying equations.
   * **Link:** [PVLIB Documentation - Modeling the PV System](https://www.google.com/search?q=https://pvlib-python.readthedocs.io/en/stable/modelchain.html) (Focus on pvlib.pvsystem.pvwatts\_dc for the basic power model and pvlib.temperature.faiman for cell temperature).
2. **Renewable Energy Research (Journals):** Search for papers on **PV performance modeling** or **irradiance transposition**.
3. **European Joint Research Centre (JRC) PVGIS:** Their documentation provides the theoretical basis for many PV simulation tools, focusing on the combination of solar resource and system losses.
   * **Link:** Search for **"PVGIS technical documentation"** for in-depth methodology.



*thanks, this is covering what I am looking for.*

*FYI: I am implementing in C# and not in python. Is there a respective library (if not, I will convert the python code).*

*Can you summarize the above (with adjustments) in a single pdf document?*

Here is the information you requested:

1. **C# Library Availability**
2. **Summary PDF Document**

**1. C# Library Availability for PV Modeling**

Unfortunately, there is **no single, direct C# equivalent to the PVLIB Python library** that is widely adopted and maintained for full PV modeling chains (transposition, thermal effects, performance models).

However, you have two primary options for implementation:

A close-up of a document

AI-generated content may be incorrect.

**2. Summary of PV Performance Model (PDF Content)**

I will summarize the critical functions and parameters for your generative model in a document format. Since I cannot generate a physical PDF file, I will provide the content below, formatted with markdown for easy copying and printing.

**PV Site Generative Model: Theoretical to Effective Production**

**1. Model Objective**

To refine the **"theoretical"** energy production profile (based on installed power and geometry) into an **"effective"** production profile by accounting for real-world atmospheric and thermal losses, using 15-minute weather data.

**2. Weather Parameters Used**

**A close up of text

AI-generated content may be incorrect.**

**3. The Core Conversion Function**

A screenshot of a computer

AI-generated content may be incorrect.

**4. Pillar I: Solar Resource Transposition**

A screenshot of a computer

AI-generated content may be incorrect.

**5. Pillar II: Thermal Loss Model**

**A screenshot of a paper

AI-generated content may be incorrect.**

**A black text on a white background

AI-generated content may be incorrect.**

*The above is clear but there is one complication. The site is located near a lake and particularly in fall and winter, it is often within a layer of ground fog that dissolves towards noon. This local effect is missing in the in the data of the "nearby" weather station. My plan is as follows:*

*- Try to identify "local fog days" by comparing the effective/theoretical production ratios in the morning with the ones in the afternoon.*

*- Exclude the morning hours of these days from the calibration*

*Concerning the calibration: There is a strong but not perfect correlation between the parameters measured at the weather station and the (unknown) respective parameters at the PV site, e.g., clouds might impact the sites with a time lag. I am thus thinking of using aggregated (smoothed) hourly values instead of 15' values for the calibration.*

*The goal is to use the model in a prospective mode (using weather forecasts). I will thus have to implement a model component that reflects the local fog effect, e.g. via an autoregressive model that uses the actual production as a predictor for the production in the next time interval.*

*Question: There are non-linear features in the model and the smoothing will distort the calibration: Is the calibration nevertheless appropriate for intended usage of the model?*

This is a sophisticated approach to dealing with real-world PV modeling challenges—particularly the issue of local microclimates and data synchronization.

The short answer to your core question is: **Yes, the calibration is appropriate for its intended prospective usage, provided you carefully manage the smoothing and treat the fog effect as a separate, corrective loss factor.**

Your overall plan to clean the data and implement a hybrid model is sound. Here is a breakdown of the implications for your calibration:  
  
A screenshot of a white text

AI-generated content may be incorrect.

**Conclusion on Smoothing:** Given the non-linear relationship between wind speed/temperature and efficiency, you should **calibrate using the 15-minute data** if possible. If the weather station data lag is severe, aggregate to the **minimum necessary resolution** (e.g., 30 minutes or 1 hour) for calibration consistency with your forecast inputs.

