`computeSOSFunnels`: A MATLAB Framework for Invariant Funnel Synthesis using Sum-of-Squares Optimization

Mohamed Khalid M Jaffar

2025

Abstract

This document serves as a tutorial and technical report for the `computeSOSFunnels` codebase, a framework for computing invariant funnels around nominal trajectories for nonlinear dynamical systems. The pipeline integrates trajectory optimization, feedback control design via time-varying Linear Quadratic Regulators (TVLQR), and Sum-of-Squares (SOS) programming to generate rigorous certificates of stability. These "funnels" guarantee that a system remains within a bounded region despite initial state deviations. Developed for MATLAB, the codebase utilizes YALMIP, IPOPT and SOSTOOLS, and requires an SDP solver such as MOSEK or SeDuMi.

1 Introduction

In robot autonomy and control systems, guaranteeing that a system stays close to a planned trajectory is critical. A *funnel* is a time-varying invariant region around a nominal trajectory such that any system state starting within the funnel inlet is guaranteed to remain inside the funnel for the entire finite time horizon, eventually exiting through the funnel outlet. This mathematical construct is particularly useful for *verified motion planning* of autonomous robots operating in uncertain environments.

This codebase implements a modular pipeline to compute these funnels:

1. **Nominal Trajectory Generation:** Computing a nominal trajectory between specified initial and final states and determining the associated feedforward control inputs.
2. **Feedback Controller Synthesis:** Designing a local feedback control law (TVLQR) to stabilize the system against deviations from the nominal trajectory.
3. **Dynamics Polynomialization:** Approximating the nonlinear error dynamics as polynomial functions (a prerequisite for SOS optimization).
4. **Funnel Synthesis:** Using Sum-of-Squares Programming (SOSP) and bilinear alternation to optimize funnel size, either by minimizing the volume of the forward reachable set or maximizing the volume of the backward reachable set.
5. **Verification:** Empirically validating through Monte Carlo rollouts using initial states sampled from the funnel inlet.

Supported Systems

This framework has been implemented and tested on three distinct nonlinear systems:

- **Unicycle:** (default on `main` branch) A non-holonomic mobile robot model.
- **Cart-Pole:** (available on `cartpole` branch) A classic underactuated benchmark system.
- **Quadrotor:** (available on `quadrotor` branch) A high-dimensional, 12-state aerial vehicle with complex nonlinear dynamics.

2 Prerequisites and Installation

To run this codebase, you must have MATLAB installed along with the following toolboxes and solvers:

- **YALMIP**: For general optimization interfaces
- **IPOPT**: Solver for the nonlinear trajectory optimization
- **SOSTOOLS**: For parsing Sum-of-Squares problems
- **Semidefinite Program Solvers**: One of the following is required:
  - MOSEK (Recommended for speed/stability)
  - SeDuMi (Free alternative)
  - SDPT3

2.1 Installation

To get started, download the source code by cloning the repository from GitHub. Open your terminal and run the following commands:

```
# Clone the repository
git clone https://github.com/khalid2696/computeSOSFunnels.git

# Navigate into the project directory
cd computeSOSFunnels
```

Once cloned, open MATLAB and navigate to this directory. The `main.m` script automatically handles adding the necessary subdirectories (`lib/`, `utils/`) to your MATLAB path.

3 Quick Start Tutorial

The entry point for the entire pipeline is `main.m`.

3.1 Running the Default Example (Unicycle)

1. Open MATLAB and navigate to the repository root.
2. Open `main.m`.
3. Run the script. It executes the five major steps in the following sequence.

```
% 1. Compute Nominal Trajectory
run("Step1_computeNominalTrajectory.m");

% 2. Synthesize TVLQR Feedback Control
run("Step2_FeedbackControllerSynthesis.m");

% 3. Polynomialize Deviation Dynamics
run("Step3_getDeviationDynamics.m");

```

```
% 4. Compute Invariant Funnels (SOS Program)
run("Step4_computeTimeSampledInvarianceCertificates.m");

% 5. (Optional) Verify with Monte Carlo Rollouts
run("./utils/empiricallyVerifyInvariance.m");
```

Code structure in `main.m`

4. Once complete, the script will generate plots showing the computed funnel projected onto 2D and 3D subspaces using `utils/plottingScript.m`.

