DRAKE

A PLANNING, CONTROL,
AND ANALYSIS TOOLBOX FOR
NONLINEAR DYNAMICAL SYSTEMS

Dynamical Systems & Automotive Components

MIT/TRI Project: Interlock for Self Driving Cars

25 April 2018
Soonho Kong

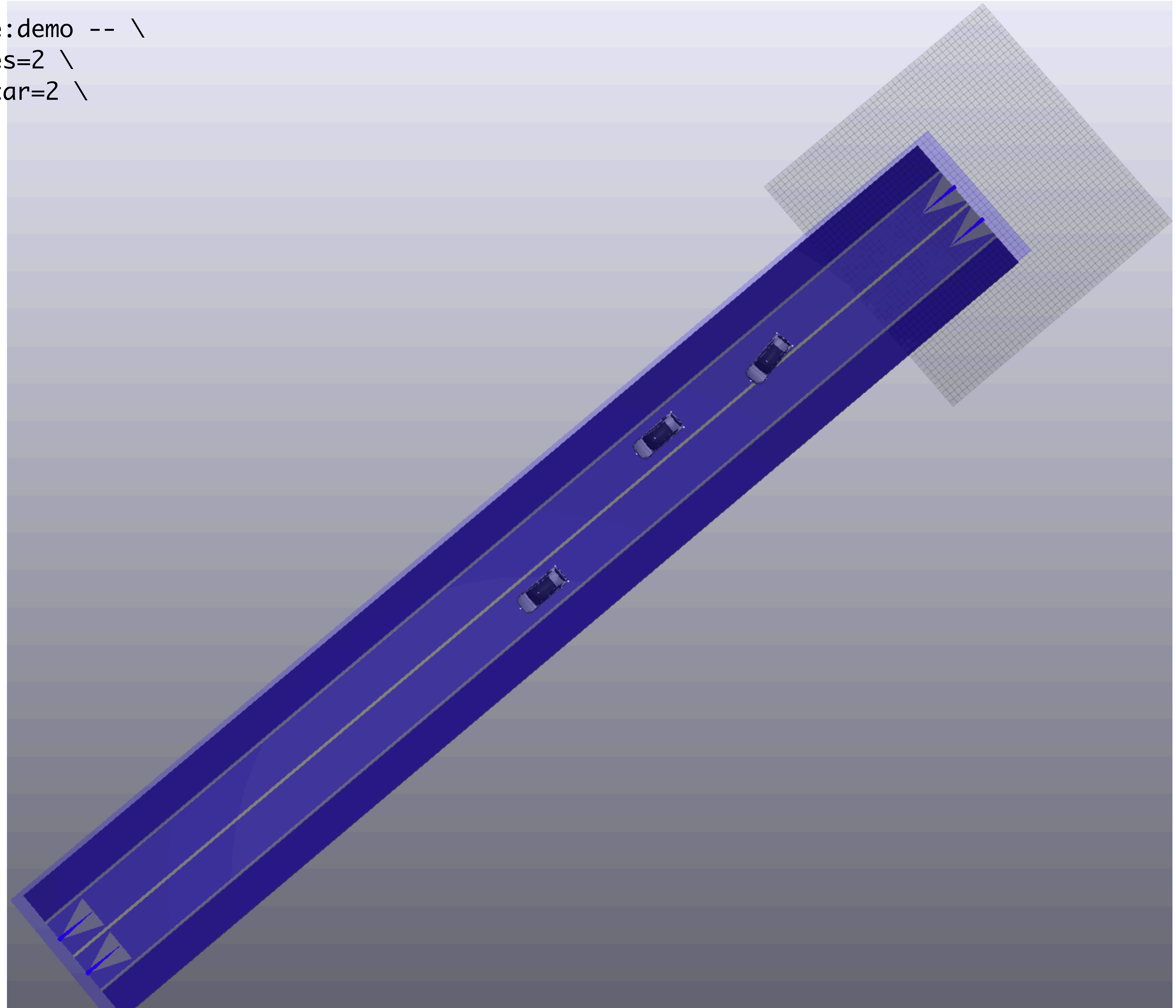
TOYOTA
RESEARCH INSTITUTE

Caution:

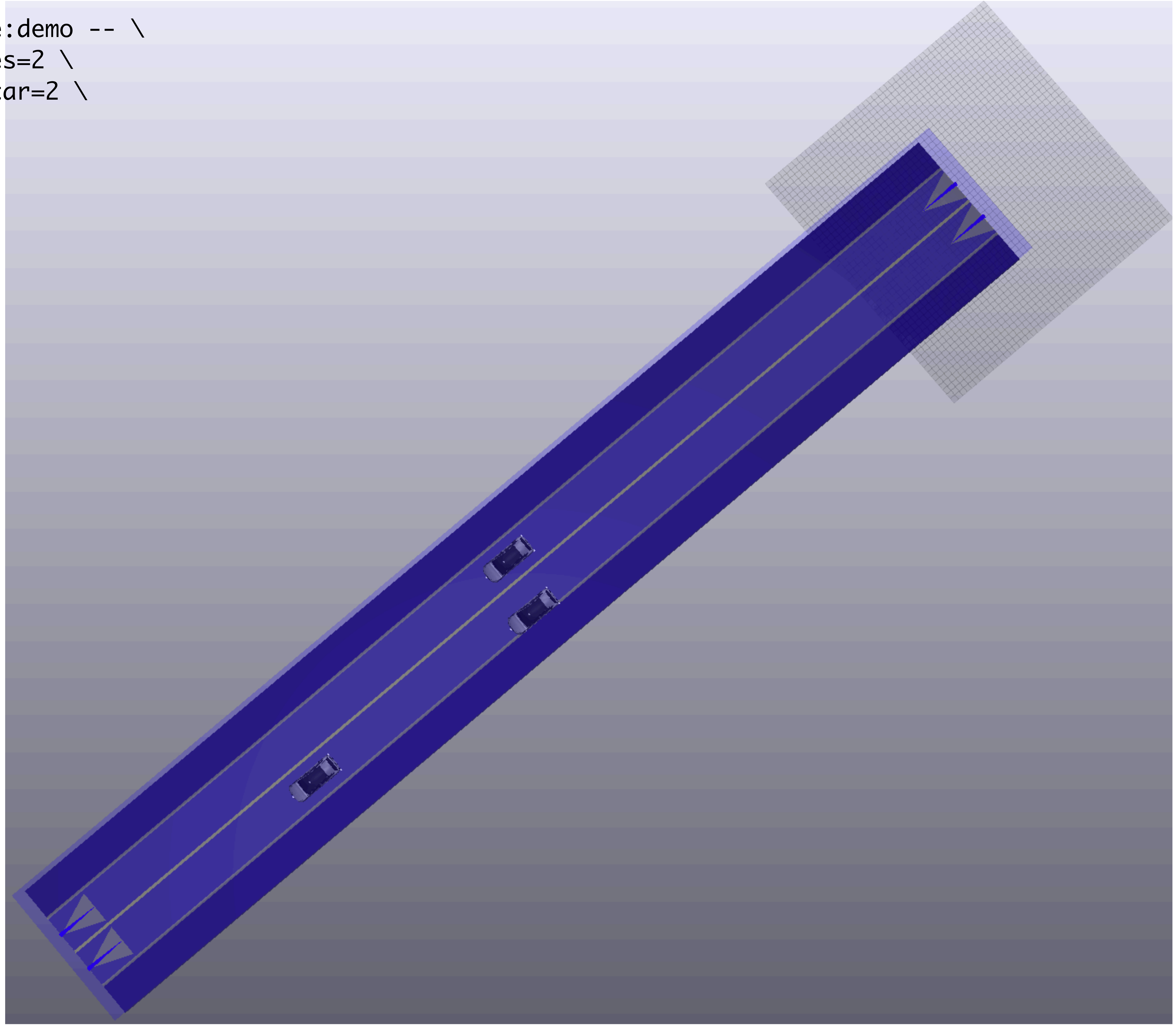
This presentation is based on the Drake version as of 25 April 2018
and it could be outdated from the current master.

Automotive Demo

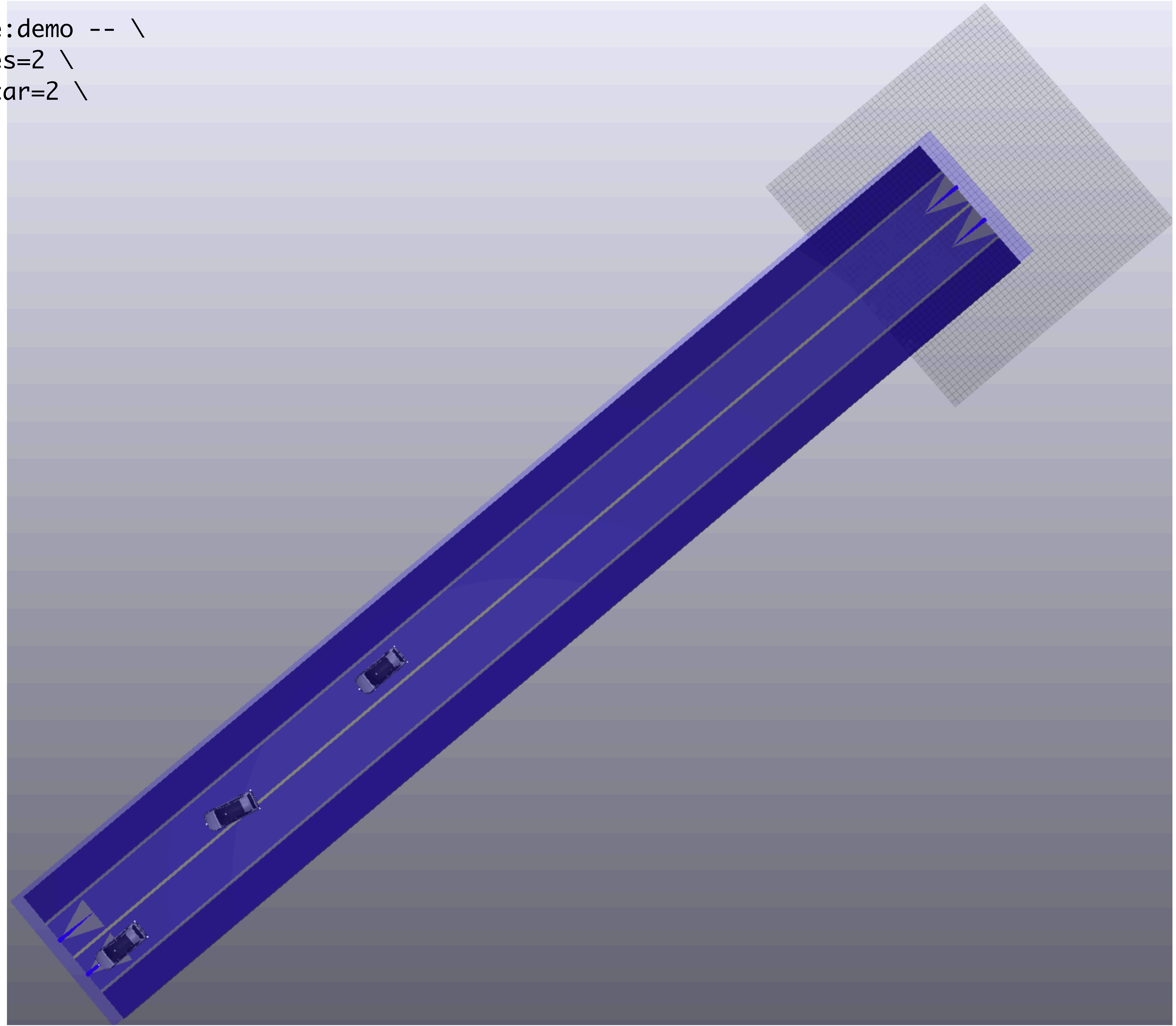
```
$ bazel run automotive:demo -- \
    --num_dragway_lanes=2 \
    --num_trajectory_car=2 \
    --num_mobil_car=1
```