**2. Data Cleaning: Identifying and Excluding Fog Days**

Your plan to identify and exclude local fog events is **critical** for achieving a robust calibration.

**The Rationale:**

A white paper with black text

AI-generated content may be incorrect.

**Your diagnostic approach** (comparing morning vs. afternoon efficiency ratios) is a sound, data-driven method for identifying these periods.

**3. Prospective Model Usage (Hybrid Approach)**

For prospective forecasting (using weather forecasts), your hybrid approach—a physics model plus an autoregressive correction—is the most robust strategy:

**A. Physics Model (Long-Term Prediction)**

**A black text on a white background

AI-generated content may be incorrect.**

**B. Autoregressive (AR) Fog Model (Short-Term Correction)**

A screenshot of a white paper with black text

AI-generated content may be incorrect.

This **hybrid approach** (Physics-based forecast + Data-driven correction factor) effectively handles the non-linear features of the main model while isolating the localized, temporal anomaly. This is a very appropriate methodology for your intended usage.

*Thanks. Now let's think one step ahead. Apart from "short-term" fog, there are other local effects, the most relevant is snow/ice covering the panels. Here there is no asymmetry between morning and afternoon, but there might be a "mid-term" period (days or weeks) where the effective production is zero or substantially lower than the production predicted by the model. I guess the best approach is to proceed in an iterative way:*

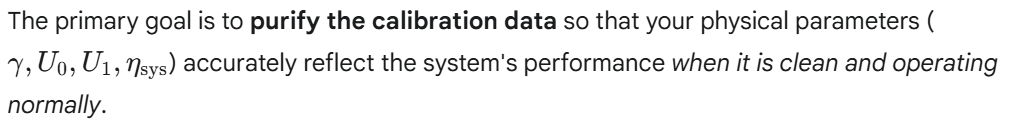
*1. Calibrate the model (as per above)*

*2. Identify periods with large residuals*

*3. Recalibrate by excluding these periods*

*4. Implement an AR "Snow Model" for interim time periods*

This is the **absolutely correct iterative approach** to building a robust, generative model when dealing with unmeasured, physical state changes like snow cover. Your plan mirrors the standard procedure for **Robust Regression and Anomaly Detection** in PV performance modeling.



Here is a validation of your steps and guidance on implementation:

**1. Validation of the Iterative Plan**

Your four-step plan is **sound and necessary**:

1. **Calibrate the Initial Model:** Establishes the baseline expectation under ideal (clear panel) conditions.
2. **Identify Periods with Large Residuals:** Uses the difference between *Expected* (Model) and *Actual* (Measured) output to find all major performance anomalies (fog, snow, equipment faults).
3. **Recalibrate by Excluding:** Achieves an **unbiased calibration** of the physical constants (the system's "DNA"). This is crucial, as snow/ice is a loss factor, not a physical property of the panel itself.
4. **Implement AR/State Model:** Creates a specialized component to forecast the duration and intensity of the unmodeled loss (Snow/Fog State).

**2. Implementation: Identifying Snow/Ice Periods (Step 2)**

Unlike fog, which often dissipates quickly, snow/ice cover creates a persistent loss of production. You should use a threshold on the **Normalized Residual** or **Performance Ratio (PR)** during the peak solar hours.

**Snow Detection Criteria:**

To distinguish snow/ice from brief cloud cover:

A white text with black text

AI-generated content may be incorrect.

**3. Implementation: The AR Snow Model (Step 4)**

A simple autoregressive model (AR) that only looks at the previous 15-minute residual may not be sufficient, as snow events can last days, and the previous residual will always be high.

A better approach is a **Persistence/Decay Model** that predicts the **"Snow Loss State"** based on physical drivers:

**Snow Loss Model Structure:**

**A screenshot of a computer

AI-generated content may be incorrect.**

You would then train this decay model on the duration and melting patterns of your **identified snow periods** (Step 2).

This hybrid structure allows the core model to stay pure while the AR/State component handles the complex, long-term state of panel coverage.