3.2 Running Other Systems

The repository uses **git branches** to manage different dynamical systems. To run the cart-pole or quadrotor examples, simply checkout the corresponding branch before running MATLAB. For example,

```
# In your terminal
git fetch origin
git checkout quadrotor
```

Switching to other nonlinear systems, for instance quadrotor system

After switching branches, clear your MATLAB workspace (`clearvars; clear functions; close all;`) and run `main.m` again. The pipeline will automatically adapt to the dynamics of the checked-out system.

4 Detailed Workflow & Framework Architecture

This framework implements a systems theory-based pipeline for computing invariant funnels. The process is divided into four sequential stages: trajectory generation, feedback stabilization, deviation dynamics approximation, and Lyapunov funnel synthesis. Figure 1 illustrates the data flow and theoretical dependencies between these stages.

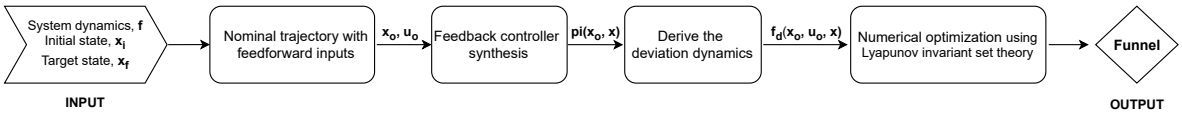

Figure 1: **Computation Pipeline Overview.** The workflow proceeds from finding a nominal trajectory to synthesizing a stabilizing feedback controller, deriving polynomial deviation dynamics, and finally computing the invariant funnel certificates using Sum-of-Squares (SOS) programming.

4.1 Step 1: Trajectory Optimization

File: `Step1_computeNominalTrajectory.m`
Library: `lib/getNominalTrajectory_using_DirectCollocation.m`

In the first step, we solve a nonlinear programming (NLP) problem to find a nominal trajectory $\mathbf{x}_{nom}(t)$ and feedforward inputs $\mathbf{u}_{nom}(t)$ that drive the system from a specified $\mathbf{x}_0$ to $\mathbf{x}_f$. It implements *direct collocation* with either trapezoidal or RK4 integration constraints.

Key Variables:

- `initialState`, `finalState`: Define boundary conditions.
- `maxTimeHorizon`: The maximum allowable finite horizon

- `N`: Number of time discretization steps (knot points)

4.2 Step 2: Feedback Controller Design

File: `Step2_FeedbackControllerSynthesis.m`
Library: `lib/compute_tvlqr_gains.m`

Open-loop trajectories are inherently sensitive to perturbations. To stabilize the system, we synthesize a time-varying Linear Quadratic Regulator (TVLQR), resulting in a closed-loop stable system. We linearize the dynamics around the nominal trajectory to obtain the time-varying system matrices $A(t)$ and $B(t)$, and solve the Riccati differential equation to find the optimal feedback gain $K(t)$ and cost-to-go matrix $P(t)$.

The resulting control law is:

$$\mathbf{u}(t) = \mathbf{u}_{nom}(t) - K(t)(\mathbf{x}(t) - \mathbf{x}_{nom}(t)) \tag{1}$$

4.3 Step 3: Deviation Dynamics (Importance of `xbar`)

File: `Step3_getDeviationDynamics.m`

We define the state deviation $\bar{\mathbf{x}} := \mathbf{x} - \mathbf{x}_{nom}$. The deviation dynamics of the closed-loop system are:

$$\dot{\bar{\mathbf{x}}} = f(\mathbf{x}_{nom} + \bar{\mathbf{x}}, \mathbf{u}_{nom} - K\bar{\mathbf{x}}) - \dot{\mathbf{x}}_{nom} \tag{2}$$

Because SOS programming requires the dynamics to be in polynomial form, this script performs a Taylor expansion (default degree 3) of these dynamics using MATLAB's Symbolic Toolbox.

Critical Implementation Note

The script converts standard MATLAB symbolic variables (`syms`) into polynomial variables (`pvar`). This conversion is *strictly necessary* because SOSTOOLS relies on `pvar` objects, and cannot process standard `sym` variables. All subsequent SOS steps rely on these `pvar` definitions in terms of `xbar`.

