StarLander

A (WIP) Framework for Arbitrary Lumped-Mass Lander Simulation

Gagandeep Thapar

July 31, 2023

Table of Contents

Table of Contents

StarLander Background: Goals

- Excited by optimal control theory and implementation
- Approx. 1 month of free-time work
- Closed-loop control system simulation for non-convex soft-landing problem
  - Followed research out of UW ACL
  - Specifically 3DOF ...for now
- Goal was to develop a modular system to simulate control systems
  - Focus on trajectory-following vehicles
  - Further personal research into stochastic optimal control
  - Simple to abstract to generic systems

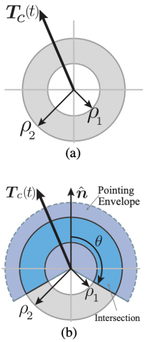

Figure: Non-Convex Thrust Bounds (Top) and Pointing Constraints (Bottom)[1]

StarLander Background: Scope and Technologies

- Independent research project
  - Written in C++17
- Used community libraries
  - Matplot++ (supports 3D plotting unlike Matplotlib-cpp)
  - Epigraph (C++ wrapper for ECOS, SOCP solver)
  - Eigen (Matrix, Vector, Linear Algebra)
- Git for version control, submoduling components
  - Each control block has its own repository
  - Can submodule/fork for other use cases
- CMake for compiling, dependency management

Table of Contents

Control System Architecture

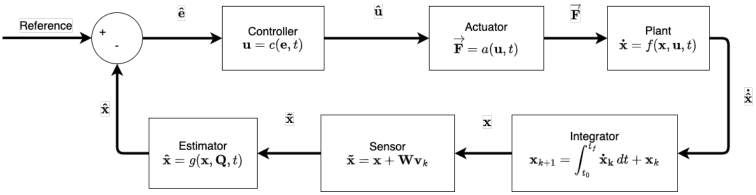

Figure: Closed Loop Feedback Control System Architecture

- Traditional control system architecture
- Each "block" is a unique C++ class with an abstract base class and several derived classes

Control System Explained: Sensor

- True state is unobservable
- Sensors measure empirical reality and return a (noisy) measurement
- `Sensor` Class allows selection of different noise profiles
- **Notice:** Noise simulation is technically interesting
  - White Noise (Gaussian) is unrealistic (inf. energy at high frequencies)
  - Pink/Colored Noise is more apt for real systems
    - Simulates random walk, bias

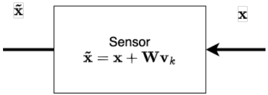

Figure: Sensor Block; Noisy State is Computed

Control System Explained: Estimator

- Measured state is too noisy; estimators reduce noise in a variety of ways based on measurements and inputs
- `State Estimator` Class allows for various styles
- **Kalman Filters** are interesting
  - Optimal linear estimators
  - Requires knowledge of system
    - Luckily we do!
  - Derived from "Spacecraft Dynamics and Control", DeRuiter

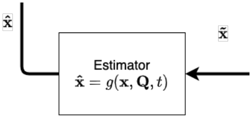

Figure: Estimator Block; Noise is Minimized

Control System Explained: Controller

- Controllers provide a signal based on difference on state and reference
- Several types (e.g., PID)
- **LQRs** are interesting
  - Optimal linear controllers
  - Requires knowledge of system
  - Combining an LQR with a Kalman Filter results in a **LQG**
- StarLander "cheats"
  - Inputs are calculated as part of trajectory generation for free

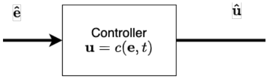

Figure: Controller Block; Control Signal to Actuators is Computed

Control System Explained: Plant

- Controllers (and Actuators together) provide an input e.g., a Force, Momentum, etc.
- Plant represents the system and contain system dynamic information
- Control input influences next state
- Plant class will contain state transition information to compute change in state
- `Plant` Class supports LTI systems currently
  - Non-Linear support is simple to add