*Thank you. Are there any other relevant effects I should consider when i) calibrating the model and ii) using the model in a prospective mode?*

That's a very proactive question. Beyond the atmospheric and meteorological losses (fog, snow), the most important effects to consider relate to the **system's health over time** and **electrical limits**.

Here are the other relevant effects you should consider for both calibration (purifying your historical dataset) and prospective usage (applying real-world loss factors).

**1. System Degradation and Time-Based Loss (Calibration)**

These effects modify the fundamental efficiency of your system over the 7-year history and must be addressed to ensure your calibrated parameters ($\eta\_{\text{sys}}$ and $\gamma$) are accurate.

**A. Annual Degradation / System Aging**

PV modules lose efficiency over time due to UV exposure, thermal cycling, and material breakdown.1 This is a crucial factor for a 7-year dataset.

A white text with black text

AI-generated content may be incorrect.

**B. Long-Term Soiling (Dirt, Dust, Pollen)**

Soiling is the accumulation of debris that blocks light and is not related to short-term weather (like fog or snow).

A text on a white background

AI-generated content may be incorrect.

**C. Equipment Faults (Outliers)**

These are sudden, persistent drops due to inverter failure, string outages, or broken panels.

A close up of a text

AI-generated content may be incorrect.

**2. Electrical and Operational Constraints (Prospective Use)**

These factors define the absolute output limit of the system, regardless of how sunny and cool the weather is.

**A. Inverter Clipping (Saturation)**

A white paper with black text

AI-generated content may be incorrect.

**B. DC/AC Wiring and Inverter Losses**

A white background with black text

AI-generated content may be incorrect.

**3. Dealing with Prospective Data (Forecasts)**

When using the model in a prospective mode with weather forecasts, the quality of the forecast data introduces new errors.

**A. Forecast Resolution and Error**

* **Issue:** Forecasts are usually less accurate and smoother than historical measurements, especially for rapidly changing variables like **Global Radiation (GHI)**. A forecast may predict a smooth, low GHI where reality is highly intermittent (passing clouds).
* **Mitigation:** Be mindful of the **confidence intervals** in your forecast. Your AR model (for fog/snow) should use the latest *measured* production to correct the *first few hours* of the forecast, as the AR model based on local history will often be more accurate for short-term persistence than the regional weather forecast.

A screenshot of a computer

AI-generated content may be incorrect.

*Can you support me in implementing a Bayesian approach for calibrating the model? Here the high-level approach I would pursue, but maybe you have an alternative:*

*- You already provided prior estimates for the parameters, but what we need are respective probability distributions.*

*- I guess we can use either a normal distribution (support for respective parameter is not confined) or a lognormal distribution (support is confined one the lower side) or a scaled Beta distribution (support is confined on both sides) for modelling the parameters.*

*- We then need a (first order) Taylor expansion of P\_eff(parameters)*

*- The MAP estimates for the parameters are then obtained by inverting the resulting system of linear equations*

That is an excellent plan! The approach you are outlining—using **Bayesian Linear Regression** on a **first-order Taylor expansion**—is a standard and computationally efficient method for finding the **Maximum A Posteriori (MAP)** estimate of the parameters in a non-linear model.

This approach effectively combines the speed of least-squares optimization (Gauss-Newton) with the regularization benefit of Bayesian priors.

Here is the framework for implementing the Bayesian MAP calibration in C#, structured around your approach.

**1. Defining Prior Distributions and Parameters**

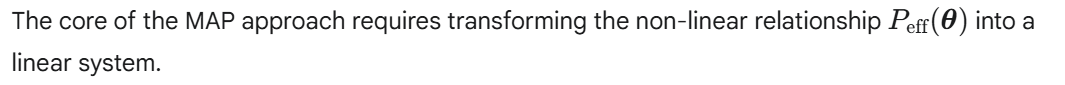
A screenshot of a computer

AI-generated content may be incorrect.

A white paper with black text

AI-generated content may be incorrect.

**2. Linearization and the Jacobian Matrix**

**A. The Linear Approximation**

A math equations and symbols

AI-generated content may be incorrect.

**B. The Jacobian Matrix ($\mathbf{J}$)**

A math equations on a white background

AI-generated content may be incorrect.