4.4 Step 4: Funnel Synthesis (Two-Step Alternation)

File: `Step4_computeTimeSampledInvarianceCertificates.m`

We seek a Lyapunov function candidate $V(\bar{\mathbf{x}}, t)$ and a scalar level set value $\rho(t)$ such that the set $\mathcal{E}(t) := \{\bar{\mathbf{x}} \mid V(\bar{\mathbf{x}}, t) \leq \rho(t)\}$ is invariant. The invariance condition (using the generalized S-procedure) requires finding Lagrange multipliers $\lambda(\bar{\mathbf{x}})$ such that:

$$-\dot{V}(\bar{\mathbf{x}}, t) + \dot{\rho}(t) - \lambda(\bar{\mathbf{x}})(\rho(t) - V(\bar{\mathbf{x}}, t)) \geq 0 \quad \forall \bar{\mathbf{x}}. \tag{3}$$

Since λ, V, and ρ are all decision variables, the SOS optimization problem is bilinear. The code implementation solves this via **Bilinear Alternation**:

1. **L-Step (Feasibility):** Fix ρ and $V(\bar{\mathbf{x}})$, then search for multipliers $\lambda(\bar{\mathbf{x}})$ at each time instance that satisfy the SOS constraints using `findPolynomialMultipliers()`.
2. **V-Step (Optimization):** Fix $\lambda(\bar{\mathbf{x}})$, then minimize the funnel volume (minimize ρ) subject to the SOS constraints using `findLyapFnAndLevelSetValues()`.
3. **Iterate:** Repeat the L-step and V-step until convergence.

An Important Remark

To initialize the alternation procedure, the user must specify an **initial guess** for the Lyapunov candidate function $V(\bar{\mathbf{x}}, t)$ and the level set value $\rho(t)$. We utilize the following guesses:

$$V_{\text{guess}}(\bar{\mathbf{x}}, t) = \bar{\mathbf{x}}^\top P(t)\bar{\mathbf{x}}, \tag{4}$$

$$\rho_{\text{guess}}(t) = \rho_0 e^{c(t-t_f)/(t_0-t_f)}. \tag{5}$$

Here, $P(t)$ represents the cost-to-go matrix obtained by solving the Riccati differential equation in Step 2, while ρ_0 and c are tunable hyperparameters. Refer to Section 6.2 for guidelines on tuning ρ_0 and c.

Hyper-parameters to Tune:

- `rhoInitialGuessConstant` (ρ_0): Initial scaling of the funnel. If the solver returns "Infeasible" immediately, try reducing this value (see Section 6.2).
- `rhoInitialGuessExpCoeff` (c): Controls the funnel expansion/contraction over time. An exponential guess is used as defined in Eq. (5).
- `maxIter`: Maximum number of iterations allowed for the bilinear alternation scheme.
- `multiplierPolyDeg`: Degree of the Lagrange multiplier polynomial (default 6).

4.5 Verification

File: `utils/empiricallyVerifyInvariance.m`

This script validates the theoretical Lyapunov guarantees through Monte Carlo analysis. It samples points from the computed inlet ellipsoid and simulates the full nonlinear dynamics forward in time. For a valid funnel, trajectories starting in the inlet should remain within the computed ellipsoids for the entire duration. This script also produces time plots of Lyapunov function values over the rollout horizon.

Technical Remark

Note that while the funnel is synthesized using the polynomial-approximated dynamics, this verification script uses the actual nonlinear system's equations of motion. In practice, the SOS-computed funnel serves as a good outer-approximation of the forward reachable invariant set of the original nonlinear system.

4.6 Data Flow Architecture

The pipeline utilizes the `precomputedData/` directory to pass time-sampled information between stages. This modular design allows users to re-run specific steps without restarting from scratch.

- `nominalTrajectory.mat`: Outputs from Step 1 (`time_instances`, `x_nom`, `u_nom`).
- `LQRGainsAndCostMatrices.mat`: Outputs from Step 2 (`K`, `P`).
- `deviationDynamics.mat`: Outputs from Step 3 (`xbar`, `deviationDynamics`).
- `setInvarianceCertificates.mat`: Final outputs from Step 4 (Funnel defined by `rho_scaling` and `candidateV`). A quadratic candidate $V(\bar{\mathbf{x}})$ is parametrized by a symmetric positive definite matrix M as $V(\bar{\mathbf{x}}) = \bar{\mathbf{x}}^\top M \bar{\mathbf{x}}$. M is saved as `ellipsoidMatrices`.

