# The Symbo Owner's Manual: A Guide to Generative Symbolic Reasoning

## 1.0 A New Paradigm: Introduction to the Symbo Engine

The Symbo engine is not an incremental improvement on existing tools; it represents a new model class engineered to address a fundamental challenge in modern AI. Its strategic purpose is to bridge the chasm between fully interpretable, classical symbolic systems and adaptive, but often opaque, generative models. By doing so, Symbo provides a robust, transparent alternative to "black-box" deep learning, enabling solutions that are both computationally flexible and mathematically auditable.

At its core, Symbo is a hybrid symbolic-numeric reasoning engine. It was built by deconstructing a corpus of 318 classical algorithms in algebra, optimization, and computational geometry and reassembling their atomic computational primitives into a cohesive, generative architecture. This design allows it to fuse the exactitude of algebraic semantics with the adaptability of modern generative modeling.

The core mission of the Symbo project is defined by four key objectives:

- **Represent complex systems symbolically** as generative manifolds, allowing for the creation of flexible and interpretable policy functions.
- **Perform exact, deterministic reasoning** using classical algebraic tools like Gröbner bases to solve complex nonlinear constraints without approximation.
- **Generate perturbative and approximate models** for intricate nonlinear systems, particularly in domains like dynamic systems modeling.
- **Ensure full mathematical interpretability and portability** for deployment in diverse environments, from high-performance computing clusters to web browsers via WebAssembly (WASM).

These capabilities are enabled by a foundational design philosophy that prioritizes the decomposition and generative recombination of mathematical concepts.

## 2.0 Core Philosophy and System Architecture

The strategic power of the Symbo engine originates from its unique design philosophy. Rather than adopting monolithic algorithms from the classical computational canon, Symbo's architecture is the result of a deliberate process of deconstruction and generative recombination. This decompositional approach frees computational primitives from their rigid, special-purpose origins, enabling their recombination into novel, general-purpose reasoning workflows that were impossible with the original monolithic algorithms.

## Algorithmic Decomposition

The foundation of Symbo was laid by meticulously deconstructing **318 foundational algorithms** spanning algebra, optimization, dynamics, and computational geometry. Each algorithm was reduced to its atomic computational primitives, which were then "de-parented" from their original context. This reassembled architecture is not arbitrary; it is a layered system where each stratum executes a specific class of the recombined primitives—from core symbolic manipulation in the `Tensor Core` to high-level geometric logic in the `Reasoning Layer`.

## System Architecture Overview

This reconstructed system is organized into a clean, layered architecture, where each layer serves a distinct function in the reasoning process.

| Layer | Main Functions |
|---|---|
| **Symbolic Tensor Core** | Defines `NanoTensor` objects and provides core symbolic operators for differentiation and substitution. |
| **Solver Layer** | Provides exact algebraic solvers using Gröbner bases, perturbation theory, and resultants. |
| **Trainer Layer** | Implements methods for fitting `NanoTensor` coefficients via symbolic, perturbative, or least-squares estimation. |
| **Interop Layer** | Handles high-speed serialization (MessagePack, Arrow) and supports WebAssembly (WASM) execution. |
| **Reasoning Layer** | Enables high-level reasoning through A* pathfinding on symbolic potential surfaces and explainability via derivative trees. |

This entire layered architecture is built around a single, powerful data structure that serves as the engine's central representational object: the `NanoTensor`.

## 3.0 The Foundational Object: Understanding the `NanoTensor`

The `NanoTensor` is the core representational object in the Symbo engine. It is a true n-dimensional symbolic tensor, meaning its entries hold SymPy expressions rather than the raw numeric values found in typical deep learning frameworks. This fundamental distinction allows it to encapsulate and operate on complex mathematical structures with perfect precision.

