

Pseudo code: Cavity-Pressure Control System for IM

Prerequisites

- **CasADi toolbox**: Version 3.5.5
- **MATLAB**: Version R24b
 - **MATLAB Support for MinGW-w64 C/C++ Compiler**
 - **Statistics_and_Machine_Learning_Toolbox**
 - **Global_Optimization_Toolbox**
 - **Simulink**

1. Action State Recorder (`record_systemDynamics`)

Module: Data Acquisition / State Logging

Dependencies: System Bus Structures (`InputData_Bus`, `ControllerInput_Bus`, `Observer_Bus`)

1.1 Purpose

This function acts as a dedicated data logger that records the action-state transitions of the machine while the controller is active. It captures the control inputs applied to the plant and the corresponding measured states, storing them in pre-allocated arrays. When the controller deactivates, it stops recording and primes the buffers for a reset upon the next activation.

1.2 Interface Definition

- **[Input] `InputData_Bus`**: Structure containing the raw machine signals (`voltage_Controller`, `position_Screw`, `velocity_Screw`, `pressure_Screw`, `pressure_Cavity`).
- **[Input] `ControllerInput_Bus`**: Structure containing the `controlActive` boolean flag to dictate when recording should occur.
- **[Input] `Observer_Bus`**: Structure containing estimated/observed data; specifically uses index 5 to extract the current recorded time.
- **[Input] `bufferLength`**: An integer defining the maximum number of steps (memory allocation) the buffers can hold.
- **[Input] `dim`**: Structure containing system dimension definitions (used to determine the number of states to record).
- **[Output] `uOut`**: The buffer containing the recorded control actions (inputs).
- **[Output] `yOut`**: The buffer containing the recorded system states and time (outputs).
- **[Output] `nUsedEntries`**: An integer indicating the total number of valid data points successfully recorded in the current session.

1.3 State / Persistent Variables

The function maintains internal memory across execution cycles:

- `uBuffer`: A 1D array storing the sequence of control inputs.
- `xBuffer`: A 2D array storing the sequence of state vectors.
- `iterCounter`: An integer tracking the current insertion index in the buffers.

- **resetRequired**: A boolean flag indicating if a new recording session is starting, which triggers the buffers to be wiped.

1.4 Algorithmic Procedure

1. **Initialization and Reset Logic**: Check if the function is running for the first time or if a reset is required AND the controller has just been activated. If true: Allocate **uBuffer** with NaN values up to **bufferLength**; Allocate **xBuffer** with NaN values (rows = state dimensions, columns = **bufferLength**); Reset **iterCounter** to 1 and set **resetRequired** to false.
2. **Data Recording (Active Control Phase)**: Check if the controller is actively running (**controlActive == true**). If true, extract the current timestep data: Control input (**u_S**) from the voltage controller; States (**V_S**, **Q_S**, **p_S**, **p_C**) from the respective screw/cavity sensors and current time from the **Observer_Bus**. Prevent memory overflow by verifying **iterCounter** is less than or equal to **bufferLength**. Write the input into **uBuffer** at the current index. Write the combined state and time column vector into **xBuffer** at the current index. Increment **iterCounter** by 1.
3. **Idle Phase Handling**: If the controller is NOT active (**controlActive == false**): Stop all recording logic. Set **resetRequired** to true so that the buffers are wiped fresh the next time the controller turns on.
4. **Output Assignment**: Assign **uBuffer** to the **uOut** output port and assign **xBuffer** to the **yOut** output port. Calculate **nUsedEntries** by taking the current iteration count minus one, ensuring it is bounded between 1 and the maximum **bufferLength**.

2. Model Optimization

Module: Model Fitting & Parameter Optimization / Model Predictive Control

Dependencies: CasADi Framework, Target Model Object, **SQPModelFit**, **get_derivativeOfStates**, **mpcActiveSetSolver**, **x_dot_func**, **get_sigmoidBasis**

2.1 Function: **get_residualJacobian**

This function generates a compiled software routine (for MATLAB or Simulink) that calculates the residual errors and the Jacobian matrix for a dynamic system. It is designed to be used by a nonlinear optimization solver (e.g., Gauss-Newton or SQP) to fit model parameters to recorded real-world data.

Interface Definition

- **[Input] obj**: The model object containing system properties, state definitions, and ODEs.
- **[Input] varargin**: Optional arguments, primarily used to specify the build target ('simulink' or 'matlab').
- **[Output] fcn_handle**: The compiled function handle (returns empty if the target is Simulink).
- **[Output] tunableParamLength**: An integer representing the total number of tunable parameters in the model.
- **[Output] J_d**: A sparse constraint/indexing matrix mapping specific parameters (weights, C-constants).