Note

All aforementioned files contain time-sampled data at each time step in `time_instances`. Note that `xbar` is a `pvar` object, while `deviationDynamics` is a polynomial symbolic expression defined in terms of `xbar`.

5 Customizing for New Systems

While the `main` branch focuses on the unicycle model, this repository includes full implementations for the cart-pole and quadrotor systems in separate branches. These serve as good templates for extending the framework to other dynamical systems.

5.1 Reference Implementations

Before implementing your own system, we highly recommend inspecting the architecture across different branches:

- **Unicycle:** `git checkout main` (3 States, 2 Inputs)
- **Cart-Pole:** `git checkout cartPole` (4 States, 1 Input)
- **Quadrotor:** `git checkout quadrotor` (12 States, 4 Inputs)

5.2 Step-by-Step Implementation Guide

To adapt the codebase for a custom robot (for example, a manipulator arm or a legged platform), follow these steps:

1. **Specify Robot Dynamics:** In `main.m`, update the `dynamicsFnHandle`.

```
% Update the system dynamics in main.m
dynamicsFnHandle = @(x, u) my_robot_dynamics(x, u, params);
```

2. **Update Trajectory Optimization (Step 1):** Modify `lib/getNominalTrajectory_using_DirectCollocation.m`. You must update the dynamics constraints inside the YALMIP setup to match your robot's equations of motion.

   Usage Tip

   Compare the `lib/` folder in the `main` branch with the `quadrotor` branch to see how higher-order state and input constraints are handled.

3. **Specify Cost Matrices for TVLQR Design (Step 2):** Define the Q, R, and P_f weight matrices. Ensure their dimensions correspond to the state and input dimensionality, n_x and n_u, respectively.

4. **Polynomialization (Step 3):** `Step3_getDeviationDynamics.m` is designed to automatically derive the polynomial deviation dynamics using a Taylor series expansion. The approximation order (default is 3) can be adjusted in the configuration.

5. **SOS Programming (Step 4):** Specify the hyperparameters, ρ_0 and c, to provide an initial guess for the SOSP-based funnel synthesis.

Usage Tip

See Sections 4.4 and 6 for tuning guidelines. You may also perform a hyperparameter sweep to identify the best initialization values (functionality included in `main.m`).

6. **Visualization:** Update `utils/plottingScript.m` to project the dimensions relevant to your new system. For example, if your system has 4 states, you might want to plot dimensions $[1, 2]$ (say position) or $[3, 4]$ (say velocities).

6 Troubleshooting

6.1 Issue: "Solver Failure" in Step 1.

Solution tip

- Ensure your initial and final states are physically reachable within `maxTimeHorizon`. If the problem is infeasible, try increasing the time horizon.
- Relax the state constraints and input saturations to verify if they are overly restrictive.
- Increase the number of knot points (`numTimeSteps`) in Step 1 to reduce numerical integration errors (inherent in the implemented trapezoidal or RK4 schemes).

6.2 Issue: "Infeasible Problem" in Step 4.

Solution tip

The SOS Program is failing to find an invariance certificate with the given initial guess.

- Decrease `rhoInitialGuessConstant` (ρ_0) in Step 4, ensuring it remains greater than 0.
- Decrease `rhoInitialGuessExpCoeff` (c) in Step 4. It is best to keep c non-negative so that the funnel volume shrinks over time. While the codebase can tolerate negative c values, this is generally not preferred.
- Check the condition number κ of the cost-to-go matrices P from the TVLQR solution in Step 2. Prevent numerical ill-conditioning by ensuring $\kappa < 10^3$ at all time steps.

Citation

If you find this codebase useful in your research, please cite the software as below:

```
M. K. M. Jaffar, "computeSOSFunnels: A MATLAB Framework for Invariant
Funnel Synthesis using Sum-of-Squares Optimization," v1.0.0, Zenodo, 2025.
DOI: 10.5281/zenodo.18047554
```

Alternatively, you may use the following BibTeX entry:

```
@software{jaffar_2025_computeSOSFunnels,
  author       = {M Jaffar, Mohamed Khalid},
  title        = {computeSOSFunnels: A MATLAB Framework for Invariant Funnel Synthesis
  using Sum-of-Squares Optimization},
  version      = {1.0.0},
  year         = {2025},
  publisher    = {Zenodo},
  doi          = {10.5281/zenodo.18047554},
  url          = {https://doi.org/10.5281/zenodo.18047554}
}
```