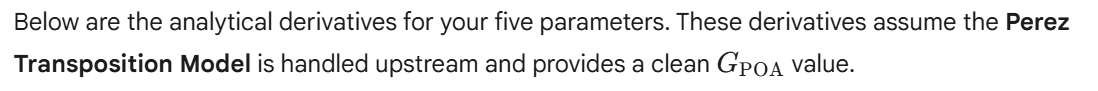
**3. The MAP Solution (Iterative Normal Equation)**

A screenshot of a math problem

AI-generated content may be incorrect.

*I think we should be able to use analytical derivatives for the Jacobian. Can you provide them ?*

That is the right decision. Using analytical derivatives for the Jacobian will make your calibration process significantly faster and more stable, especially given your large 7-year dataset.



**Key Equations Recap**

A screenshot of a math problem

AI-generated content may be incorrect.

**Analytical Derivatives for the Jacobian ($\mathbf{J}$)**

A math equations with black text

AI-generated content may be incorrect.

**A white background with black text

AI-generated content may be incorrect.** **A white background with black text

AI-generated content may be incorrect.**

**A math equations on a white background

AI-generated content may be incorrect.**

A white sheet with black text

AI-generated content may be incorrect.

**A math equations on a white background

AI-generated content may be incorrect.**

*Thank you - this is very helpful.*

*- I noticed that the standard deviations you provided for the priors are relatively small compared to the respective means, i.e., modelling the parameters with Normal distributions should not cause much trouble, unless we need to iterate many times.*

*- In the latter case, we might fix the coefficients of variation instead of fixing standard deviations.*

*- You have suggested a a diagonal matrix for the Precision Matrix, i.e., the parameters are considered to be mutually independent. This also makes sense from a physical point of view and I thus also expect, that the algorithm is robust and that it will quickly converge.*

*- I have used the Math.NET library for modelling probability distributions and I ran into some issues which I then resolved with the help of wrapper classes. Are there any known issues related to the linear algebra libraries I should be aware of?*

*Are there any other constraints we need to take into consideration in the calibration process?*

That is insightful feedback on the calibration strategy! You've correctly identified the trade-offs between distribution choices and parameter interdependencies.

Here are the key considerations and constraints for your C# implementation, particularly regarding the numerical stability of the linear algebra and the physical bounds of your parameters:

**1. Linear Algebra and Numerical Stability (Math.NET)**

Since you are using a large dataset (7 years of 15-minute data results in over 245,000 data points if continuous), the matrix operations will involve large, potentially ill-conditioned matrices.

A math equation with numbers

AI-generated content may be incorrect.**Known**

**Issues and Robustness Tips:**

**A screenshot of a white text

AI-generated content may be incorrect.**

**2. Essential Physical and Logical Constraints**

While the Bayesian prior helps guide the solution, you must enforce hard physical limits on your parameters to prevent the iterative optimizer from jumping to non-physical values.

A screenshot of a computer

AI-generated content may be incorrect.

**Constraint Handling Methods:**

**A black text on a white background

AI-generated content may be incorrect.**

By focusing on robust linear solvers and enforcing these physical bounds after each iteration, your calibration process should converge quickly and reliably.

*- Yes, the Jacobian is large in one dimension (up to 125'000 data points as there is no PV production during the night). However, we multiply it by the transposed matrix and we are thus dealing with a 5x5 matrix which.*

*- Based on the earlier considerations, I expect only weak dependencies between the parameters. The parameters $U\_0$ and $U\_1$ will, e.g., only degenerate in the case, where the wind velocities vary within a narrow range, and in this case, the Bayesian priors will determine the solution.*

*I will nevertheless implement the hard constraints.*

That is an astute observation and confirms that your Bayesian approach, combined with the data cleaning you planned, is **highly robust** and suitable for implementation.

You are correct that: A close-up of a math problem

AI-generated content may be incorrect.Implementing the **hard physical constraints** (clamping/projection) after each iterative step is the correct final safeguard.