Algorithmic Procedure

1. **Parameter Initialization**: Scan the model object to identify all parameters marked as tunable. Calculate **tunableParamLength** by summing the dimensions of these identified parameters. Determine the start

and end indices for mapping a flat parameter array back to structured variables.

2. **Symbolic Variable Declaration:** Initialize symbolic variables to represent the flat vector of `tunableParams`, the measured data (`recordedStates` and `recordedInputs`), observed state derivatives, and external boundary parameters.
3. **Forward Simulation:** Map the flat, symbolic `tunableParams` vector into a structured format compatible with the model. Evaluate the system's differential equations by passing the recorded data and structured parameters into the model's ODE function to yield the `simulatedDerivatives`.
4. **Constraint Matrix (`J_d`) Construction:** Initialize `J_d` as a matrix to track specific parameter indices. Assign a value of 1 to specific rows, including an identity block for `weights_K_C`, and specific single indices for C1, C2, and C3.
5. **Residual and Jacobian Computation:** Compute the `residual` vector by taking the difference between the actual measured derivatives and the `simulatedDerivatives`. Compute the `jacobianMatrix` by calculating the partial derivatives of the `residual` with respect to the `tunableParams`.
6. **Code Compilation and Export:** Define the final mathematical function grouping the symbolic inputs and computed outputs. Execute the code generator for Simulink or MATLAB.

2.2 Function: Online Model Parameter Optimization (`fcn`)

This function performs online parameter estimation for a dynamic system. It uses a Sequential Quadratic Programming (SQP) approach with an active-set solver to minimize the residual error between measured state derivatives and model-predicted state derivatives. It operates iteratively, storing state variables across executions.

Interface Definition

- **[Input] `V_S_init, paramIndexV_S_init`:** Initial screw volume and its corresponding parameter index.
- **[Input] `InputData_Bus, ControllerInput_Bus`:** Structures containing status flags and control booleans.
- **[Input] `initialModelParams`:** The baseline set of system parameters.
- **[Input] `U_stored, X_Stored`:** Buffers containing historical control inputs and measured states.
- **[Input] `nUsedEntries`:** Number of valid data points currently residing in the buffers.
- **[Input] `codegenTime`:** Boolean flag indicating if the system is in a code-generation phase.
- **[Input] `optim, dim`:** Structures containing optimization hyperparameters and system dimensions.
- **[Output] `modelParamsOut`:** The updated, optimized vector of system parameters.
- **[Output] `RMSE`:** A 4-element vector containing the Root Mean Square Error.
- **[Output] `bulkModulusOut`:** Array representing the calculated bulk modulus curve over a predefined volume range.

Persistent Variables

- `modelParams` & `savedModelParams`: Working set and frozen set of parameters.
- `isOptimized, RMSE_pers, bulkModulus`: Trackers for optimization state, error metrics, and bulk modulus curve.

Algorithmic Procedure

1. **Initialization:** On the first run, initialize all persistent variables. Update the initial screw antechamber volume parameter if in idle/reset phase. Freeze parameter updates if the controller is actively running.

2. **Data Preparation:** Extract measured states from historical buffer, calculate numerical time derivatives, and extract tunable parameters.
 3. **SQP Loop:** Loop through historical data points and call `SQPModelFit` to compute partial Jacobians and residuals, assembling them into global matrices. Compute Hessian approximation and check if positive definite. If rank deficient, add Levenberg-Marquardt damping. Solve for parameter delta step using `mpcActiveSetSolver` and update tunable parameters.
 4. **Post-Processing:** Predict new state derivatives, calculate RMSE, and calculate the bulk modulus curve using a sigmoid basis function. Export outputs.
-

3. Nonlinear Model Predictive Controller (NMPC)

Module: Controller Synthesis / Code Generation & Advanced Control

3.1 Function: `generate_NMPC`

This function sets up and compiles the symbolic optimization problem for an NMPC. It does not run the control loop; rather, it uses algorithmic differentiation (via CasADi) to define the system's cost function, dynamics, and constraints. It formulates a Sequential Quadratic Programming (SQP) subproblem using a Gauss-Newton approximation and generates highly optimized C-code or Simulink S-functions.

Algorithmic Procedure

1. **Symbolic Variable Initialization:** Initialize CasADi symbolic variables for states, inputs, slack variables, references, parameters, and weights. Pack these into a single flat vector `myParams` for the generated C-code.
2. **Cost Function:** Formulate the residual vector tracking error, actuator penalty, and overshoot penalty.
3. **Equality Constraints:** Load `modelParams` and formulate shooting equality constraints ensuring predictions match physical differential equations.
4. **Inequality Constraints:** Formulate input bounds, state constraints (using slack variables), and strictly positive slack variable constraints.
5. **Gauss-Newton Approximation:** Group optimization variables, compute the Jacobian, approximate the Hessian, and linearize equality/inequality constraints.
6. **Code Generation:** Execute the CasADi code generator for Simulink or MATLAB.