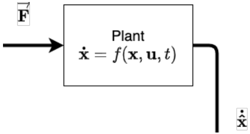

Figure: Plant Block; Derivative of State is Computed

Control System Explained: Integrator

- Plant computes derivative of state
- Next state computed as...

$$\mathbf{x}_{k+1} = \int_{\Delta t} \dot{\mathbf{x}}_k \mathrm{d}t + \mathbf{x}_k$$

- `ODESovler` Class allows selection to trial different methods
- ForwardEuler is unstable:

$$\mathbf{x}_{k+1} = \dot{\mathbf{x}}_k \Delta t + \mathbf{x}_k$$

- **Runge-Kutta-4/5** is relatively simple and stable for non-stiff systems

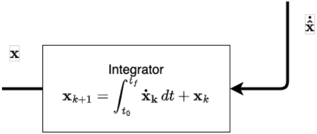

Figure: Integrator Block; New State is Computed

Control System Explained: Reference Trajectory

- Control system aligns Plant state with some reference
- Landing trajectories in the context of `StarLander`
- `TrajectoryGenerator` Class opens interface for computing path given initial point and returning closest point along path
- `SplineTrajectory` performs soft-landing kinematically but ignores constraints
- `GFOLD` performs soft-landing and abides by constraints

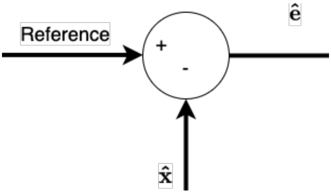

Figure: Reference Block; Deviation in State is Computed

Table of Contents

Software Implementation: System Overview

- Abstract classes provide public interfaces
  - Computation methods are overridden in derived class
- Instantiate each block:
  - $\begin{cases} \in [1, n] & \texttt{if Sensor} \\ 1 & \texttt{otherwise} \end{cases}$
- Initialize `ControlSystem` Class
  - System wrapper to control signal flow
  - Stores bus data at each time step
- Call `simulate` method

```
while ((t <= time) && (t_step > SIM_CONST::SMALL)) {
    // apply control
    actuation = reference->world.PLANET_G;
    if (col_idx < reference->input().cols()) {
      actuation = reference->input().col(col_idx) + reference->world.PLANET_G;
    }
    // update state (use true state for propagation)
    true_state = plant->update(truth_bus.col(col_idx - 1), actuation, t);
    truth_bus.col(col_idx) = true_state;
    // measure state vector
    for (Sensor *sensor : sensor_set) {
      int state_id = (int)sensor->sensor_id / SENSOR_IDENT - 1;
      meas_state[state_id] = sensor->sample(true_state[state_id]);
    }
    measurement_bus.col(col_idx) = meas_state;
    // estimate state
    est_state = estimator->estimate(meas_state, actuation);
    estimation_bus.col(col_idx) = est_state;
    // get reference signal
    ref_state = reference->get_reference(est_state);
    reference_bus.col(col_idx) = ref_state;
    // update time
    time_bus[col_idx] = t;
    // check time bounds
    t_step = (t + t_step > time) ? (time - t) : (t_step);
    t += t_step;
    col_idx++;
  }
```

Figure: `ControlSystem simulate` Method to Run Control System Over Timespan

Software Implementation: Sensor

- Derived `IdealSensor`, `WhiteSensor`, and `PinkSensor` from abstract Sensor Class
  - Abstract class "samples" reality
  - Derived class overrides how noise is generated
- Pink Noise generation transcribed from MATLAB IMU model process

$$\begin{aligned}\beta_k &= \frac{1}{2}\beta_{1,k-1} + Bw_k \\ &+ \beta_{2,k-1} + w_k\left(\frac{RW}{\sqrt{\frac{1}{2}T_s}}\right)\end{aligned}$$