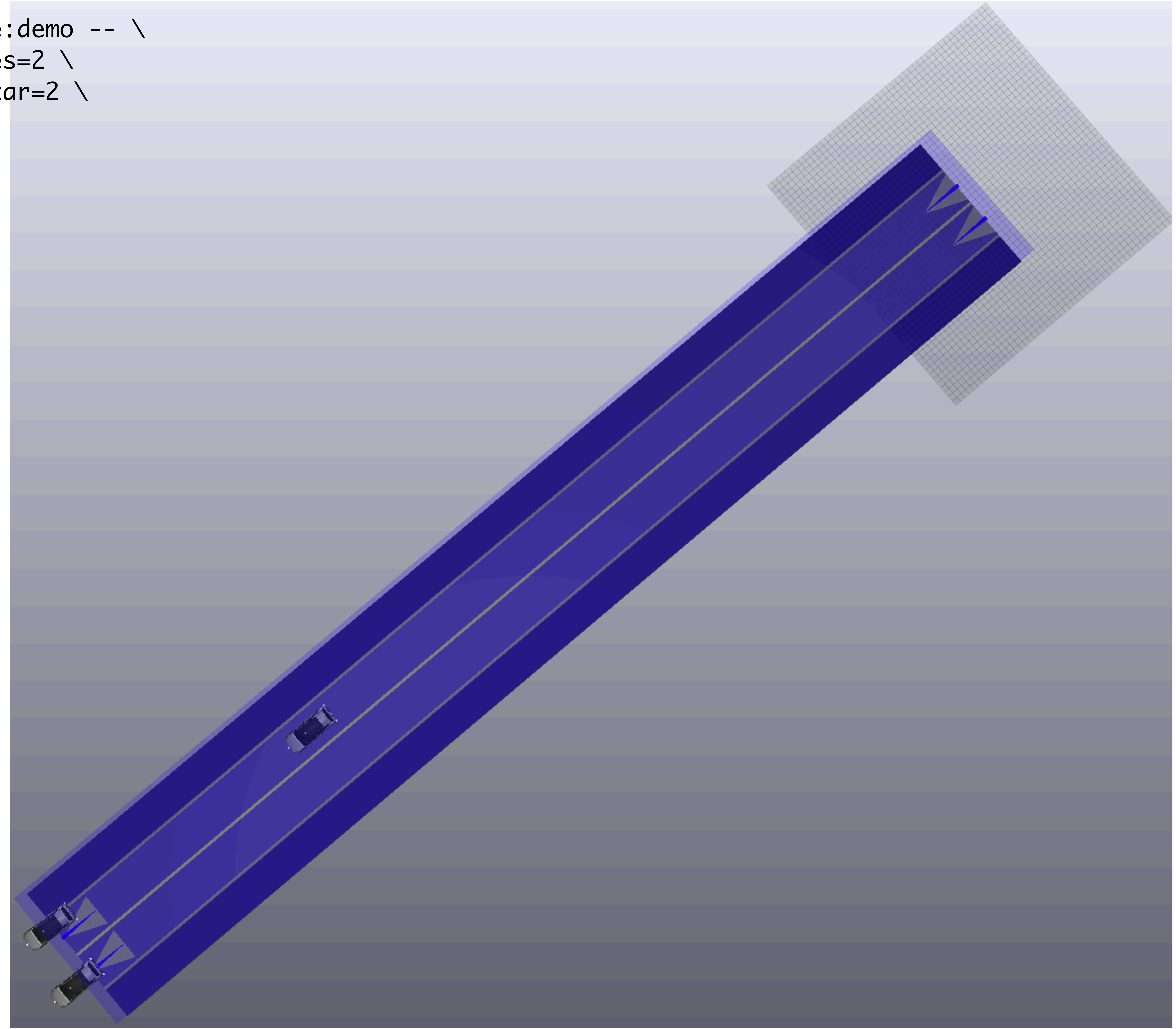
```
$ bazel run automotive:demo -- \
    --num_dragway_lanes=2 \
    --num_trajectory_car=2 \
    --num_mobil_car=1
```



```
$ bazel run automotive:demo -- \
    --num_dragway_lanes=2 \
    --num_trajectory_car=2 \
    --num_mobil_car=1
```



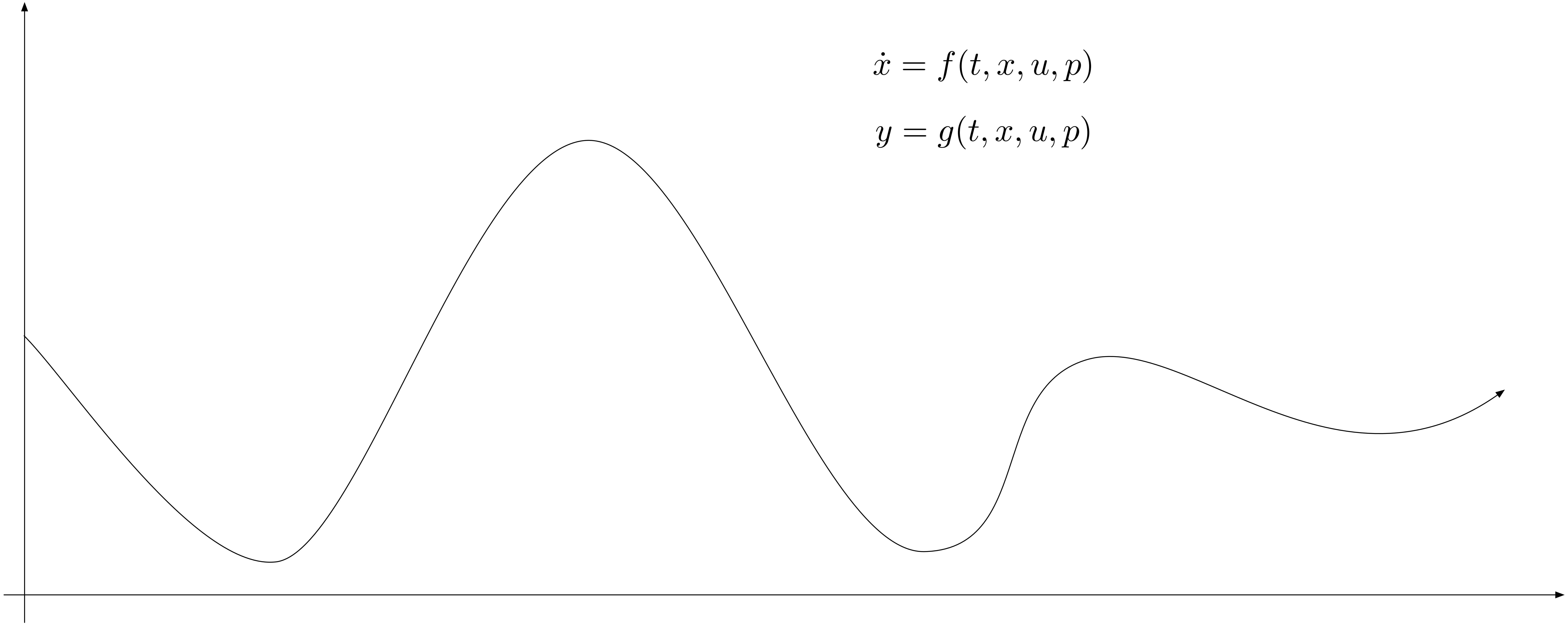
```
$ bazel run automotive:demo -- \
    --num_dragway_lanes=2 \
    --num_trajectory_car=2 \
    --num_mobil_car=1
```



Dynamical Systems

Dynamical Systems

(Continuous)



$$\dot{x} = f(t, x, u, p)$$

$$y = g(t, x, u, p)$$

Drake System Architecture

Include all the information needed
to simulate a system at a given time step:
Time, State, Input, Parameter

$$\text{System} : \text{Context} \to \text{Computation}$$

Including:
Time Derivatives (continuous part),
State Updates (discrete part),
Output,
...