The primary function of a `NanoTensor` is to hold and operate on multivariate Taylor-expansion-based policy functions and model approximations. This polynomial is the mathematical realization of the "generative manifolds" mentioned in Symbo's core objectives, serving as a fully interpretable, local approximation of a system's behavior. Mathematically, each tensor encapsulates a multivariate Taylor expansion around a steady-state vector $\mathbf{x}$*:

f(x) \approx f(x^*) + \sum_i g_i (x_i - x_i^*) + \frac{1}{2}\sum_{i,j} g_{ij} (x_i - x_i^*)(x_j - x_j^*) + \cdots

This representation is powerful because its coefficients (e.g., *gi*, *gij*) are not merely numeric weights; they are the analytically derived partial derivatives of the system. This provides a direct, symbolic map of the system's sensitivities, where each coefficient has a clear, interpretable meaning. These coefficients can be solved for analytically using exact algebraic methods or learned from data.

Initializing a `NanoTensor` instance is straightforward and requires only a few key parameters:

- `shape`: A tuple defining the dimensions of the tensor (e.g., `(1,)` for a scalar model).
- `max_order`: The maximum order of the Taylor expansion (typically `1` or `2`).
- `base_vars`: A list of strings defining the names of the core state variables (e.g., `['k', 'a', 'eps']`).
- `name`: An optional human-readable identifier for the tensor.

The true power of this structure is that once instantiated, it provides access to a powerful suite of analytical tools immediately, without requiring any training.

## 4.0 The Symbolic Toolkit: Capabilities Without Training

A key strategic advantage of the Symbo engine is the powerful suite of deterministic, mathematically exact tools it provides "out-of-the-box." From the moment a `NanoTensor` is instantiated, it is not an empty container but a fully capable instrument for first-principles analysis. These untrained capabilities allow for precise exploration of mathematical systems before any data fitting or learning occurs.

## 4.1 Core Symbolic Manipulation

Symbo provides fundamental operators for analyzing and transforming symbolic expressions, primarily through differentiation and substitution. These tools are essential for studying the sensitivity and behavior of a model with analytical precision.

- **Symbolic Differentiation (`.diff()`):** Calculates the exact partial derivative of every expression within the tensor with respect to a specified variable. This is crucial for gradient-based analysis and understanding how a model's output changes in response to an infinitesimal change in an input.
- **Symbolic Substitution (`.subs()`):** Replaces symbols within the tensor's expressions with specific numeric or other symbolic values. This allows for model evaluation, scenario analysis, and simplification.

### Step-by-Step Guide

1. **Instantiate a `NanoTensor`:** Create a `NanoTensor` object to hold your expression.
2. **Populate with an Expression:** Define a symbolic expression using SymPy and place it into the tensor's data array.
3. **Differentiate:** Call the `.diff(wrt: sp.Symbol)` method on the tensor. This returns a *new* `NanoTensor` containing the exact partial derivative.
4. **Substitute:** Call the `.subs(sub_dict: Dict)` method, passing a dictionary that maps symbols to their replacement values. This returns a new, substituted `NanoTensor`.

## 4.2 Exact Algebraic Solving with Gröbner Bases

For complex systems of nonlinear polynomial equations, Symbo provides an exact, deterministic solver based on Gröbner bases. Unlike iterative numeric solvers that provide approximations and can be sensitive to initial guesses, this method finds every possible exact solution to the system algebraically. This capability is critical for domains like robotics or chemical engineering where systems are governed by non-negotiable physical constraints that must be solved for exactly, not approximated.

- The `.groebner_solve()` method takes a list of polynomial equations and a list of variables, returning all valid solutions.
- The `stream_groebner_basis()` method is designed for interoperability, computing a basis and yielding its polynomials as JSON objects for consumption by other processes.

### Step-by-Step Guide

1. **Define the System:** Create a list of polynomial equations as SymPy expressions, where each equation is implicitly set to zero.
2. **Define the Variables:** Create a list of the SymPy symbols you wish to solve for.
3. **Execute the Solver:** Call `.groebner_solve(poly_system, vars_to_solve)` on a `NanoTensor` instance.
4. **Interpret the Output:** The method returns a list of dictionaries. Each dictionary represents a single, valid solution, mapping variable names to their solved symbolic or numeric values.

## 4.3 Geometric Reasoning via Manifold Pathfinding

Symbo can translate algebraic expressions into geometric landscapes, enabling reasoning about trajectories and optimal paths. Using the A* pathfinding algorithm, it can identify the most efficient path—such as a low-cost or high-utility trajectory—across a symbolic energy or value surface.