3.2 Function: Nonlinear Model Predictive Controller (`fcn`)

This function implements an NMPC utilizing an SQP strategy. It computes an optimal control trajectory to track a given reference while adhering to system constraints. To achieve real-time capability, it employs a "condensing" algorithm, which eliminates the state variables from the QP problem, drastically reducing matrix dimensions for the solver.

Algorithmic Procedure

1. **Initialization:** Replicate the currently observed state and handle inactive control states by outputting a safe guess.
2. **SQP Loop:** Expand the reference and define soft bounds for overshoot. Iterate for `nSQPSteps`.
 - **Full Problem Generation:** Pack parameters and call `get_SQPForNMPC` to obtain Hessian, gradient, and constraint matrices.

- **Condensing:** Partition matrices and use equality constraints to express future states entirely as a function of future inputs and the initial state. Substitute this relation back into the objective function and inequality constraints to create smaller condensed matrices, eliminating equality constraints entirely.
 - **Solving:** Regularize the condensed Hessian and call `mpcActiveSetSolver` to find optimal update steps. Reconstruct the state update step and update trajectories.
3. **Output Assignment:** Extract and export the first control action, handling fallbacks if the solver fails.
-

4. Part Weight Optimization Script

Module: Process Optimization / Setpoint Generation

Dependencies: `PartWeightModel`, `PressureReference`

4.1 Purpose

This top-level script manages the high-level cycle optimization for an injection molding process. It aims to maintain a consistent part weight despite material variations by adapting the cavity pressure setpoint from cycle to cycle. It utilizes a data-driven surrogate model (Bayesian Optimization / Gaussian Process Regression) to learn the mapping between cavity pressure and final part weight based on historical shot data.

4.2 Configuration & Inputs

- **Cycle Constants:** Pressure (`p_End`) and Time (`t_End`) at cycle end, and `t_Switch` for constant/linear reference switching.
- **partWeightReference:** Current target value for the part weight.
- **Obs (Historical Data):** Matrix containing previously applied cavity pressures, resulting part weights, and active references (exploratory shots generally excluded).

4.3 Outputs

- **myModel:** The fully trained `PartWeightModel` object with optimized hyperparameters.
- **nextPressureReference:** The newly calculated optimal cavity pressure setpoint for the next machine cycle.

4.4 Algorithmic Procedure

1. **Trajectory Reference Setup:** Define a baseline pressure trajectory reference function mapped to user-defined cycle constants.
2. **Data Extraction:** Extract arrays for Cavity Pressure and Part Weight, and instantiate the `PartWeightModel` object.
3. **Model Training:** Feed each historical data pair into the model using `add_observation`. Trigger `train_model` to optimize hyperparameters and output the results for verification.
4. **Next-Cycle Optimization:** Query the trained model using `calculate_newPcRef`. The model evaluates its surrogate function to find the pressure likely to yield the target weight. Store and display this new reference for the upcoming shot.

5. Complete Physical Plant Model (CompleteModel)

Module: Plant Modeling / Digital Twin

Dependencies: NonlinearModel, DriveModel, SigmoidBulkModulus, myFlowCoefficient

5.1 Purpose

This class acts as the central physics-based digital twin of the injection molding system. It aggregates subsystems (drive dynamics, flow rate, and bulk modulus) into a single continuous-time state-space representation. It provides the core Ordinary Differential Equations (ODEs) required by both the Extended Kalman Filter for state prediction and the NMPC for trajectory optimization.

5.2 System States and Inputs

- Inputs (u): Screw velocity control signal (u_S).
- States (x): Screw volume/position (V_S), volumetric flow rate induced by the screw velocity (Q_S), screw pressure (p_S), cavity pressure (p_C), and a tracking time variable (t).
- Outputs (y): Directly maps to the physical states (V_S , Q_S , p_S , p_C).