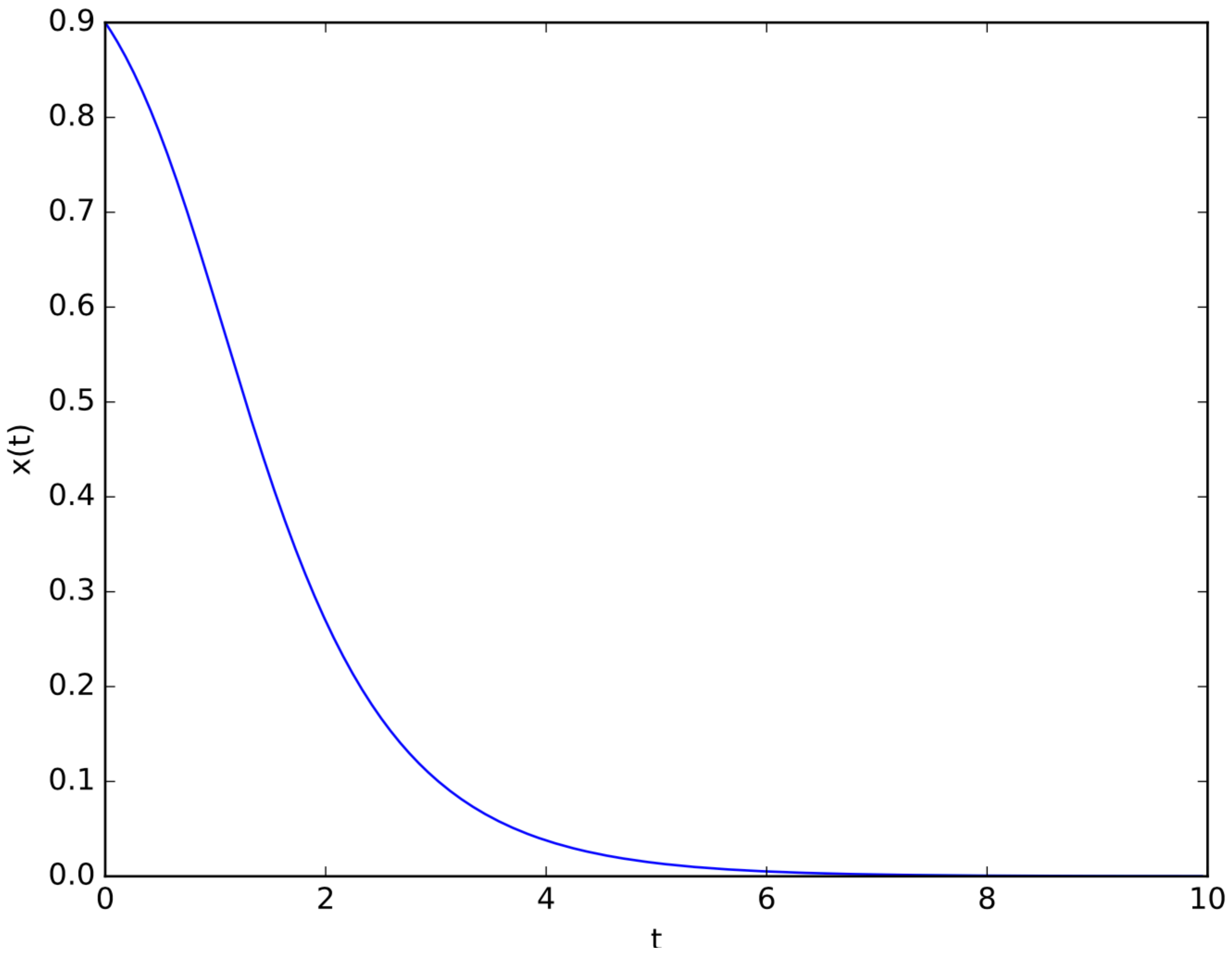
System = Stateless/Immutable Function

Drake System Architecture

```
// Simple Continuous Time System
//   xdot = -x + x^3
//   y = x
class SimpleContinuousTimeSystem : public drake::systems::VectorSystem<double> {
 public:
  SimpleContinuousTimeSystem()
      : drake::systems::VectorSystem<double>(0,     // Zero inputs.
                                             1) {  // One output.
    this->DeclareContinuousState(1);                // One state variable.
  }

 private:
  // xdot = -x + x^3
  virtual void DoCalcVectorTimeDerivatives(
      const drake::systems::Context<double>& context,
      const Eigen::VectorBlock<const Eigen::VectorXd>& input,
      const Eigen::VectorBlock<const Eigen::VectorXd>& state,
      Eigen::VectorBlock<Eigen::VectorXd>* derivatives) const {
   (*derivatives)(0) = -state(0) + std::pow(state(0), 3.0);
  }

  // y = x
  virtual void DoCalcVectorOutput(
      const drake::systems::Context<double>& context,
      const Eigen::VectorBlock<const Eigen::VectorXd>& input,
      const Eigen::VectorBlock<const Eigen::VectorXd>& state,
      Eigen::VectorBlock<Eigen::VectorXd>* output) const {
   *output = state;
  }
};
```

Drake System Architecture

```
// Create the simple system.
SimpleContinuousTimeSystem system;

// Create the simulator.
drake::systems::Simulator<double> simulator(system);

// Set the initial conditions x(0).
drake::systems::ContinuousState<double>& state =
    simulator.get_mutable_context().get_mutable_continuous_state();
state[0] = 0.9;

// Simulate for 10 seconds.
simulator.StepTo(10);

// Make sure the simulation converges to the stable fixed point at x=0.
DRAKE_DEMAND(state[0] < 1.0e-4);
```

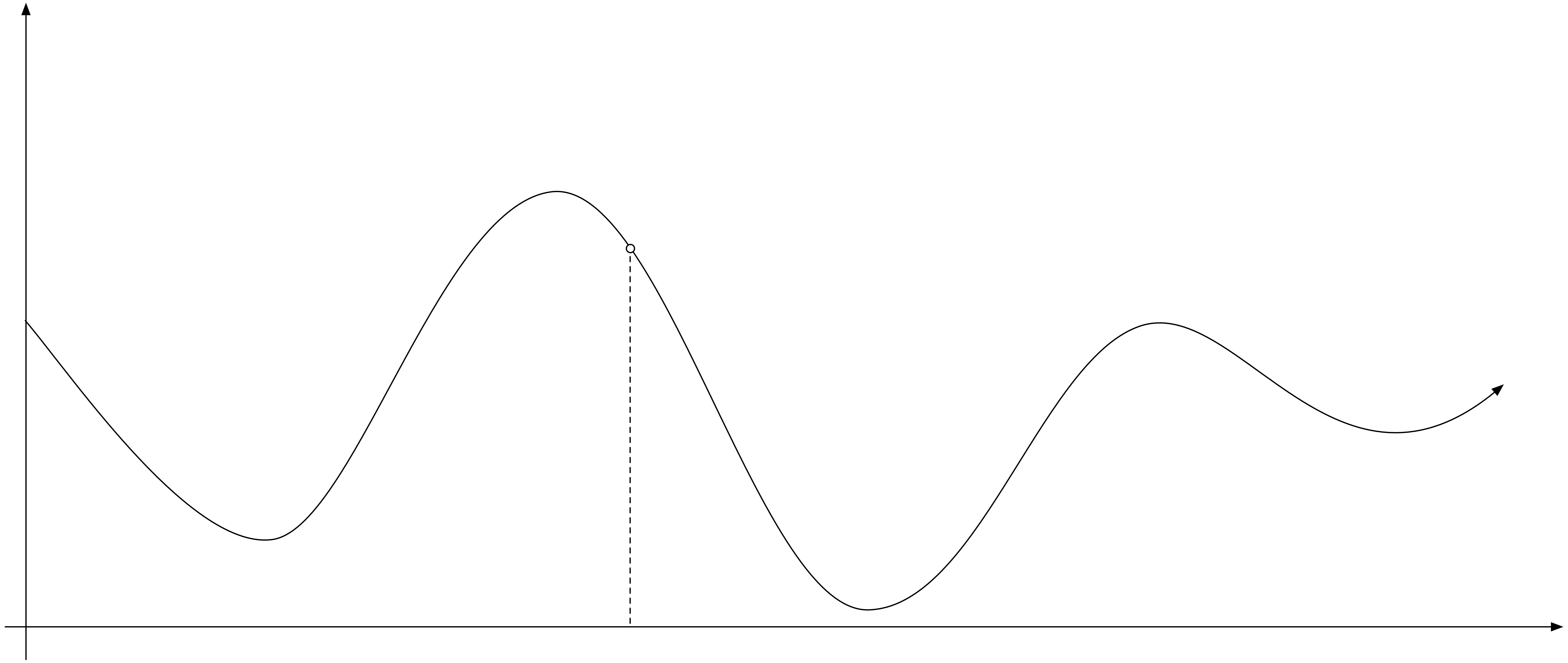


https://github.com/RobotLocomotion/drake/blob/master/examples/simple_continuous_time_system.cc