The A* algorithm finds the optimal path by minimizing the function `f(n) = g(n) + h(n)`, where `g(n)` is the cumulative cost to reach node `n` and `h(n)` is a heuristic estimate of the cost from `n` to the goal. Symbo employs the Manhattan distance for this heuristic.

**Step-by-Step Guide**

1. **Generate a Symbolic Landscape:** Use the `.compute_grid()` method on a fitted `NanoTensor`. This evaluates the tensor's symbolic expression over a 2D grid of two specified variables, creating a cost surface `Z`.
2. **Define Start and Goal:** Specify the start and goal points as `(row, column)` indices into the `Z` grid.
3. **Find the Path:** Call `.find_path_on_grid(Z, start_idx, goal_idx, mode='min')`. The `mode` can be set to `'min'` to find a path through valleys (low cost) or `'max'` to find a path along ridges (high utility).
4. **Visualize the Result:** The returned path, a list of indices, can be passed to the `.plot_grid_with_path()` method to generate a contour plot of the landscape with the optimal trajectory overlaid.

## 4.4 Explainability via Derivative Trees

As a core component of its commitment to Explainable AI (XAI), Symbo can generate "derivative trees." These trees provide direct, symbolic insight into how changes in input variables mechanistically influence a model's output. It is a powerful tool for causal attribution and model introspection.

The `.deriv_tree(wrt_vars: List[str])` method computes the partial derivative of the model with respect to each variable in a given list, providing a clear map of influence.

**Step-by-Step Guide**

1. **Define the Model:** Begin with a `NanoTensor` that contains a defined or fitted symbolic model.
2. **Specify Influence Variables:** Create a list of variable names (strings) for which you want to assess influence (e.g., `['k', 'a', 'eps']`).
3. **Generate the Tree:** Call the `.deriv_tree()` method, passing the list of variable names.
4. **Analyze the Output:** The result is a dictionary that maps each variable name to its simplified partial derivative expression. This expression mathematically represents that variable's direct influence on the model's output.

These untrained capabilities form a robust foundation for symbolic analysis, but Symbo's true generative power is unlocked when the `NanoTensor` is made adaptive by training its symbolic coefficients.

# 5.0 Generative Modeling: Training and Fitting the `NanoTensor`

While the untrained `NanoTensor` is a powerful tool for static analysis, its capabilities can be extended to dynamic, adaptive modeling. In the Symbo paradigm, "training" is the process of determining the values of the symbolic coefficients within the Taylor expansion. This allows the `NanoTensor`'s internal model to fit observational data, satisfy complex system dynamics, or adhere to specified equilibrium conditions.

## 5.1 The `SymbolicTrainer` Class

The `SymbolicTrainer` is the primary interface for fitting `NanoTensor` models. It encapsulates several distinct methodologies for estimating the symbolic coefficients, accessible through a unified `.fit(data, method)` function. The choice of method depends on the nature of the problem and the available data.

- **`method='symbolic'`:** For exact coefficient fitting when a few precise data points are available. It formulates the problem as a system of polynomial equations and solves for the coefficients using Gröbner bases. This method requires a list of `(state_dictionary, target_value)` tuples.
- **`method='perturbation'`:** Designed for dynamic systems modeling, specifically for performing a full second-order perturbation analysis of a system's core equations around a steady state. This method requires a list containing two elements: the symbolic residual equation `R` and a dictionary of fixed model parameters.

- **`method='lsq'`:** The classical approach for coefficient fitting when you have noisy or abundant data. It uses ordinary least squares (LSQ) regression to find the best-fit values for the model's parameters. This method requires a list of `(state_dictionary, target_value)` tuples.