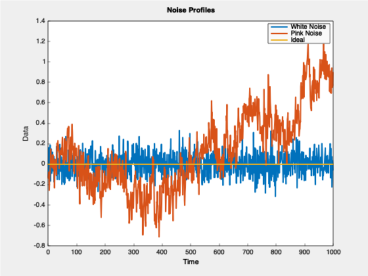

Figure: Ideal, White, and Pink Noise Profiles

Software Implementation: Estimator

- Derived `NULLEstimator` and (LTI) `KalmanFilter` from abstract `StateEstimator` Class
  - `NULLEstimator` does not estimate (returns measurement)
  - `KalmanFilter` employs Kalman Filter with provided state transition matrices s/t $\mathbf{x}_{k+1} = \mathbf{F}\mathbf{x}_k + \mathbf{G}\mathbf{u}_k$ and $\mathbf{y}_k = \mathbf{H}\mathbf{x}_k + \mathbf{M}\mathbf{v}_k$
- Can assume $\mathbf{F} = \mathbf{A}\Delta t + \mathbf{I}$, $\mathbf{G} = \mathbf{B}\Delta t$ but will lead to error (see *Results*, *Appendix* section)

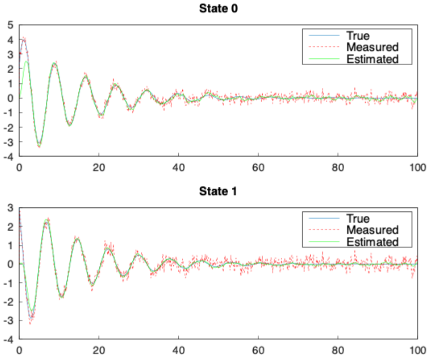

Figure: Measured VS KF Estimated State for Mass-Spring-Dampener

Software Implementation: Plant

- Derived (LTI) `LinearSystem` from abstract `Plant` Class
  - `Plant` exposes interface for propagating state using integrator
  - `LinearSystem` accepts constant state transition matrices
  - `NonLinearSystem` is simple to implement (see `Integrator`)
- `Plant` classes store key information for simulation
  - Number of States
  - Number of Inputs

```cpp
// enum to specify ode type
enum ODE_TYPE { FE, RKF45 };

class Plant {

public:
  // returns x_dot
  virtual Eigen::VectorXd update(Eigen::VectorXd state, Eigen::VectorXd input,
                                 double time) = 0;

protected:
  void init_ode_system(ODE_TYPE ode_type, int num_states, double dt);

  // applies xdot = f(x, u, t)
  virtual Eigen::VectorXd apply_transition(double time, Eigen::VectorXd state,
                                           Eigen::VectorXd input) = 0;

protected:
  ODESOLVER::ODESolver *m_ode;

public:
  int num_states;
  int num_inputs;
  double t_step;
};
```

Figure: Implementation of `Plant` Abstract Base Class

Software Implementation: Integrator

- Derived `ForwardEuler` and `RKF45` from abstract `ODESolver`
- `ForwardEuler` is unstable but simple
- `RKF45` is stable (for most systems)
- User supplied ODE, tolerance, time-span
  - Heavy influence from MATLAB ode45 or Python `solve_ivp`