Templated System Framework

`System<T>` **where** `T` **can be:**

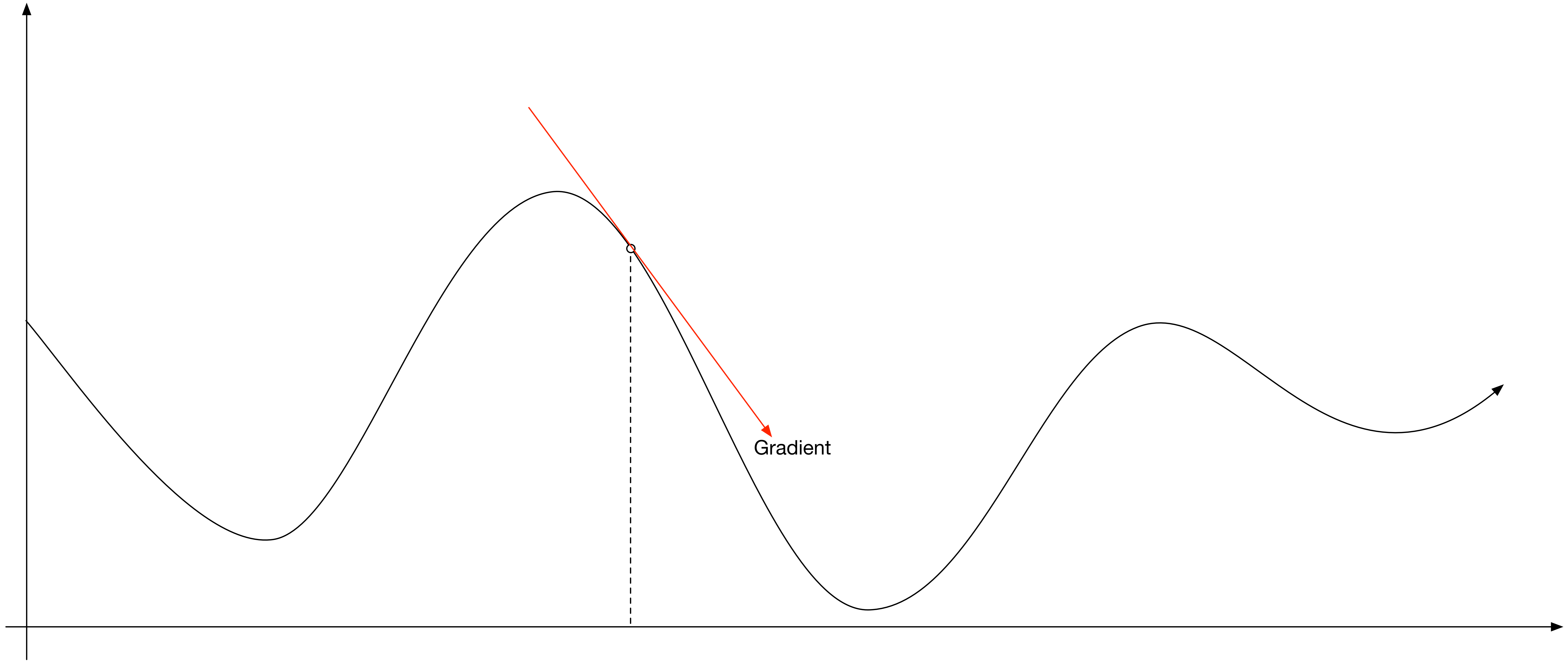
- `double` **for Simulation / Testing**



Templated System Framework

`System<T>` **where** `T` **can be:**

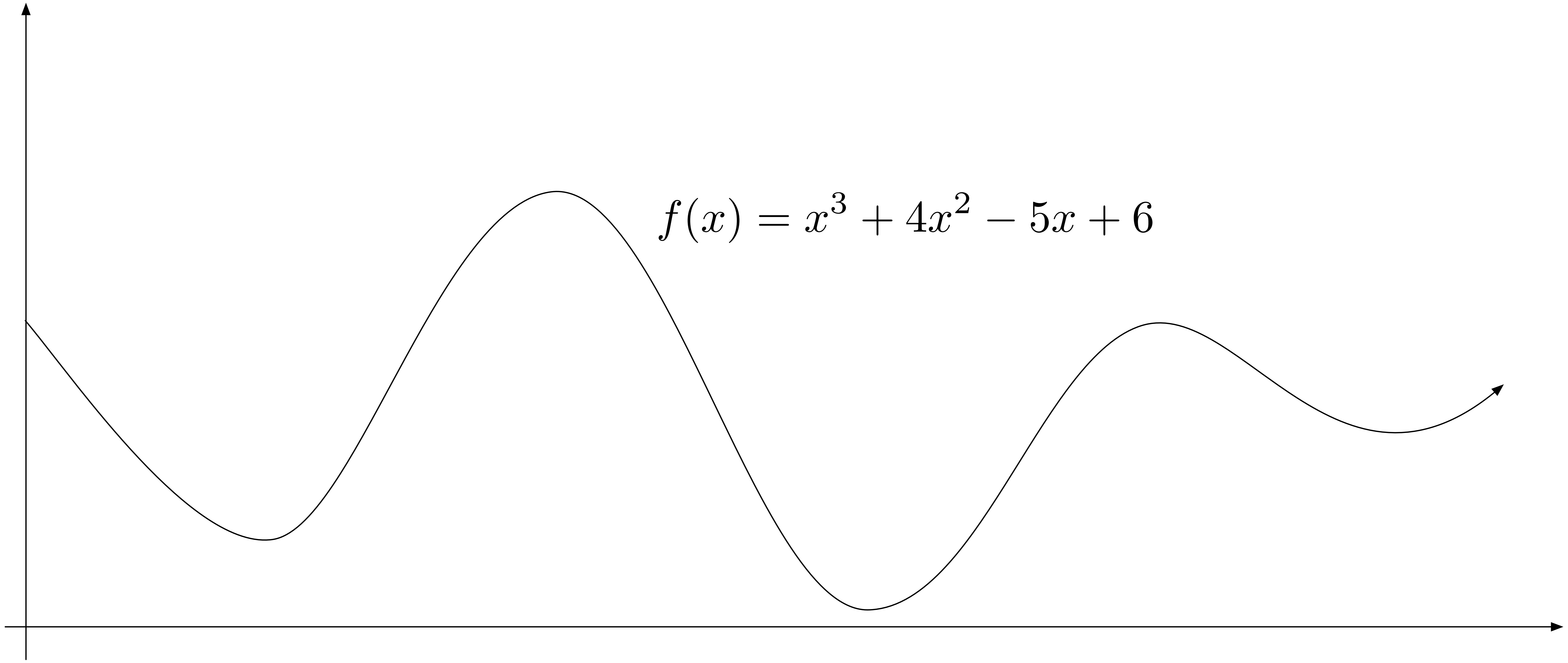
- `double` **for Simulation / Testing**
- `AutoDiff` **for Optimization-based Analysis & Design**



Templated System Framework

`System<T>` **where** `T` **can be:**

- `double` — **for Simulation / Testing**
- `AutoDiff` — **for Optimization-based Analysis & Design**
- `symbolic::Expression` — **for Symbolic Analysis & Verification (e.g. SMT)**



Diagram

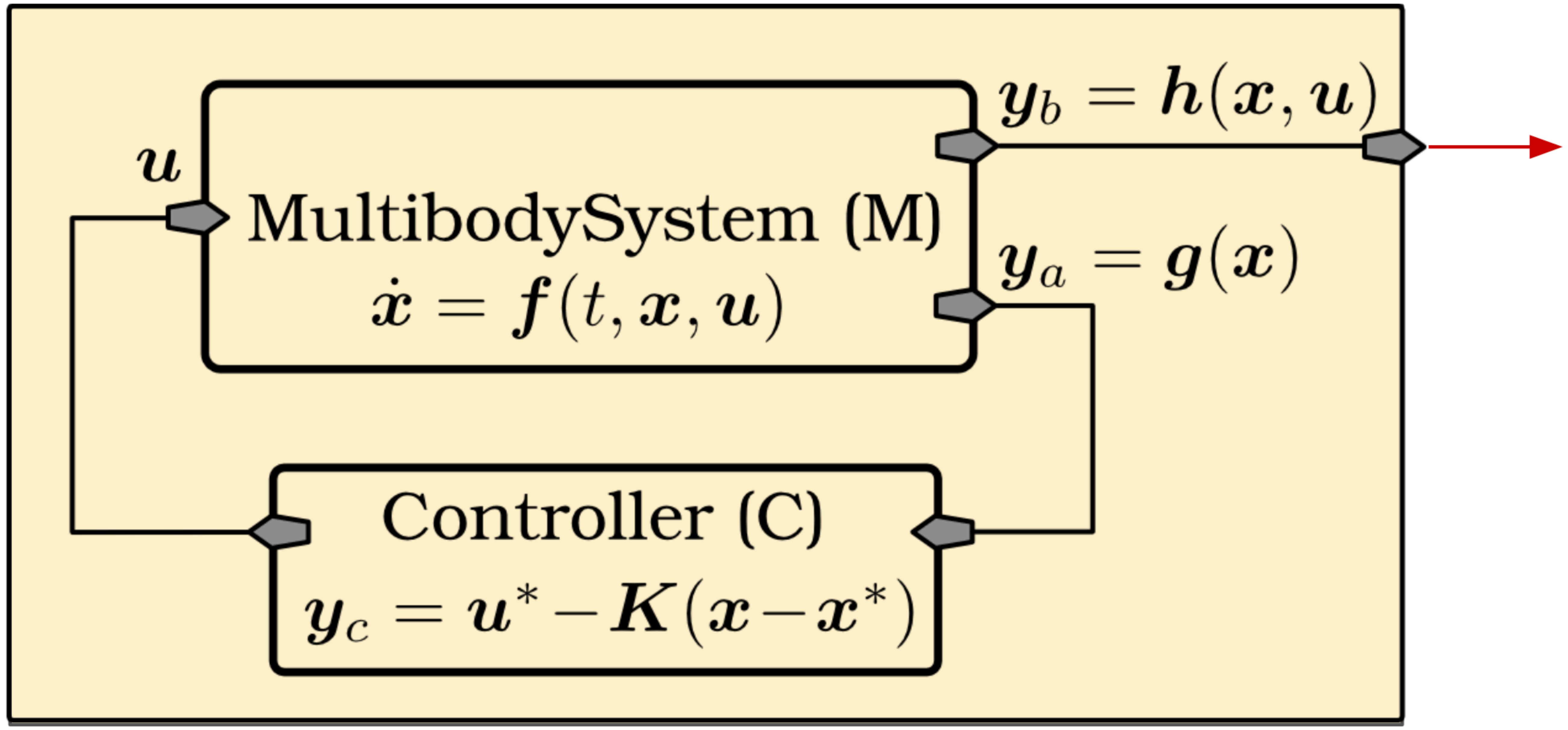
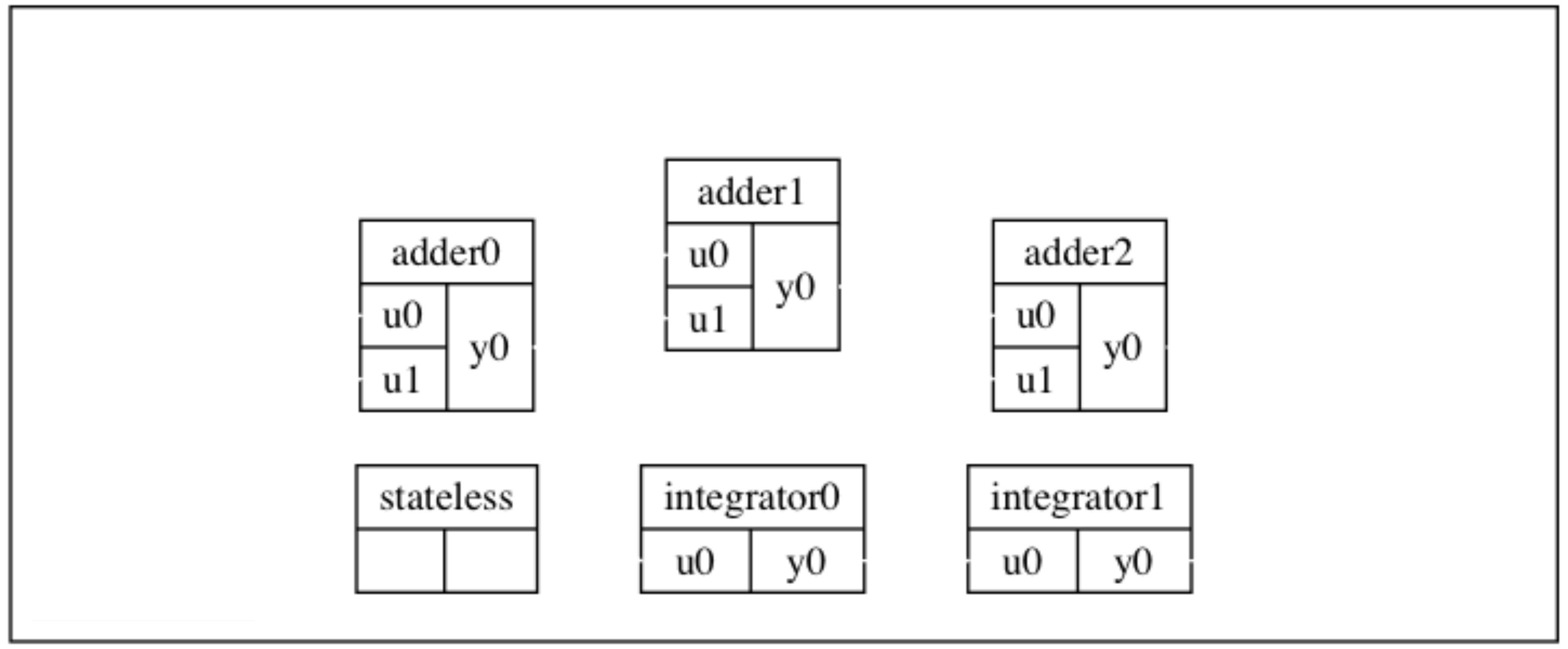