## 5.2 Step-by-Step Example: Perturbation Analysis

The `full_perturbation` routine is Symbo's most advanced analytical method, ideal for deriving policy functions in dynamic models (e.g., in macroeconomics). Here is a step-by-step guide to its use:

1. **Define the Model Residual:** Start with the core symbolic equation of the system, known as the residual `R`. This is an equation (like an Euler equation in economics) that must equal zero at the system's equilibrium.
2. **Instantiate the `NanoTensor` and `SymbolicTrainer`:** Create the necessary objects. For a second-order analysis, it is critical to specify `max_order=2` and provide the `base_vars` involved in the system.
3. **Execute the Fit:** Call the fit method with the appropriate arguments: `trainer.fit([R, params], method='perturbation')`. Note that for this specific method, the `data` argument is a list containing the symbolic residual `R` and the parameter dictionary `params` directly.
4. **Analyze the Fitted Coefficients:** The routine automatically computes the system's steady state, then symbolically differentiates `R` to solve for the first-order (`g_k`, `g_a`) and second-order (`g_k_k`, `g_k_a`) coefficients. These analytically derived values are stored in the `trainer.fitted_coeffs` dictionary.
5. **Make Predictions:** The fully fitted `NanoTensor` can now be used as a policy function. Use the `trainer.predict(state_point)` method, passing a dictionary of state variable values, to get the model's output.

## 5.3 Advanced Hybrid Training

For problems that require blending symbolic structure with neural network-based learning, the `HybridTrainer` class extends the core fitting capabilities into the neuro-symbolic domain.

- **`symbolic_regression()`:** This method provides a workflow similar to popular tools like PySR. It automatically evolves the complexity (i.e., the maximum order) of the Taylor polynomial to find the symbolic structure that best fits a given dataset.
- **`torch_fit()`:** This powerful method fully integrates a `NanoTensor` into a standard deep learning pipeline. It registers the symbolic coefficients as trainable parameters in a PyTorch module, allowing them to be optimized using any PyTorch optimizer (e.g., Adam) and loss function (e.g., MSE) within a standard training loop.

### 5.4 The `KnowledgeBase`: Storing and Querying Learned Facts

To manage the outputs of training runs, Symbo includes the `KnowledgeBase` class. This is a lightweight system for storing, organizing, and retrieving learned policy coefficients and other symbolic facts. It uses both a NetworkX graph for structural representation and `kanren` logic rules for declarative queries.

After a model is fitted, facts can be stored using `.add_fact('policy_name', 'coefficient_name', value)`. These facts can later be retrieved using `.query('policy_name', 'coefficient_name')`. The `KnowledgeBase` thus transforms the ephemeral results of a training run into a persistent, queryable set of symbolic facts, allowing the system to build a long-term, auditable memory of what it has learned about a system's dynamics.

## 6.0 Actionable Blueprints: From Theory to Application

This section serves as the "idea ignition" portion of the manual, providing concrete, actionable blueprints for using Symbo to solve challenging problems in different domains. These blueprints are derived directly from the demonstration cases included with the engine and show how to translate theory into practical application.

### 6.1 Blueprint 1: Macroeconomic Modeling (RBC Perturbation)

- **Problem:** Derive a policy function for a standard Real Business Cycle (RBC) model to understand how the next period's capital stock (`k'`) responds to changes in the current capital stock (`k`), technology level (`a`), and random shocks (`eps`).
- **Symbo Solution:** Use the `full_perturbation` method within the `SymbolicTrainer` to perform a second-order Taylor expansion of the model's core Euler equation around its deterministic steady state. This process analytically derives the coefficients of the policy function, providing a closed-form, interpretable approximation of the model's dynamics.
- **Implementation Steps:**
    1. Define all model symbols and parameters (`k`, `a`, `alpha`, `beta`, etc.) using SymPy.
    2. Construct the symbolic residual equation `R`, which represents the Euler equation that must hold at equilibrium.
    3. Instantiate a `NanoTensor` with `max_order=2` and specify the state variables in `base_vars=['k', 'a', 'eps', 'sig']` (where `'sig'` represents the standard deviation of the shock process).

4. Create a `SymbolicTrainer` instance and call the `.fit()` method using the `'perturbation'` option, passing the residual `R` and a dictionary of model parameters.
5. Inspect the `trainer.fitted_coeffs` dictionary to see the interpretable, analytically derived policy coefficients (e.g., `g_k`, `g_a`, `g_k_k`).
6. Use the trained model's `.predict()` or `.plot_contour()` methods to analyze the behavior of the resulting policy function.