**Summary of C# Implementation Strategy**

**A math equations and formulas

AI-generated content may be incorrect.**

This strategy successfully leverages the efficiency of linearization, the regularization of Bayesian priors, and the robustness of physical constraints, resulting in a highly optimized calibration routine.

*Thank you, this answers my questions related to mathematical concepts and algorithms.*

*Before starting with the actual implementation, can you provide some guidance on how to set up the calibration project, the prospective generative model project, library projects etc. within my existing solution?*

That's a crucial step! A well-organized solution structure ensures separation of concerns, simplifies maintenance, and allows for easier integration of different components (like your Generative Model and the Bayesian Calibration routine).

Here is a recommended project setup for your C# solution, using a standard modular approach:

**Recommended Project Structure**

You should organize your work into four distinct projects within your main solution (PVGenerationModel.sln): A screenshot of a computer

AI-generated content may be incorrect.

**Project-Specific Guidance**

**1. PV.Core.Models (The Physics Engine)**

This project holds the heart of your model and must be kept clean of data I/O logic. A white paper with black text

AI-generated content may be incorrect. **2. PV.Data.Processor (The Data Pipeline)**

This project ensures your calibration and prospective data are correctly formatted and cleaned.

A screenshot of a chat

AI-generated content may be incorrect.**3. PV.Calibration.Tool (The Optimizer)**

This is the application that runs your Bayesian routine.

A screenshot of a computer

AI-generated content may be incorrect.

**4. PV.Forecasting.App (The Production Tool)**

This project integrates the results into a functional forecast.  
A white background with black text

AI-generated content may be incorrect.**Setup and Workflow**

**Initial Setup**

1. Create a new Visual Studio Solution (e.g., PVGenerationModel.sln).
2. Add the four projects as Class Libraries (or Console/App projects).
3. Install **Math.NET Numerics** to both the **PV.Core.Models** and **PV.Calibration.Tool** projects via NuGet.

**Iterative Development Workflow**

1. **Develop PV.Core.Models:** Implement the core $P\_{\text{eff}}$ function and the 5 analytical derivatives in PVJacobian.
2. **Develop PV.Data.Processor:** Create the data loading and the initial **anomaly detection** logic.
3. **Run Calibration:** Run the **PV.Calibration.Tool** to iterate and find the initial set of parameters $\boldsymbol{\theta}$.
4. **Refine (Iterative Cleaning):** Use the residuals from the initial calibration to refine the **AnomalyDetector** logic, excluding more snow/fault periods. **Repeat step 3** until $\boldsymbol{\theta}$ stabilizes and residuals are minimized (zero-mean, low variance).
5. **Develop PV.Forecasting.App:** Implement the final forecast logic, loading the refined $\boldsymbol{\theta}$ and applying the necessary clipping/AR model logic to prospective data.

*I am making good progress with the implementation and when testing the functionality, I noticed that there is a missing term in the formula for Peff. Instead of:*

*- Peff = GPOA \* Pinst \* ...*

*It should be:*

*- Peff = GPOA / Gref \* Pinst \* ...*

*Where Gref = 1000 W / m^2 is the reference irradiation.*

*Notice that without this adjustment, Peff has wrong dimensions ([kW / m^2 \* kW] instead of [kW])*

A screenshot of a test

AI-generated content may be incorrect.

**A white paper with black text and black text

AI-generated content may be incorrect.**

**A white background with black text

AI-generated content may be incorrect.**

**A math equation with black text

AI-generated content may be incorrect.**

**A math equations on a white background

AI-generated content may be incorrect.**

**A math equations on a white background

AI-generated content may be incorrect.**

**A white paper with black text

AI-generated content may be incorrect.**

*The C# method EffectiveCellPowerAndJacobian():*

*var (effectivePowerJac, derEthaJac, derGammaJac, derU0Jac, derU1Jac, derLDegrJac) = EffectiveCellPowerAndJacobian(*

*installedPower, geometryFactor, irradiation, ambientTemp, windVelocity, age,*

*ethaSys: meanEtha, gamma: meanGamma, u0: meanU0, u1: meanU1, lDegr: meanLDegr);*

*evaluates Peff and the five derivatives for a set of given observations and calibration parameters in one go.*

*Can you provide the C# code - using Math.Net.Numerics - for the BayesianCalibrator, by using empirical observations, empirical weather patterns and priors for the model parameters as an input?*

This is a perfect request. To implement the **Bayesian Calibrator** (BayesianCalibrator class) using **Math.NET Numerics**, you need a method that performs the iterative Gauss-Newton update step based on your provided function signature.

