Okay, thank you for providing the expanded 3D model code and explaining the new aspects: 3D dynamics, kinetic/kinematic driving, torque matching calculations, increased logged parameters, different joint types, the ~1000 block limit constraint for your license, and your goal for versatility/adaptability and future optimization work.
Analyzing the packed 3DModel_packed.txt, I see the structure is similar to the previous version, containing several .m files consolidated by your packer script. The core scripts seem to be in the 3DModel/Scripts folder within that packed structure. I will focus on the suggestions for the *source files* that were packed, ignoring the explicit "backup" and "archive" content as requested.
Based on the new code and your goals, the previously identified areas for improvement are even *more* critical now, alongside new considerations for efficiency and the block count constraint.
Here are suggestions for improving your 3D model code, tailored to your needs:
1. Core Code Structure and Readability (More Important Now):

- **Refactor Scripts into Functions:** This remains the single most important step. Operating on the base workspace (cd, assignin, clear) becomes significantly harder to manage with increased complexity and more scripts interacting.
    - **Benefit:** Encapsulates logic, defines clear inputs/outputs, prevents unintended variable side effects, makes code reusable and independently testable. This is *essential* for building a versatile and adaptable model, and crucial groundwork for optimization.
    - **How:** Convert each logical .m file (like SCRIPT_TableGeneration_3D.m, SCRIPT_CHPandMPPCalculation_3D.m, the plotting scripts, and the new calculation scripts) into a function that takes necessary data (e.g., the simulation output object, tables) as input and returns the processed data or results as output. Your main MASTER_SCRIPT would then call these functions, managing the data flow explicitly.
- **Eliminate cd:** Relying on cd calls scattered throughout the code makes it brittle.
    - **Benefit:** Code becomes location-independent and more robust. You can run the master script from anywhere if your project folders are on the MATLAB path.
    - **How:** Use addpath at the start of your master script to add all necessary subfolders (Scripts, Scripts/Plots, etc.) to the MATLAB path. Then, call scripts/functions by name. For file operations (save, savefig, mkdir), use fullfile to construct paths relative to a project root obtained using fileparts(mfilename('fullpath')).
- **Improve SCRIPT_TableGeneration_3D.m:** You mentioned this script handles more parameters. While the loop structure for extracting data from out.logsout is fine, formalizing it as a function (as discussed previously) is key.
    - **Benefit:** Cleanly extracts data, easily adaptable if logged signals change (as long as names are consistent or mapping is handled), reusable.
    - **How:** Implement it as function Data = generateDataTable3D(simOutput).

2. Efficiency and Performance (Crucial for Optimization & Speed):

- **Vectorize Calculations:** While some vector calculations are already good (like those in

SCRIPT_TableGeneration_3D.m), review all scripts, especially any new calculation scripts (SCRIPT_RegressionCalcs.m, SCRIPT_PolynomialFitting.m, SCRIPT_KinematicTorqueGeneration.m) and the work/impulse scripts, for explicit for loops that process data element-by-element.
    - **Benefit:** MATLAB is highly optimized for vectorized operations. Replacing loops with functions like diff, sum, prod, element-wise operators (.*, ./, .^), and matrix operations can yield significant speedups, which is vital for optimization workflows.
    - **How:** Identify loops that can be replaced by single function calls or vectorized expressions acting on entire columns or matrices of data. For example, cumulative sums (cumsum) or cumulative integration (cumtrapz) are already vectorized and efficient.
- **Optimize ZTCF/ZVCF Data Generation Loops:** Running sim repeatedly in a loop is the likely bottleneck.
    - **Benefit:** Reduces overall simulation time.
    - **How:**
        - **Pre-allocate Tables:** Always pre-allocate tables (ZTCFTable, ZVCFTable) before entering loops where you add rows. ZVCFTable = table('Size', [numIterations, numVars], 'VariableTypes', varTypes, 'VariableNames', varNames); then ZVCFTable(i,:) = ...
        - **Parallel Simulation (parsim):** If you have the Parallel Computing Toolbox and your Simulink models are compatible (generally, if they run independently without file dependencies on each other during the loop), parsim is designed for exactly this type of workload (running many independent simulations). This could provide a major speed boost by utilizing multiple processor cores.
        - **Simulink.SimulationInput:** Instead of assignin inside the loop, create an array of Simulink.SimulationInput objects *before* the loop. Each object in the array can have different parameter overrides (in(i) = in(i).setVariable('ParamName', value);) and potentially initial states. You then pass this array to parsim or a single sim call (out = sim(in)). This is generally cleaner and required for parsim.