### 6.2 Blueprint 2: Solving Algebraic Differential Equations (Kamke ADE)

- **Problem:** Find a parametric representation for a complex, implicit algebraic differential equation (ADE) where the derivative appears as a variable in a polynomial, such as `(y')^2 + 3y' - 2y - 3x = 0`.
- **Symbo Solution:** Leverage Symbo's high-level algebraic manipulation capabilities to solve this problem symbolically. The engine can compute the *resultant* of two polynomials to eliminate a variable, a technique that can be used to transform the implicit ADE into an explicit parametric form `(x(t), y(t))`.
- **Implementation Steps:**
    1. Define the implicit equation `F` using SymPy symbols for `x`, `y`, and the derivative `yp`.
    2. Instantiate a `NanoTensor` to access the algebraic toolkit.
    3. Call the `.parametrize_curve(F)` method on the tensor instance.
    4. The method will return the `x_param` and `y_param` expressions, which are the symbolic solutions `x(t)` and `y(t)` that describe the curve.

## 7.0 Deployment, Serialization, and Performance

For any computational engine to be effective in real-world applications, it must be interoperable, efficient, and deployable across different platforms. Symbo was designed from the ground up with high-speed I/O and cross-platform execution in mind, ensuring that its powerful symbolic models can be used in web applications, data pipelines, and distributed systems.

The engine's deployment and serialization capabilities are summarized below:

| Capability | Technology | Use Case |
|---|---|---|

|  |  |  |
|---|---|---|
| **WASM Execution** | `wasm_eval_expression`, `wasm_groebner_solve_json` | Enabling browser-side inference and interactive web applications without a Python backend. |
| **High-Speed I/O (Binary)** | `pyarrow`, `serialize_basis_arrow` | Efficiently streaming or storing large algebraic systems for consumption by other Arrow-compatible tools. |
| **High-Speed I/O (Compact)** | `msgpack`, `serialize_basis_msgpack` | Compact, fast serialization of algebraic models for transmission over networks or for lightweight storage. |

### Performance Considerations

The Symbo architecture intentionally prioritizes analytical purity over raw computational speed. While benchmarks show symbolic differentiation is an order of magnitude slower than numeric equivalents like NumPy, this is a calculated cost. The return is analytically exact gradients, which eliminates a significant source of numerical instability in complex optimization and hybrid neuro-symbolic training loops—a trade-off that is essential for model robustness.

Symbo's architecture ensures it can integrate smoothly into a larger ecosystem of specialized reasoning tools.

## 8.0 The Broader Vision: The Recursive AI Devs Ecosystem

Symbo is not a standalone project but a foundational component of a larger ecosystem of reasoning engines developed by Recursive AI Devs. Each model in this ecosystem is designed to handle a specific mode of abstract reasoning, with Symbo providing the core symbolic-algebraic capabilities.

The other models in the ecosystem include:

- **`FortArch`**: An encrypted container for secure model and data handling.
- **`Topo`**: An engine for performing topological reasoning.
- **`Chrono`**: A temporal propagation engine for modeling dynamic and sequential systems.
- **`Morpho`**: A transformational generative engine.

By combining the precision of algebra, the structure of topology, and the dynamics of temporal logic, this ecosystem aims to build a comprehensive framework for transparent and robust artificial intelligence. Symbo's unique value proposition within this vision is its role as an interpretable, generative, and mathematically grounded reasoning engine, capable of building adaptable models from first principles.

## 9.0 Appendix: Licensing and Contribution

The Symbo project is licensed under the **Apache License, Version 2.0**. This is a permissive open-source license that allows for broad use and modification.

The permissions granted by the Apache 2.0 license include:

- Commercial use
- Private modifications
- Distribution
- Patent protection
- Use in closed-source or open-source projects

### Authors and Attribution

The project authors are:

- **Damien Davison**
- **Michael Maillet**
- **Sacha Davison**

The foundational research was conducted by the **Recursive AI Devs** team.

### How to Contribute

Issues, feature requests, and pull requests are welcome. All contributions will be reviewed for mathematical integrity, symbolic correctness, and architectural compatibility. If you use the Symbo engine in your research or production systems, please cite and attribute the authors to acknowledge their work.