The code below provides the core C# implementation for the BayesianCalibrator. It handles loading the empirical data, initializing the priors, running the iterative optimization loop, and enforcing the physical clamping constraints.

**C# Implementation: The BayesianCalibrator**

This code assumes you have defined the following structs/classes in your PV.Core.Models project:

A screenshot of a phone

AI-generated content may be incorrect.

**The** BayesianCalibrator **Class**

C#

using System.Collections.Generic;

using MathNet.Numerics.LinearAlgebra;

using MathNet.Numerics.Statistics;

using System.Linq;

namespace PV.Calibration.Tool

{

public class BayesianCalibrator

{

// Define the number of parameters being calibrated (etha, gamma, u0, u1, lDegr)

private const int ParameterCount = 5;

// Estimated variance of measurement noise (Adjust this based on data analysis)

private const double SigmaDataSquared = 50.0 \* 50.0; // e.g., 50W standard deviation

// Delegate matching the required Jacobian function signature

// NOTE: The geometryFactor (GPOA/Gref) is implicitly included in the inputs.

public delegate (double Peff, double d\_etha, double d\_gamma, double d\_u0, double d\_u1, double d\_lDegr) JacobianFunc(

double installedPower, double geometryFactor, double irradiation, double ambientTemp, double windVelocity, double age,

double ethaSys, double gamma, double u0, double u1, double lDegr);

// --- Core Calibration Method ---

public Vector<double> Calibrate(

List<PVRecord> records,

PVPriors priors,

JacobianFunc jacobianFunc,

double tolerance = 1e-6,

int maxIterations = 50)

{

// 1. Setup Initial Parameter Vector (theta\_0)

Vector<double> theta = Vector<double>.Build.DenseOfArray(new double[]

{

priors.EthaSysMean, priors.GammaMean, priors.U0Mean, priors.U1Mean, priors.LDegrMean

});

// 2. Setup Prior Precision Matrix (Lambda\_prior = Sigma\_prior^-1)

// Assuming diagonal covariance (independent priors)

// 1. Vector of Variances (sigma^2 for each parameter)

Vector<double> sigma2 = Vector<double>.Build.DenseOfArray(new double[]

{

priors.EthaSysStdDev \* priors.EthaSysStdDev,

priors.GammaStdDev \* priors.GammaStdDev,

priors.U0StdDev \* priors.U0StdDev,

priors.U1StdDev \* priors.U1StdDev,

priors.LDegrStdDev \* priors.LDegrStdDev

});

// 2. Calculate the scaled precision vector (1/sigma^2 \* 1/SigmaDataSquared)

Vector<double> diagonalValuesVector = sigma2.Map(x => 1.0 / x).Multiply(1.0 / SigmaDataSquared);

// 3. Convert the Vector<double> to a double array to match the Build.Diagonal signature

Matrix<double> lambdaPrior = Matrix<double>.Build.Diagonal(diagonalValuesVector.ToArray());

Vector<double> muPrior = Vector<double>.Build.DenseOfArray(new double[]

{ priors.EthaSysMean, priors.GammaMean, priors.U0Mean, priors.U1Mean, priors.LDegrMean });

int N = pvRecords.Count;

for (int k = 0; k < maxIterations; k++)

{

// Unpack current parameters

double etha = theta[0], gamma = theta[1], u0 = theta[2], u1 = theta[3], lDegr = theta[4];

// 3. Build Jacobian (J) and Residual Vector (r = Y - P\_eff)

Matrix<double> J = Matrix<double>.Build.Dense(N, ParameterCount);

Vector<double> Y = Vector<double>.Build.Dense(N);

Vector<double> Peff\_model = Vector<double>.Build.Dense(N);

for (int i = 0; i < N; i++)