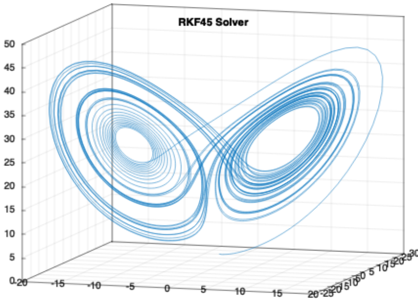

Figure: Lorenz Attractor Integrated via `RKF45` Solver

Software Implementation: Reference Trajectory

- Derived `SplineTrajectory` and `GFOLD` from abstract `TrajectoryGenerator` Class
  - `SplineTrajectory` will solve kinematically (**0** at end) but does not consider constraints
  - `GFOLD` uses Convex Optimization to solve constrained problem
- `TrajectoryGenerator` exposes interfaces:
  - Compute trajectory
  - Determine optimal reference state

```cpp
// abstract base class
class TrajectoryGenerator {
public:
  // main methods to override
  Eigen::VectorXd get_reference(Eigen::VectorXd state);

  const Eigen::MatrixXd &state() const;
  const Eigen::MatrixXd &input() const;
  const Eigen::MatrixXd &mass() const;

protected:
  virtual void compute_trajectory(Eigen::VectorXd position,
                                  Eigen::VectorXd velocity) = 0;

public:
  lander_data lander;
  trajectory_constraint traj_init;
  env_data world;
  double time_of_flight;
  const std::string name = "Abstract Trajectory Generator";

protected:
  // member variables
  Eigen::MatrixXd m_trajectory, m_input, m_mass;
  Eigen::Vector3d m_rF{{0.0, 0.0, 0.0}};
  Eigen::Vector3d m_vF{{0.0, 0.0, 0.0}};
};
```

Figure: Implementation of Trajectory Abstract Base Class

Software Implementation: Spline Trajectory

- Cubic Spline with initial conditions and final conditions set
- Assume linear acceleration profile
  - Integrate for velocity, position
  - Use boundaries for coefficients
- Generalize to 3-dimensions
- See Appendix

$$j = A \tag{1}$$

$$a = At + B \tag{2}$$

$$v = \frac{1}{2}At^2 + Bt + C \tag{3}$$

$$x = \frac{1}{6}At^3 + \frac{1}{2}Bt^2 + Ct + D \tag{4}$$

$$A = \frac{6v}{t^2} + \frac{12r}{t^3} \tag{5}$$

$$B = -\frac{v}{t} - \frac{1}{2}At \tag{6}$$

$$C = v_0 \tag{7}$$

$$D = r_0 \tag{8}$$

Software Implementation: GFOLD

- "Guidance for Fuel Optimal Large Diverts"
- First researched by Acikmese (UW), Blackmore (SpaceX)
- Pose constraints at each time step, use SOCP Solver

$$\mathbf{x}[0:3,0],[4:6,0] = \mathbf{r}_0, \mathbf{v}_0 \quad (9)$$

$$\mathbf{x}[0:3, ToF],[4:6, ToF] = \mathbf{0}, \mathbf{0} \quad (10)$$

$$\texttt{while t > ToF:} \quad (11)$$

$$\mathbf{x}_{k+1} = (\mathbf{A}\mathbf{x}_k + \mathbf{B}(\mathbf{g} + \mathbf{u_k}))\Delta t + \mathbf{x}_k \quad (12)$$

$$\mathbf{z}_{k+1} = \mathbf{z}_k - \alpha\sigma_k \quad (13)$$

$$\mathbf{u}_k^T \mathbf{n}_z \geq \cos\theta_{\texttt{max}} \quad (14)$$

$$\mathbf{r}_k^T \mathbf{n}_z \geq \sigma_k \cos\gamma \quad (15)$$

$$||\mathbf{u}_k|| \leq \sigma_k ... \quad (16)$$

Software Implementation: Compiling and Linking

- Learned basics of CMake
- Each component has its own CMake
- Flags set at compile time for build targets:
  - Executable (compiling examples)
  - Libraries (linking to `StarLander`)
- Links community libraries

```cmake
cmake_minimum_required(VERSION 3.26)
project(ControlSystem)

set (CMAKE_CXX_STANDARD 17)
set (CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/..)
set (CMAKE_EXPORT_COMPILE_COMMANDS 1)

# plotting libary
find_package(Matplot++ REQUIRED)

# source epigraph (submodule)
set(ENABLE_ECOS TRUE)

# ODE library
set(BUILD_ODE_LIB ON CACHE BOOL "Build as a library" FORCE)
add_subdirectory(ODESolver)

# Sensor library
set(BUILD_SENS_LIB ON CACHE BOOL "Build as a library" FORCE)
add_subdirectory(Sensor)

# Sensor library
set(BUILD_EST_LIB ON CACHE BOOL "Build as a library" FORCE)
add_subdirectory(StateEstimator)

# Trajectory library
set(BUILD_TRAJ_LIB ON CACHE BOOL "Build as a library" FORCE)
add_subdirectory(TrajectoryGenerator)

add_executable(example.out landing_simulation.cpp plant.cpp control_system.cpp)
target_link_libraries(example.out Matplot++::matplot)
target_link_libraries(example.out odesolver_lib)
target_link_libraries(example.out sensor_lib)
target_link_libraries(example.out stateestimator_lib)
target_link_libraries(example.out trajectorygenerator_lib)
```