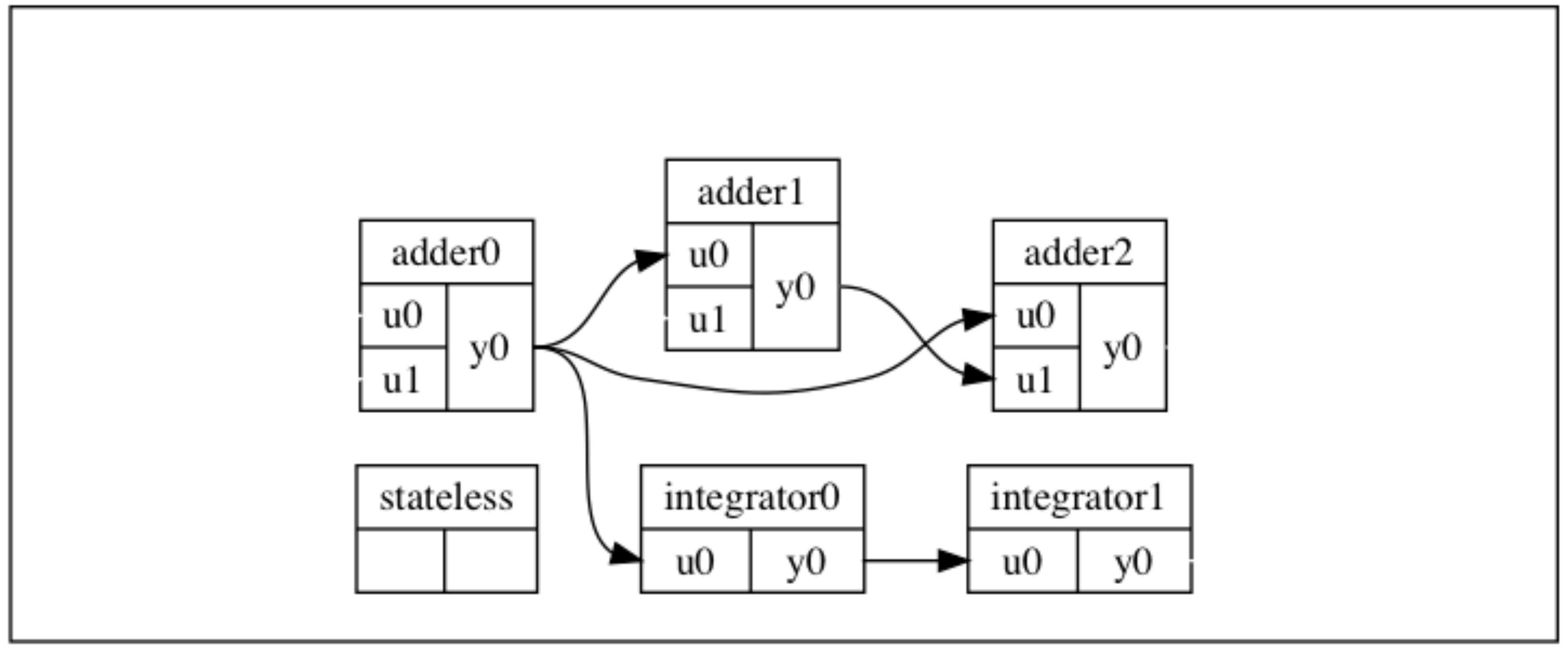
Diagram = A Graph of Systems = A System

Diagram Builder




```
DiagramBuilder<double> builder;
adder0_ = builder.AddSystem<Adder<double>>(2 /* inputs */, size);
adder0_->set_name("adder0");
adder1_ = builder.AddSystem<Adder<double>>(2 /* inputs */, size);
adder1_->set_name("adder1");
adder2_ = builder.AddSystem<Adder<double>>(2 /* inputs */, size);
adder2_->set_name("adder2");
stateless_ = builder.AddSystem<analysis_test::StatelessSystem<double>>(
    1.0 /* trigger time */,
    WitnessFunctionDirection::kCrossesZero);
stateless_->set_name("stateless");

integrator0_ = builder.AddSystem<Integrator<double>>(size);
integrator0_->set_name("integrator0");
integrator1_ = builder.AddSystem<Integrator<double>>(size);
integrator1_->set_name("integrator1");
```

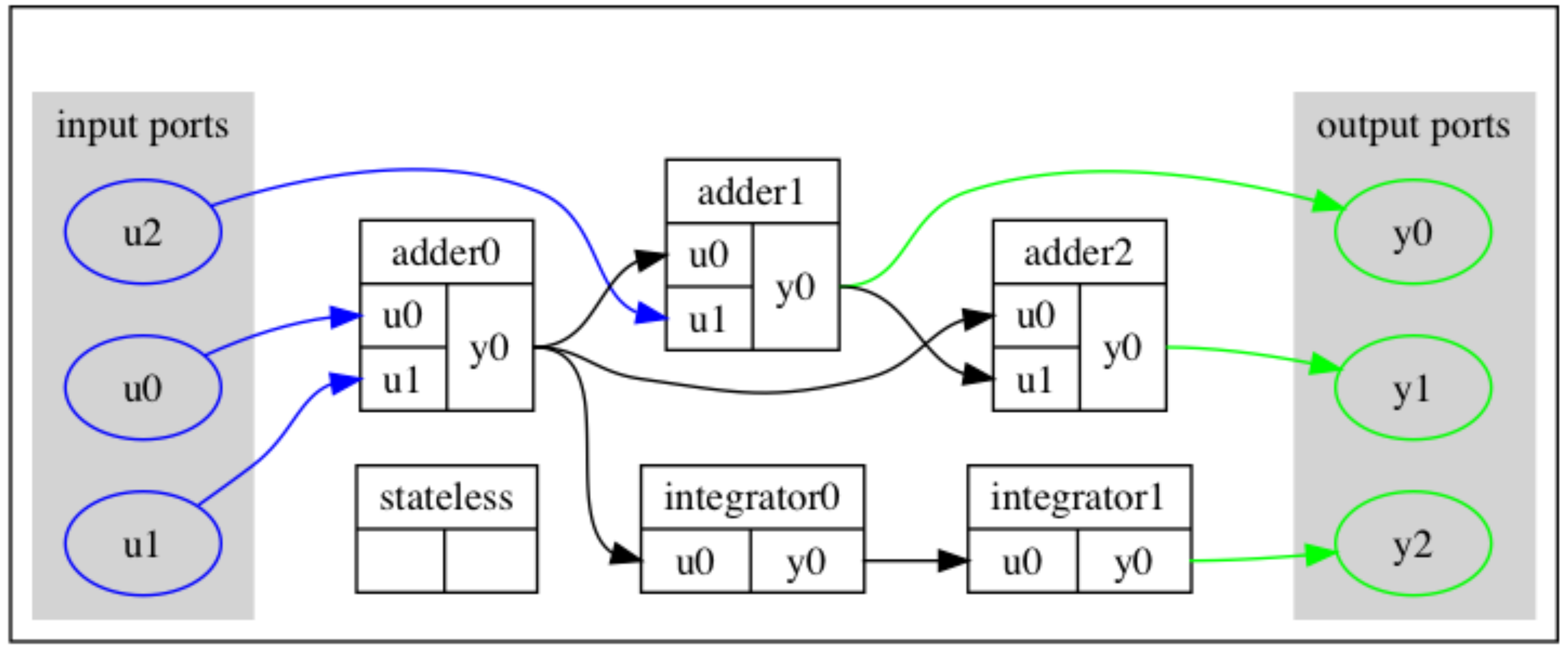
Diagram Builder




```
builder.Connect(adder0_->get_output_port(), adder1_->get_input_port(0));
builder.Connect(adder0_->get_output_port(), adder2_->get_input_port(0));
builder.Connect(adder1_->get_output_port(), adder2_->get_input_port(1));

builder.Connect(adder0_->get_output_port(),
                integrator0_->get_input_port());
builder.Connect(integrator0_->get_output_port(),
                integrator1_->get_input_port());
```

https://github.com/RobotLocomotion/drake/blob/master/systems/framework/test/diagram_test.cc

Diagram Builder




```
builder.ExportInput(adder0_->get_input_port(0));
builder.ExportInput(adder0_->get_input_port(1));
builder.ExportInput(adder1_->get_input_port(1));
builder.ExportOutput(adder1_->get_output_port());
builder.ExportOutput(adder2_->get_output_port());
builder.ExportOutput(integrator1_->get_output_port());

diagram_ = builder.Build();
```

Automotive Systems

Goal: Demystify Automotive Demo


```
$ bazel run automotive:demo -- \
    --num_dragway_lanes=2 \
    --num_trajectory_car=2 \
    --num_mobil_car=1
```

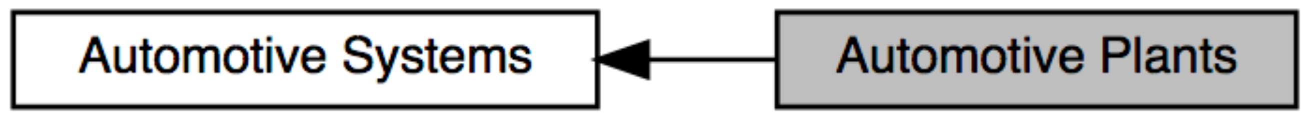
Automotive Plants

Classes

Automotive Plants

Modeling Dynamical Systems » Automotive Systems

Collaboration diagram for Automotive Plants:



Classes

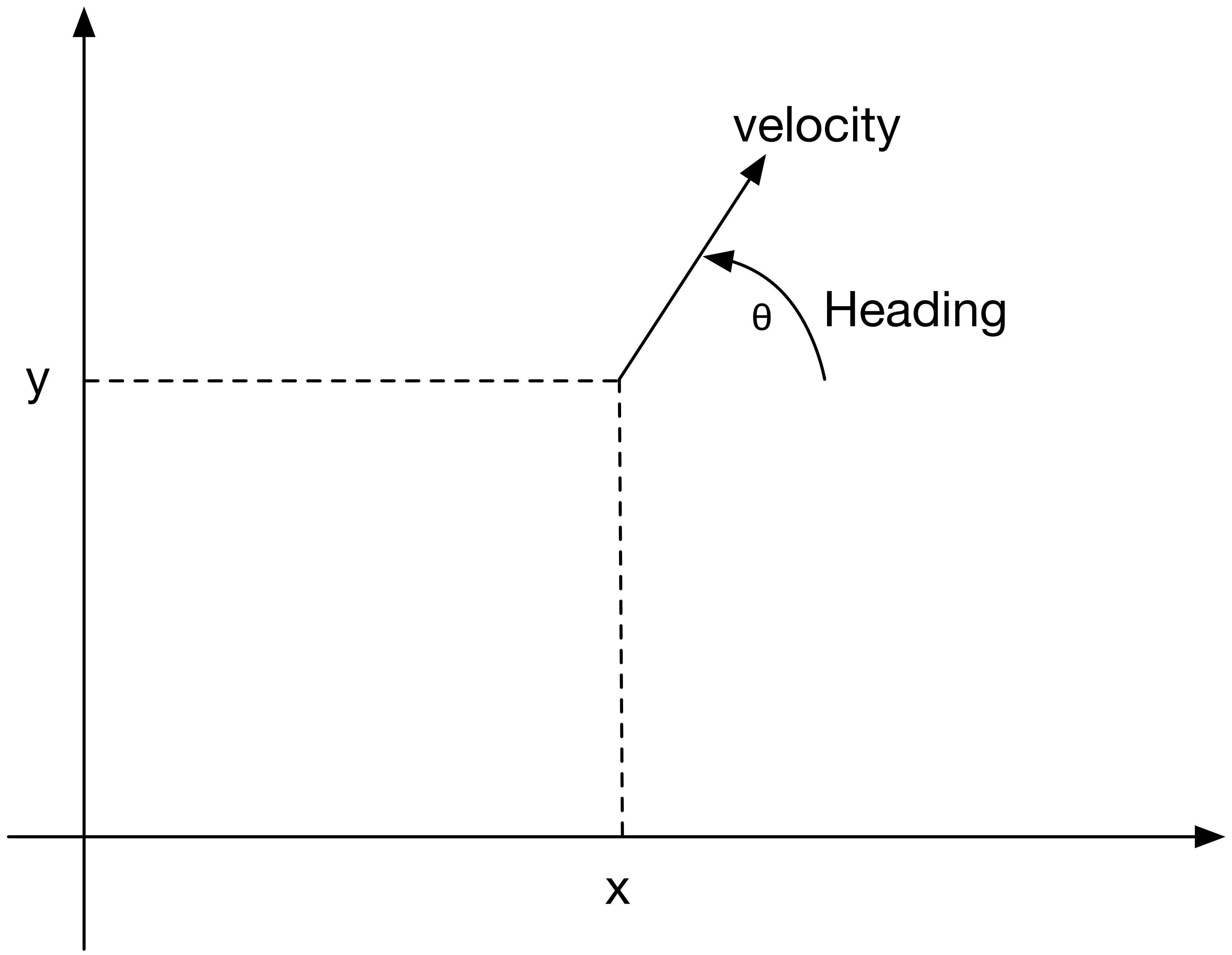
class	**BicycleCar< T >** **BicycleCar** implements a nonlinear rigid body bicycle model from Althoff & Dolan (2014) [1]. More...
class	**MaliputRailcar< T >** **MaliputRailcar** models a vehicle that follows a **maliput::api::Lane** as if it were on rails and neglecting all physics. More...
class	**SimpleCar< T >** **SimpleCar** models an idealized response to driving commands, neglecting all physics. More...
class	**SimplePowertrain< T >** **SimplePowertrain** models a powertrain with first-order lag. More...
class	**TrajectoryCar< T >** **TrajectoryCar** models a car that follows a pre-established trajectory. More...

Detailed Description

Actuated System models related to automotive software.

SimpleCar

State




```
element {
    name: "x"
    doc: "x"
    default_value: "0.0"
}
element {
    name: "y"
    doc: "y"
    default_value: "0.0"
}
element {
    name: "heading"
    doc: "heading"
    default_value: "0.0"
}
element {
    name: "velocity"
    doc: "velocity"
    default_value: "0.0"
}
```

https://github.com/RobotLocomotion/drake/blob/master/automotive/simple_car_state.named_vector

SimpleCar

Parameter

```
namespace: "drake::automotive"

# The defaults in this file approximate a 2010 Toyota Prius.
element {
    name: "wheelbase"
    doc: "The distance between the front and rear axles of the vehicle."
    doc_units: "m"
    default_value: "2.700"
    min_value: "0.0"
}
element {
    name: "track"
    doc: "The distance between the center of two wheels on the same axle."
    doc_units: "m"
    default_value: "1.521"
    min_value: "0.0"
}
element {
    name: "max_abs_steering_angle"
    doc: "The limit on the driving_command.steering angle input
         (the desired steering angle of a virtual center wheel);
          this element is applied symmetrically to both left- and right-turn limits."
    doc_units: "rad"
    default_value: "0.471"  # 27 degrees.
    min_value: "0.0"
}
```

```
element {
    name: "max_velocity"
    doc: "The limit on the car's forward speed."
    doc_units: "m/s"
    default_value: "45.0"
    min_value: "0.0"
}
element {
    name: "max_acceleration"
    doc: "The limit on the car's acceleration and deceleration."
    doc_units: "m/s^2"
    default_value: "4.0"
    min_value: "0.0"
}
element {
    name: "velocity_limit_kp"
    doc: "The smoothing constant for min/max velocity limits."
    doc_units: "Hz"
    default_value: "10.0"
    min_value: "0.0"
}
```

https://github.com/RobotLocomotion/drake/blob/master/automotive/simple_car_params.named_vector

SimpleCar

Input

```
namespace: "drake::automotive"

element {
     name: "steering_angle"
     doc: "The desired steering angle of a virtual center wheel, positive results in the vehicle turning left."
     doc_units: "rad"
     default_value: "0.0"
}

element {
     name: "acceleration"
     doc: "The signed acceleration, positive means speed up; negative means slow down, but should not move in reverse."
     doc_units: "m/s^2"
     default_value: "0.0"
}
```

https://github.com/RobotLocomotion/drake/blob/master/automotive/driving_command.named_vector

SimpleCar

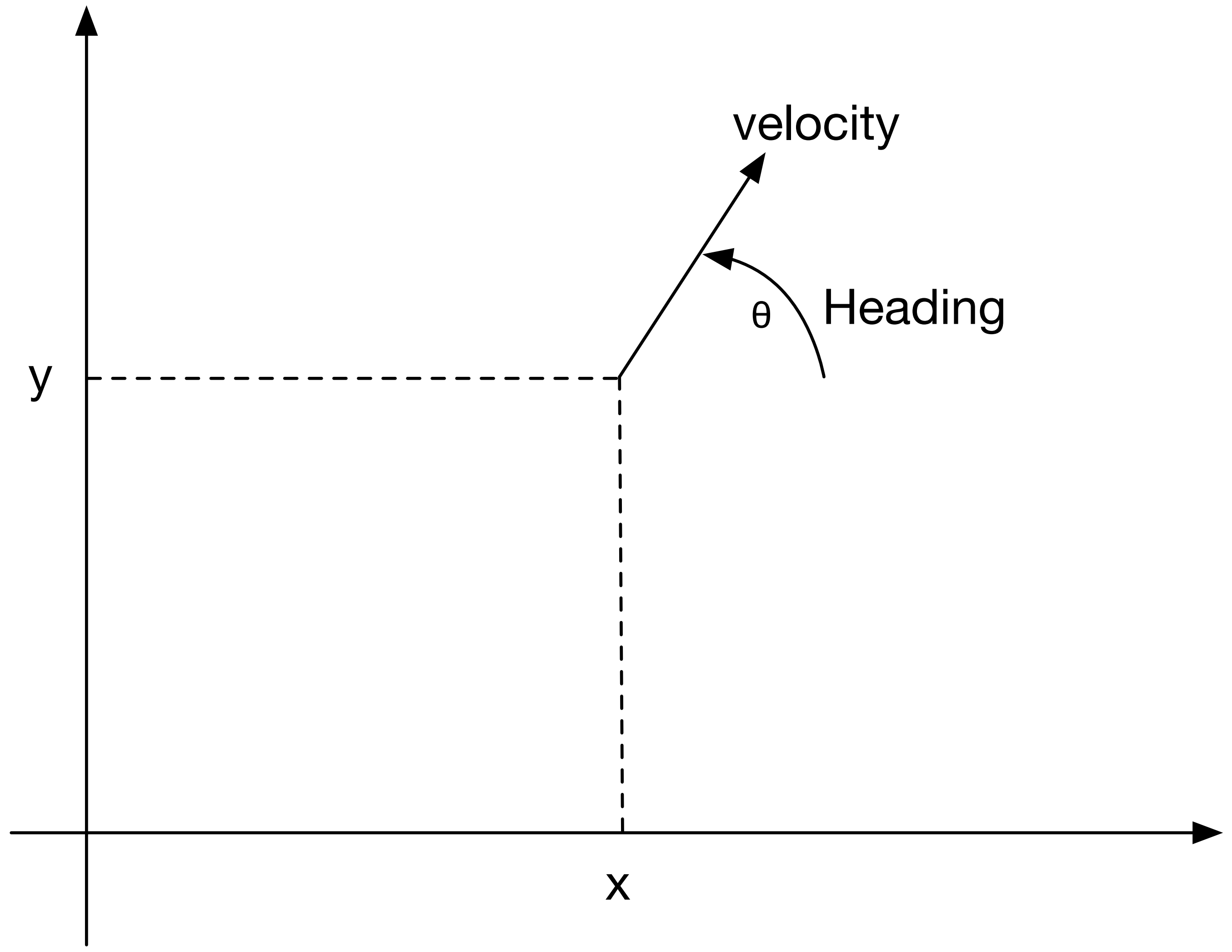
Output

1. **State vector : Position (x, y, heading)**
   + Velocity

SimpleCar

Output

1. **State vector : Position (x, y, heading)**
   + Velocity
2. **PoseVector (7D)**
   Translation (3D) + Rotation (4D)




```
template <typename T>
void SimpleCar<T>::CalcPose(const systems::Context<T>& context,
                            PoseVector<T>* pose) const {
  const SimpleCarState<T>& state = get_state(context);
  pose->set_translation(Eigen::Translation<T, 3>(state.x(), state.y(), 0));
  const Vector3<T> z_axis{0.0, 0.0, 1.0};
  const Eigen::AngleAxis<T> rotation(state.heading(), z_axis);
  pose->set_rotation(Eigen::Quaternion<T>(rotation));
}
```