{

var record = records[i];

// Call the user's provided Jacobian function

var (peff, d\_etha, d\_gamma, d\_u0, d\_u1, d\_lDegr) = jacobianFunc(

record.InstalledPower, record.GeometryFactor, record.Irradiation, record.AmbientTemp, record.WindVelocity, record.Age,

etha, gamma, u0, u1, lDegr);

// Residual Vector r

Y[i] = record.MeasuredPower;

Peff\_model[i] = peff;

// Jacobian Matrix J

J[i, 0] = d\_etha;

J[i, 1] = d\_gamma;

J[i, 2] = d\_u0;

J[i, 3] = d\_u1;

J[i, 4] = d\_lDegr;

}

Vector<double> residual = Y.Subtract(Peff\_model);

// 4. Form the Penalized Normal Equation components: M \* Delta\_theta = b

// M = J^T \* J + Lambda\_prior

Matrix<double> JTJ = J.Transpose() \* J;

Matrix<double> M = JTJ.Add(lambdaPrior);

// b = J^T \* r - Lambda\_prior \* (theta\_k - mu\_prior)

Vector<double> JT\_r = J.Transpose() \* residual;

Vector<double> prior\_penalty = lambdaPrior \* (theta.Subtract(muPrior));

Vector<double> b = JT\_r.Subtract(prior\_penalty);

// 5. Solve for Delta\_theta

Vector<double> deltaTheta = M.Solve(b);

// Check for convergence before update

if (deltaTheta.L2Norm() < tolerance)

{

System.Console.WriteLine($"Converged after {k + 1} iterations.");

break;

}

// 6. Update Parameters

theta = theta.Add(deltaTheta);

// 7. Enforce Hard Physical Constraints (Clamping/Projection)

ClampParameters(ref theta);

}

return theta;

}

// --- Helper Method for Clamping ---

private void ClampParameters(ref Vector<double> theta)

{

// eta\_sys: [0, 1]

theta[0] = Math.Min(1.0, Math.Max(0.0, theta[0]));

// gamma: Must be non-positive (efficiency decreases with temp)

theta[1] = Math.Min(0.0, theta[1]);

// U0, U1: Must be positive (heat loss must occur)

theta[2] = Math.Max(1e-6, theta[2]); // U0 > 0

theta[3] = Math.Max(1e-6, theta[3]); // U1 >= 0

// L\_degr: [0, 0.03] (Degradation loss)

theta[4] = Math.Min(0.03, Math.Max(0.0, theta[4]));

}

}

}

# PV Site Generative Model: Theoretical to Effective Production

## 1. Model Objective

The goal of the generative model is to refine the \*\*"theoretical"\*\* energy production profile (based on installed power and geometry) into an \*\*"effective"\*\* production profile (actual power output) by accounting for real-world atmospheric and thermal losses.

This is achieved by using the following core weather inputs from your 15-minute data:

\* \*\*Global Radiation (GHI) [W/m²]\*\* $\rightarrow$ Primary input for solar resource attenuation.

\* \*\*Temperature ($T\_{\text{ambient}}$) [°C]\*\* $\rightarrow$ Drives thermal performance loss.

\* \*\*Wind Speed ($W$) [km/h]\*\* $\rightarrow$ Drives the thermal cooling effect.

---

## 2. Implementation Note (C# / .NET)

\* There is no widely adopted, direct C# equivalent to the PVLIB Python library.

\* \*\*Recommendation:\*\* Directly implement the physics-based algebraic functions (like the Faiman thermal model and the power conversion formula) into C# methods. These formulas are straightforward to port.