Figure: CMake File for Top Level `StarLander`

Software Implementation: All Together Now

- `ControlSystem` initialized with proper components
- `Sensor` "measures" reality with noise profile
- `Estimator` attempts to remove all noise
- `Controller` compares against `Trajectory`
- `Plant` reacts to input
- `Integrator` propagates to next state
- `ControlSystem` provides plotting interface

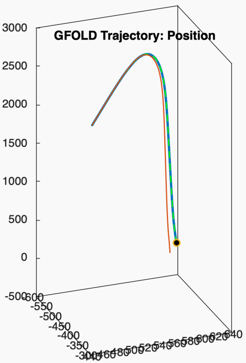

Figure: GFOLD Trajectory Example

Table of Contents

Results: Parameters

Case A

- 6 `IdealSensors`
- `NULLEstimator`
- `SplineTrajectory`
- Trajectory-based Input
- RK-45 System Integrator

Case B

- 6 `PinkSensors`
- `KalmanFilter`
- `GFOLD` Trajectory
- Trajectory-based Input
  - To-be LQR
- RK-45 System Integrator

$$\mathbf{x}_0 = [-330m, 450m, 2400m, -30m/s, 20m/s, 40m/s]^T$$

Case A: Spline Trajectory

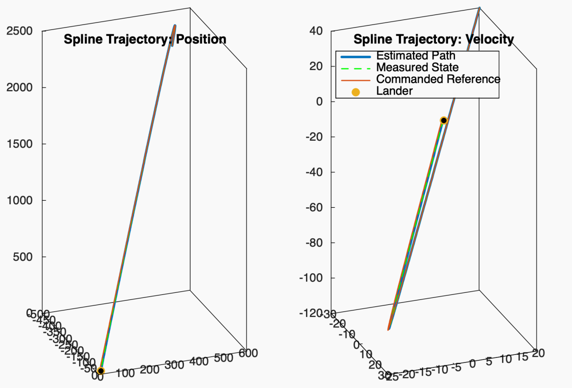

Figure: Spline-based Trajectory with Ideal State Knowledge; $\mathbf{x}_F = [0m, 0m, 0m, 0m/s, 0m/s, 0m/s]^T$

Case B: GFOLD Trajectory

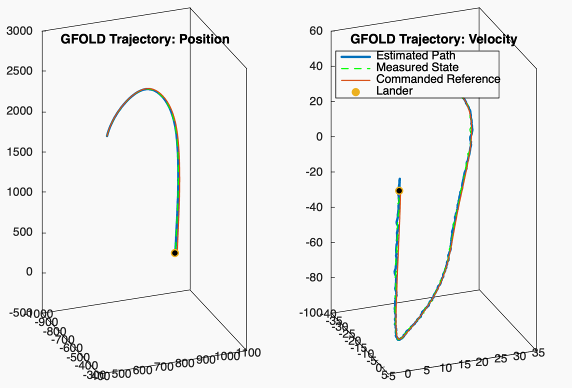

Figure: GFOLD-based Trajectory with Estimated State Knowledge;
$\mathbf{x}_F = [-979.59, 1032.24, -5.96e-5, -7.48e-5, 7.32e-5, 1.20e-3]^T$

Case B: GFOLD Trajectory

See GIF

Table of Contents

Conclusion

- Landing rockets is hard!
- Simulation architecture is important!
  - Post-Processing
  - Debugging
- Currently Open-Loop Control
  - LQR Coming Soon
- Important not to "optimize early"
  - Focus on the project and not the broader application
- `StarLander` is a huge Work in Progress
  - Strong foundation will make the future easier
  - Fix bugs and add features before it gets complicated