3. Simulink Model Management (Addressing Block Count & Parameters):

- **Monitor Block Count:** You are already doing this with SCRIPT_NonvirtualBlockCount.m. Keep this script handy.
    - **Benefit:** Ensures compliance with your license.
    - **How:** Run this script periodically as you develop to ensure you stay under the limit.
- **Virtual vs. Non-virtual Subsystems:** Group related logic into Subsystem blocks.
    - **Benefit:** Improves model organization and readability without increasing the non-virtual block count if they remain *virtual* subsystems.
    - **How:** Use Subsystem blocks generously for logical grouping. Only convert them to non-virtual (like Atomic Subsystems) if there's a specific simulation or code

generation reason that outweighs the block count penalty.

- **Parameter Management:** You're using Simulink.Parameter objects via the model workspace, which is a standard way to manage tunable parameters.
    - **Benefit:** Clear parameter definition, potentially tunable during simulation (if models are built that way).
    - **How:** Continue using Simulink.Parameter. As suggested under efficiency, explore passing sets of parameters via Simulink.SimulationInput when running simulations programmatically instead of repeated assignin calls.
- **Kinetic/Kinematic Switching:** How this switching is implemented in the models is key.
    - **Benefit:** Versatility in model usage.
    - **How:** Ensure the switching mechanism (e.g., using Variant Subsystems, Switch blocks driven by parameters, or separate model files) is clear and robust. If using separate model files (GolfSwing3D_KineticallyDriven.slx vs. GolfSwing3D_KinematicallyDriven.slx), your master script needs to manage which model is loaded and run. The current script seems to focus on the kinetically driven one first.

4. Specific Calculations (Regression/Polynomial Fitting):

- **Modularize Calculations:** The new regression and polynomial fitting steps are complex.
    - **Benefit:** Makes these calculations understandable, testable, and reusable. You can easily swap out different fitting methods or parameters.
    - **How:** Implement the logic in SCRIPT_RegressionCalcs.m, SCRIPT_PolynomialFitting.m, and SCRIPT_KinematicTorqueGeneration.m as functions that take the input data (e.g., position/velocity data from the kinematic model, torque data from the kinetic model) and return the fitted functions, coefficients, or generated torque inputs for the kinetic model.

5. Data Handling (Tables and Timetables):

- **Streamline Timetable Conversions:** The process of converting tables to timetables, re-timing, and converting back seems necessary for interpolation/resampling, but ensure the steps are minimal and clear.
    - **Benefit:** Reduces code complexity.
    - **How:** As suggested before, ensure your initial tables have a proper time vector (already seems to be the case). Use table2timetable specifying the 'Time' column. retime is the correct tool for resampling time series data.

In Summary:

Your 3D model introduces valuable complexity (kinetic/kinematic driving, torque matching). However, the existing code structure, inherited from the 2D version, is not well-suited for this increased complexity, especially with optimization goals and the block count constraint.

The most impactful changes you can make to your source .m files are:

1. **Convert Scripts to Functions:** This is fundamental for modularity, testability, and managing state.
2. **Eliminate cd and Base Workspace Reliance:** Use fullfile and pass data via function arguments.
3. **Vectorize and Pre-allocate:** Improve performance, especially in data processing and

table building loops.

4. **Leverage Simulink.SimulationInput and potentially parsim:** Optimize the repeated simulation runs for ZTCF/ZVCF generation.

Applying these improvements to your individual source files will make your project much more adaptable and prepare it well for future optimization work.