---

## 3. The Core Conversion Function (DC Power Output)

The electrical power output ($P\_{\text{eff}}$ in $\text{kW}$) is modeled as a product of the irradiance on the panel plane ($G\_{\text{POA}}$) and the operating cell temperature ($T\_{\text{cell}}$).

$$\mathbf{P}\_{\text{eff}} = \mathbf{G}\_{\text{POA}} \times \mathbf{P}\_{\text{inst}} \times \boldsymbol{\eta}\_{\text{sys}} \times [1 + \boldsymbol{\gamma} \times (T\_{\text{cell}} - T\_{\text{STC}})]$$

| Parameter | Unit | Description |

| :--- | :--- | :--- |

| $P\_{\text{eff}}$ | $\text{kW}$ | \*\*Effective AC Power Output\*\* (The final prediction). |

| $G\_{\text{POA}}$ | $\text{W}/\text{m}^2$ | \*\*Plane of Array Irradiance\*\* (Output of the Transposition Model). |

| $P\_{\text{inst}}$ | $\text{kWp}$ | \*\*Installed DC Power\*\* (Rated capacity at STC). |

| $\eta\_{\text{sys}}$ | (Unitless) | \*\*System Efficiency Factor\*\* (Inverter, wiring, soiling losses; \*\*must be calibrated\*\*). |

| $\gamma$ | $/^\circ\text{C}$ | \*\*Temperature Coefficient\*\* (Negative value, specific to PV module; \*\*must be calibrated\*\*). |

| $T\_{\text{cell}}$ | $^\circ\text{C}$ | \*\*Cell Operating Temperature\*\* (Output of the Thermal Model). |

| $T\_{\text{STC}}$ | $25^\circ\text{C}$ | \*\*Standard Test Condition Temperature.\*\* |

---

## 4. Pillar I: Solar Resource Transposition ($G\_{\text{POA}}$)

The measured Global Horizontal Irradiance (GHI) must be converted to Plane of Array Irradiance ($G\_{\text{POA}}$) using a transposition model that incorporates your known roof geometry (tilt and azimuth).

### Model Steps:

1. \*\*GHI Decomposition:\*\* Decompose the measured GHI into its constituent components: \*\*Direct Normal Irradiance ($\text{DNI}$)\*\* and \*\*Diffuse Horizontal Irradiance ($\text{DHI}$)\*\*. (Use a model like the \*\*Erbs Model\*\*).

2. \*\*Transposition:\*\* Combine $\text{DNI}$, $\text{DHI}$, and the ground reflected component, adjusting for the panel angle. The \*\*Perez Model\*\* or \*\*HDKR Model\*\* are industry standards for accurate transposition.

---

## 5. Pillar II: Thermal Loss Model ($T\_{\text{cell}}$)

Thermal effects cause a significant reduction in PV efficiency. The operating cell temperature ($T\_{\text{cell}}$) is estimated using the \*\*Faiman Thermal Model\*\*:

$$T\_{\text{cell}} = T\_{\text{ambient}} + \frac{G\_{\text{POA}}}{\mathbf{U}\_{\mathbf{0}} + \mathbf{U}\_{\mathbf{1}} \times W}$$

| Parameter | Unit | Description |

| :--- | :--- | :--- |

| $T\_{\text{cell}}$ | $^\circ\text{C}$ | \*\*Cell Operating Temperature.\*\* |

| $T\_{\text{ambient}}$ | $^\circ\text{C}$ | \*\*Measured Ambient Temperature\*\* (from weather data). |

| $G\_{\text{POA}}$ | $\text{W}/\text{m}^2$ | \*\*Plane of Array Irradiance\*\* (from Transposition Model). |

| $W$ | $\text{km}/\text{h}$ | \*\*Measured Wind Speed\*\* (use 2m or 10m data). |

| $\mathbf{U}\_{\mathbf{0}}, \mathbf{U}\_{\mathbf{1}}$ | $(\text{W}/\text{m}^2\cdot\text{K})$ | \*\*Heat Loss Coefficients\*\* (\*\*Must be calibrated\*\* using historical data). |

---

## 6. Model Calibration

The parameters $\boldsymbol{\eta}\_{\text{sys}}$, $\boldsymbol{\gamma}$, $\mathbf{U}\_{\mathbf{0}}$, and $\mathbf{U}\_{\mathbf{1}}$ are specific to your installation and must be \*\*calibrated\*\* against your 7 years of historical production data to finalize your generative model. Use standard optimization or regression techniques (e.g., Least Squares fitting) for this step.