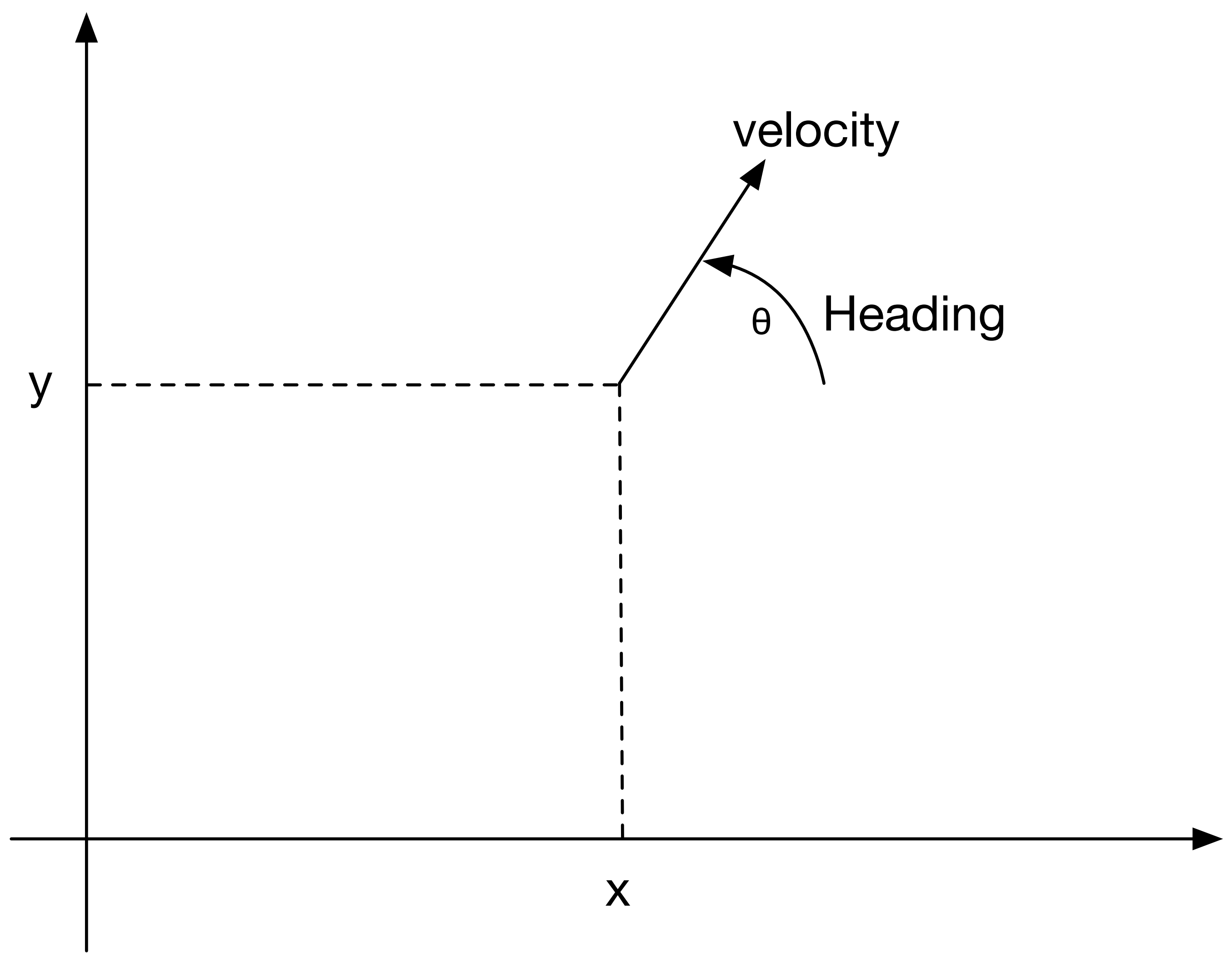
https://github.com/RobotLocomotion/drake/blob/master/automotive/simple_car.cc

SimpleCar

Output

1. **State vector : Position (x, y, heading)**
   + Velocity
2. **PoseVector (7D)**
3. **FrameVelocity (6D)**

Derivatives of x-y-z translation (3D) + the derivatives of x-y-z rotation (3D)



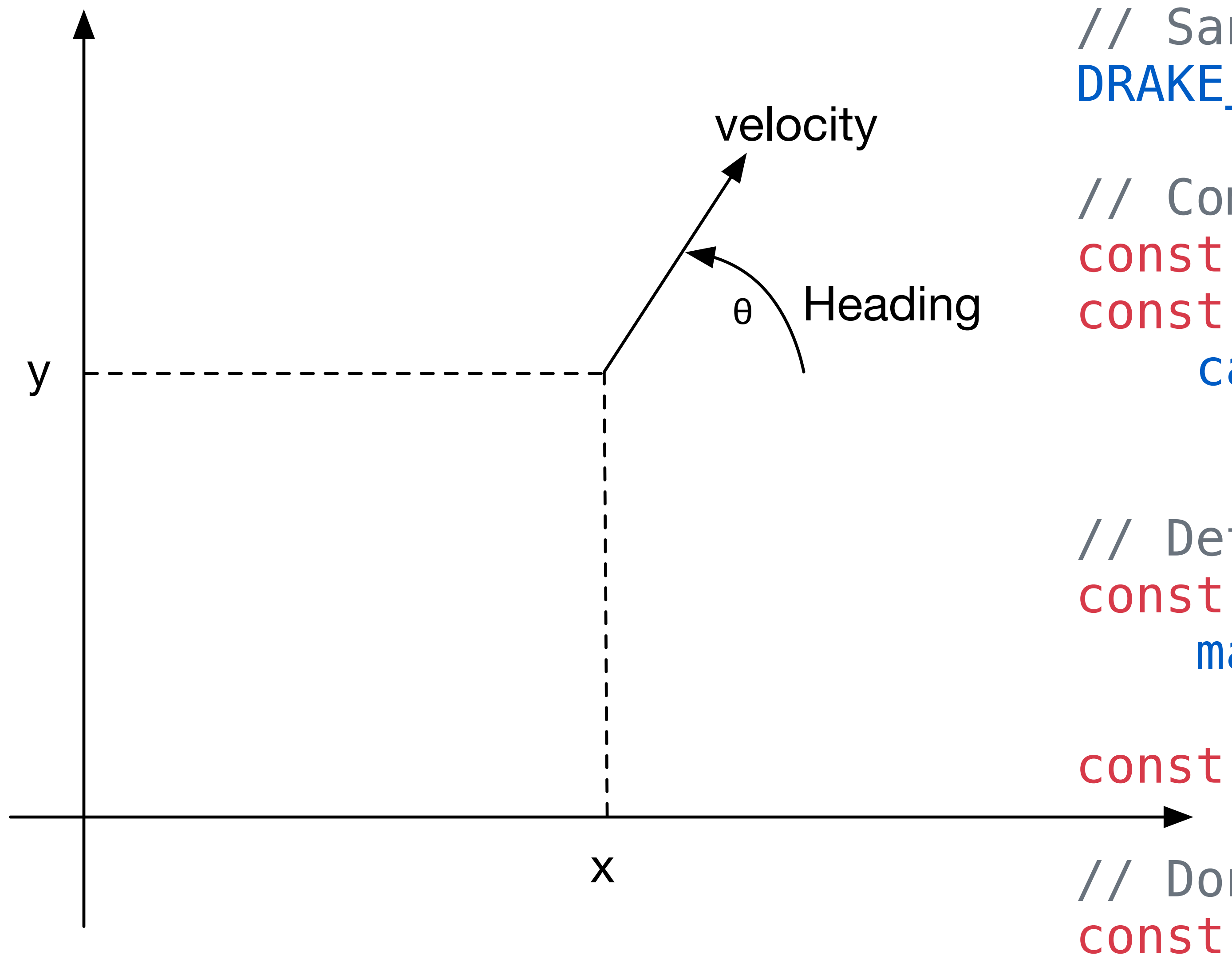

```
template <typename T>
void SimpleCar<T>::CalcVelocity(
    const systems::Context<T>& context,
    systems::rendering::FrameVelocity<T>* velocity) const {
  const SimpleCarState<T>& state = get_state(context);
  const T nonneg_velocity = max(T(0), state.velocity());

  // Convert the state derivatives into a spatial velocity.
  multibody::SpatialVelocity<T> output;
  output.translational().x() = nonneg_velocity * cos(state.heading());
  output.translational().y() = nonneg_velocity * sin(state.heading());
  output.translational().z() = T(0);
  output.rotational().x() = T(0);
  output.rotational().y() = T(0);
  // The rotational velocity around the z-axis is actually rates.heading(),
  // which is a function of the input steering angle. We set it to zero so that
  // this system is not direct-feedthrough.
  output.rotational().z() = T(0);
  velocity->set_velocity(output);
}
```

https://github.com/RobotLocomotion/drake/blob/master/automotive/simple_car.cc

SimpleCar

Dynamics

```
template <typename T>
void SimpleCar<T>::ImplCalcTimeDerivatives(const SimpleCarParams<T>& params,
                                           const SimpleCarState<T>& state,
                                           const DrivingCommand<T>& input,
                                           SimpleCarState<T>* rates) const {

  using std::abs;
  using std::cos;
  using std::max;
  using std::sin;

  // Sanity check our input.
  DRAKE_DEMAND(abs(input.steering_angle()) < M_PI);

  // Compute the smooth acceleration that the vehicle actually executes.
  const T desired_acceleration = input.acceleration();
  const T smooth_acceleration =
      calc_smooth_acceleration(desired_acceleration, params.max_velocity(),
                               params.velocity_limit_kp(), state.velocity());

  // Determine steering.
  const T saturated_steering_angle =
      math::saturate(input.steering_angle(), -params.max_abs_steering_angle(),
                     params.max_abs_steering_angle());
  const T curvature = tan(saturated_steering_angle) / params.wheelbase();

  // Don't allow small negative velocities to affect position or heading.
  const T nonneg_velocity = max(T(0), state.velocity());

  rates->set_x(nonneg_velocity * cos(state.heading()));
  rates->set_y(nonneg_velocity * sin(state.heading()));
  rates->set_heading(curvature * nonneg_velocity);
  rates->set_velocity(smooth_acceleration);
}
```



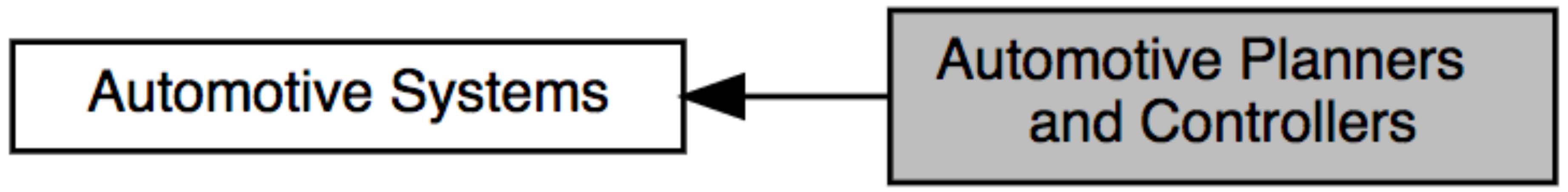
Automotive Planners and Controllers

Classes

Automotive Planners and Controllers

Modeling Dynamical Systems » Automotive Systems

Collaboration diagram for Automotive Planners and Controllers:



Classes

class **IdmController< T >**
IdmController implements the IDM (Intelligent Driver Model) planner, computed based only on the nearest car ahead. More...

class **MobilPlanner< T >**
MOBIL (Minimizing Overall Braking Induced by Lane Changes) [1] is a planner that minimizes braking requirement for the ego car while also minimizing (per a weighting factor) the braking requirements of any trailing cars within the ego car's immediate neighborhood. More...

class **PurePursuitController< T >**
PurePursuitController implements a pure pursuit controller. More...