Future Work

- Implement LQR Controller
- Fix GFOLD Exit Codes
  - Likely a conflicting constraint
- Implement multi-threading
- Investigate landing under stochastic conditions
- Improve memory usage
  - References and pointers instead of copies where possible
- Add additional block options for wider support
- Improve animation generation pipeline

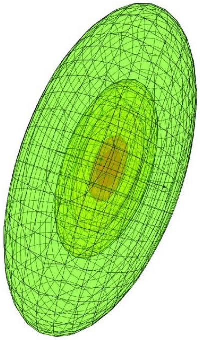

Figure: Probability Ellipsoid of Lumped Mass[2]

References

1. "Lossless Convexification of Nonconvex Control Bound and Pointing Constraints of the Soft Landing Optimal Control Problem", Acikmese et al. (2013)
2. "Calculating collision probability for long-term satellite encounters through the reachable domain method", Wen et al. (2022)

Questions?

Table of Contents

Appendix A: Lossless Convexification

1. "Lossless Convexification of Nonconvex Control Bound and Pointing Constraints of the Soft Landing Optimal Control Problem", Acikmese et al. (2013)
2. "In Search of the Material Composition of Refuse-Derived Fuels by Means of Data Reconciliation and Graphical Representation", Schwarzback et al. (2023)

Appendix B: Kalman Filter State Transition Matrices

$$
\begin{aligned}
\mathbf{x}_{k+1} &= \mathbf{F}\mathbf{x}_k + \mathbf{G}\mathbf{u}_k + \mathbf{L}\mathbf{w}_k \quad ; \quad w \in \mathcal{N}(0, \mathbf{Q}^2) && (17)\\
\dot{\mathbf{x}}_k &= \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k && (18)\\
\text{NOTE} &: \text{Using Forward Euler...} && (19)\\
\mathbf{x}_{k+1} &= \dot{\mathbf{x}}_k \Delta t + \mathbf{x}_k && (20)\\
\mathbf{x}_{k+1} &= (\mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k)\,\Delta t + \mathbf{x}_k && (21)\\
\mathbf{x}_{k+1} &= (\mathbf{A}\Delta t + \mathbf{I})\,\mathbf{x}_k + \mathbf{B}\Delta t\mathbf{u}_k && (22)\\
\mathbf{F} &= \mathbf{A}\Delta t + \mathbf{I} && (23)\\
\mathbf{G} &= \mathbf{B}\Delta t && (24)
\end{aligned}
$$

Appendix C.1: Spline Trajectory Coefficients

$$j = A \tag{25}$$

$$a = At + B \tag{26}$$

$$v = \frac{1}{2}At^2 + Bt + C \tag{27}$$

$$x = \frac{1}{6}At^3 + \frac{1}{2}Bt^2 + Ct + D \tag{28}$$

$$x(0) = D = r_0 \tag{29}$$

$$v(0) = C = v_0 \tag{30}$$

Appendix C.2: Spline Trajectory Coefficients

$$\begin{aligned}
\frac{v_0 + v_f}{2} &= \frac{A}{4}\left(t_0^2 + t_f^2\right) + \frac{B}{2}\left(t_0 + t_f\right) + C && (31)\\
\frac{x_f - x_0}{t_f - t_0} &= \frac{A}{6}\left(t_0^2 + t_0 t_f + t_f^2\right) + \frac{B}{2}\left(t_0 + t_f\right) + C && (32)\\
A &= 6\frac{v_0 + v_f}{(t_f - t_0)^2} - 12\frac{r_f - r_0}{(t_f - t_0)^3} && (33)\\
B &= \frac{v_f - v_0}{t_f - t_0} - \frac{1}{2}At && (34)\\
v_f = r_f = t_0 &= 0 && (35)\\
A &= \frac{6v_0}{t^2} + \frac{12r_0}{t^3} && (36)\\
B &= -\frac{v_0}{t} - \frac{1}{2}At && (37)
\end{aligned}$$