5.3 Algorithmic Procedure (ODE Evaluation)

The ODE function calculates the state derivatives ($x_{\dot{}}$) given the current state vector x , input u , and parameter struct p :

1. Drive Dynamics: Evaluates the `driveModel` to obtain the mechanical state derivatives:
 2. Flow Dynamics: Calculates the pressure gradient $\Delta p = p_S - p_C$. It then queries the `flowRateModel` to calculate the time-variant non-Newtonian flow coefficient C based on Δp and current time.
 3. Pressure Build-up (Screw): Calculates the rate of change in the screw antechamber using the constant bulk modulus K_S
 4. Pressure Build-up (Cavity): Calculates the displaced volume $\Delta V = V_{S0} - V_S$ and evaluates the `bulkModulusModel` (a series of sigmoid functions) to find the current cavity bulk modulus K_C . The cavity pressure derivative is then calculated using a fixed cavity volume V_C and a parameterized pressure sink:
 5. Output: Returns the aggregated column vector containing all calculated derivatives, alongside $t_{\dot{}} = 1$.
-

6. State Estimation

Module: Observer / Sensor Fusion

Dependencies: CompleteModel (via state-space and next-step S-functions)

6.1 Function: `kalmanFilterFunction`

This function implements an Extended Kalman Filter (EKF) to provide optimal, noise-filtered estimates of the system's unmeasurable or noisy states in real-time.

Interface Definition

- [Input] u, Y : Current control input and raw sensor measurements.
- [Input] dT : Controller sample time.
- [Input] Q_{kalman}, R_{kalman} : Tuning matrices for process noise covariance and measurement noise covariance.
- [Input] `controlActive`: Boolean flag; the filter resets if the controller is inactive.
- [Output] X_{est}, Y_{est} : The filtered state vector and estimated output.

Algorithmic Procedure

1. Initialization/Reset: If it is the first execution within a production cycle, if the controller is idle (`controlActive == false`), or if a mathematical fault occurs (NaNs detected in the covariance matrix), the filter resets. The initial state vector x_0 is populated directly from the current raw measurements Y , and the covariance matrix P_0 is zeroed out.
2. Linearization & Discretization: Extracts the continuous-time state-space matrices (A, B, C) by linearizing the nonlinear plant model around the previous operating point. The continuous system is then discretized into A_d based on the sample time dT using an exponential matrix approximation.
3. A-Priori Update (Prediction):
 - Predicts the next state x_{next} and expected output Y_{est} by advancing the nonlinear model via numerical integration.
 - Updates the prior covariance matrix: $P_0 = A_d * P_0 * A_d^T + Q$
4. A-Posteriori Update (Correction):
 - Computes the optimal Kalman gain K : $K = P_0 * C^T * \text{inv}(C * P_0 * C^T + R)$
 - Corrects the state prediction using the measurement residual: $x_{next} = x_{next} + K * (Y - Y_{est})$
 - Updates the posterior covariance matrix: $P_0 = (I - K * C) * P_0$
5. State Output: Saves x_{next} as the new operating point and outputs it as the estimated state X_{est} .

7. Detailed Implementation: Part Weight Surrogate Model (PartWeightModel)

Module: Process Optimization / Data-Driven Modeling

Dependencies: Gaussian Process Regression (GPR), Optimization Toolbox (`fmincon`, `fminunc`)

7.1 Purpose

This class expands upon the top-level part weight optimization script. It acts as the mathematical engine for the cycle-to-cycle controller, modeling the mapping from cavity pressure reference (p_{C_ref}) and cycle index to the final part weight using Gaussian Process Regression.

7.2 Model Architecture

- Prior Function: A global linear trend is assumed and fitted first to capture the baseline relationship between pressure and weight. $w_{prior} = \theta_1 + \theta_2 * p_{C_ref}$
- Surrogate (GPR) Model: Captures the nonlinear, localized deviations from the prior. It uses a custom combined kernel (Squared Exponential + Wiener) to account for time-varying disturbances (like machine wear or temperature drift over cycle indices).
- Input Bounds: Pressure queries are strictly bounded between 150 bar and 350 bar.

7.3 Core Methods

1. `add_observation`: Appends new cycle data (pressure reference, resulting part weight, cycle index) to internal historical arrays.
2. `train_model`:
 - Calls `fit_prior` to optimize the linear prior parameters (`theta_1`, `theta_2`) using unconstrained minimization (`fminunc`) to minimize the L2 norm of the prediction error.
 - Subtracts the optimized prior from the measured part weights to isolate the residual errors.
 - Trains the underlying GPR model on these residuals to update the kernel hyperparameters.
3. `predict_mu` & `predict_sigma`: Evaluates a given pressure reference by summing the deterministic output of the prior function and the probabilistic expected value (`mu`) from the trained GPR. It also computes the standard deviation (`sigma`) to gauge model uncertainty.
4. `calculate_newPcRef`: Calculates the optimal pressure setpoint for the next cycle. It defines a loss function as the L2 norm between the target part weight and the model's prediction (`mu`). It uses `fmincon` to find the pressure value within the 150-350 bar bounds that minimizes this loss.