SET DESIRED GAP

Photo Credit: Youtube - Adaptive Cruise Control (ACC)- Animated Quick Guide

SET DESIRED GAP

Photo Credit: Youtube - Adaptive Cruise Control (ACC)- Animated Quick Guide

SET DESIRED GAP

IDM (Intelligent Driver Model)

In traffic flow modeling, the **intelligent driver model** (**IDM**) is a time-continuous car-following model for the simulation of freeway and urban traffic. It was developed by Treiber, Hennecke and Helbing in 2000 to improve upon results provided with other "intelligent" driver models such as Gipps' model, which lose realistic properties in the deterministic limit.

Contents [hide]

Model definition [edit]

As a car-following model, the IDM describes the dynamics of the positions and velocities of single vehicles. For vehicle α, x_α denotes its position at time t, and v_α its velocity. Furthermore, l_α gives the length of the vehicle. To simplify notation, we define the *net distance* $s_\alpha := x_{\alpha-1} - x_\alpha - l_{\alpha-1}$, where $\alpha - 1$ refers to the vehicle directly in front of vehicle α, and the velocity difference, or *approaching rate*, $\Delta v_\alpha := v_\alpha - v_{\alpha-1}$. For a simplified version of the model, the dynamics of vehicle α are then described by the following two ordinary differential equations:

$$\dot{x}_\alpha = \frac{\mathrm{d}x_\alpha}{\mathrm{d}t} = v_\alpha$$

$$\dot{v}_\alpha = \frac{\mathrm{d}v_\alpha}{\mathrm{d}t} = a\left(1 - \left(\frac{v_\alpha}{v_0}\right)^\delta - \left(\frac{s^*(v_\alpha, \Delta v_\alpha)}{s_\alpha}\right)^2\right)$$

$$\text{with } s^*(v_\alpha, \Delta v_\alpha) = s_0 + v_\alpha T + \frac{v_\alpha \Delta v_\alpha}{2\sqrt{a\,b}}$$

IdmPlanner

template<typename T>
class drake::automotive::IdmPlanner< T >

IdmPlanner implements the IDM (Intelligent Driver Model) equation governing longitudinal accelerations of a vehicle in single-lane traffic [1, 2].

It is derived based on qualitative observations of actual driving behavior and captures objectives such as keeping a safe distance behind a lead vehicle, maintaining a desired speed, and accelerating and decelerating within comfortable limits.

The IDM equation produces accelerations that realize smooth transitions between the following three modes:

- Free-road behavior: when the distance to the leading car is large, the IDM regulates acceleration to match the desired speed `v_0`.
- Fast-closing-speed behavior: when the target distance decreases, an interaction term compensates for the velocity difference, while keeping deceleration comfortable according to parameter `b`.
- Small-distance behavior: within small net distances to the lead vehicle, comfort is ignored in favor of increasing this distance to `s_0`.

See the corresponding .cc file for details about the IDM equation.

Instantiated templates for the following kinds of T's are provided:

- double
- **drake::AutoDiffXd**
- **drake::symbolic::Expression**

They are already available to link against in the containing library.

[1] Martin Treiber and Arne Kesting. Traffic Flow Dynamics, Data, Models, and Simulation. Springer, 2013.

[2] https://en.wikipedia.org/wiki/Intelligent_driver_model.

IdmPlanner

template<typename T>
class drake::automotive::IdmPlanner< T >

IdmPlanner implements the IDM (Intelligent Driver Model) equation governing longitudinal accelerations of a vehicle in single-lane traffic [1, 2].

It is derived based on qualitative observations of actual driving behavior and captures objectives such as keeping a safe distance behind a lead vehicle, maintaining a desired speed, and accelerating and decelerating within comfortable limits.

The IDM equation produces accelerations that realize smooth transitions between the following three modes:

LONG
MEDIUM
SHORT

- Free-road behavior: when the distance to the leading car is large, the IDM regulates acceleration to match the desired speed `v_0`.
- Fast-closing-speed behavior: when the target distance decreases, an interaction term compensates for the velocity difference, while keeping deceleration comfortable according to parameter `b`.
- Small-distance behavior: within small net distances to the lead vehicle, comfort is ignored in favor of increasing this distance to `s_0`.

See the corresponding .cc file for details about the IDM equation.

Instantiated templates for the following kinds of T's are provided:

- double
- **drake::AutoDiffXd**
- **drake::symbolic::Expression**

They are already available to link against in the containing library.

[1] Martin Treiber and Arne Kesting. Traffic Flow Dynamics, Data, Models, and Simulation. Springer, 2013.

[2] https://en.wikipedia.org/wiki/Intelligent_driver_model.

http://drake.mit.edu/doxygen_cxx/classdrake_1_1automotive_1_1_idm_planner.html

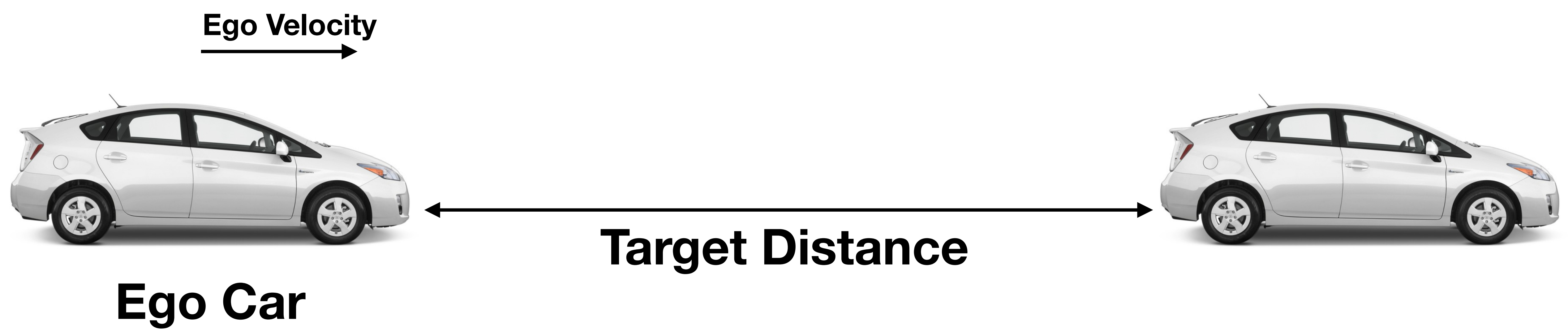
IdmPlanner

```
const T Evaluate ( const IdmPlannerParameters< T > &  params,
                   const T &                          ego_velocity,
                   const T &                          target_distance,
                   const T &                          target_distance_dot
                 )
```

static

Evaluates the IDM equation for the chosen planner parameters `params`, given the current velocity `ego_velocity`, distance to the lead car `target_distance`, and the closing velocity `target_distance_dot`.

The returned value is a longitudinal acceleration.



IdmPlanner

```cpp
template <typename T>
const T IdmPlanner<T>::Evaluate(const IdmPlannerParameters<T>& params,
                                const T& ego_velocity, const T& target_distance,
                                const T& target_distance_dot) {
  const T& v_ref = params.v_ref();
  const T& a = params.a();
  const T& b = params.b();
  const T& s_0 = params.s_0();
  const T& time_headway = params.time_headway();
  const T& delta = params.delta();

  // Compute the interaction acceleration terms.
  const T& closing_term =
      ego_velocity * target_distance_dot / (2 * sqrt(a * b));
  const T& too_close_term = s_0 + ego_velocity * time_headway;
  const T& accel_interaction =
      cond(target_distance < std::numeric_limits<T>::infinity(),
           pow((closing_term + too_close_term) / target_distance, 2.), T(0.));

  // Compute the free-road acceleration term.
  const T accel_free_road = pow(max(T(0.), ego_velocity) / v_ref, delta);

  // Compute the resultant acceleration (IDM equation).
  return a * (1. - accel_free_road - accel_interaction);
}
```

IdmController

Input

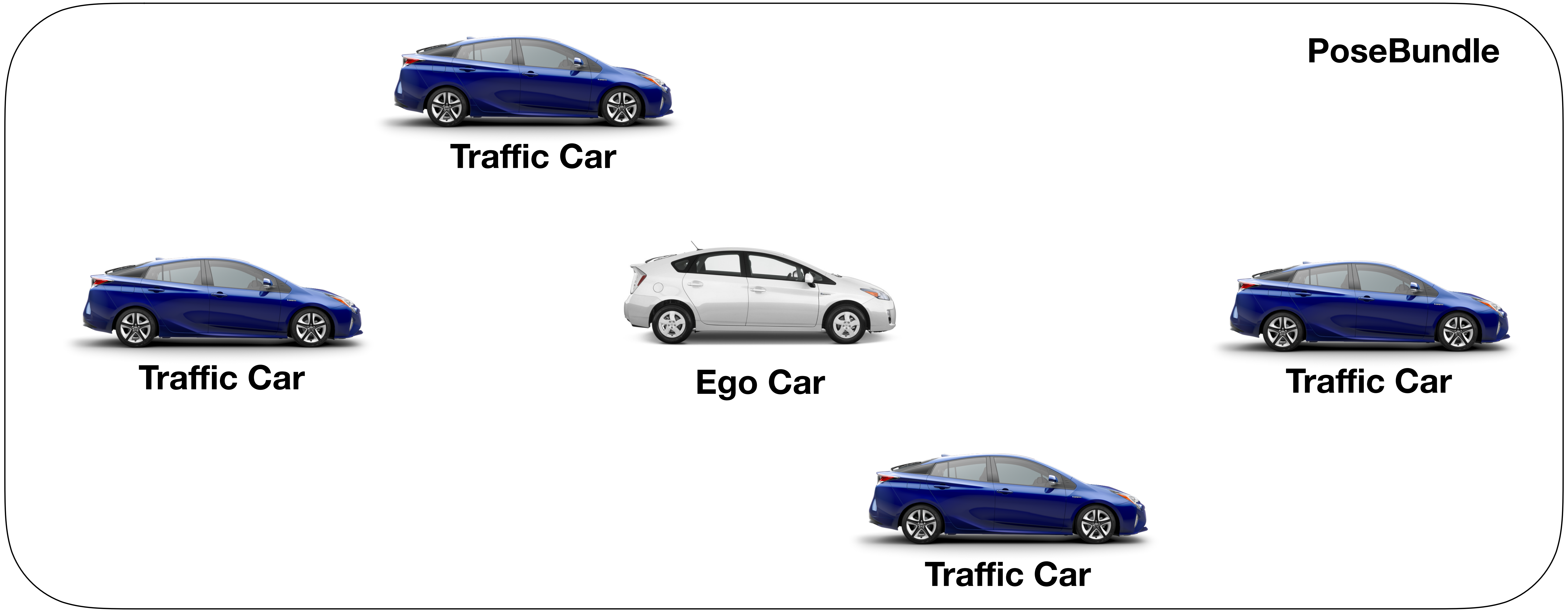
1. **PoseVector(7D) for the ego car**
2. **FrameVelocity (6D) of the ego car**
3. **PoseBundle for the traffic cars**

Ego Car

IdmController

Input

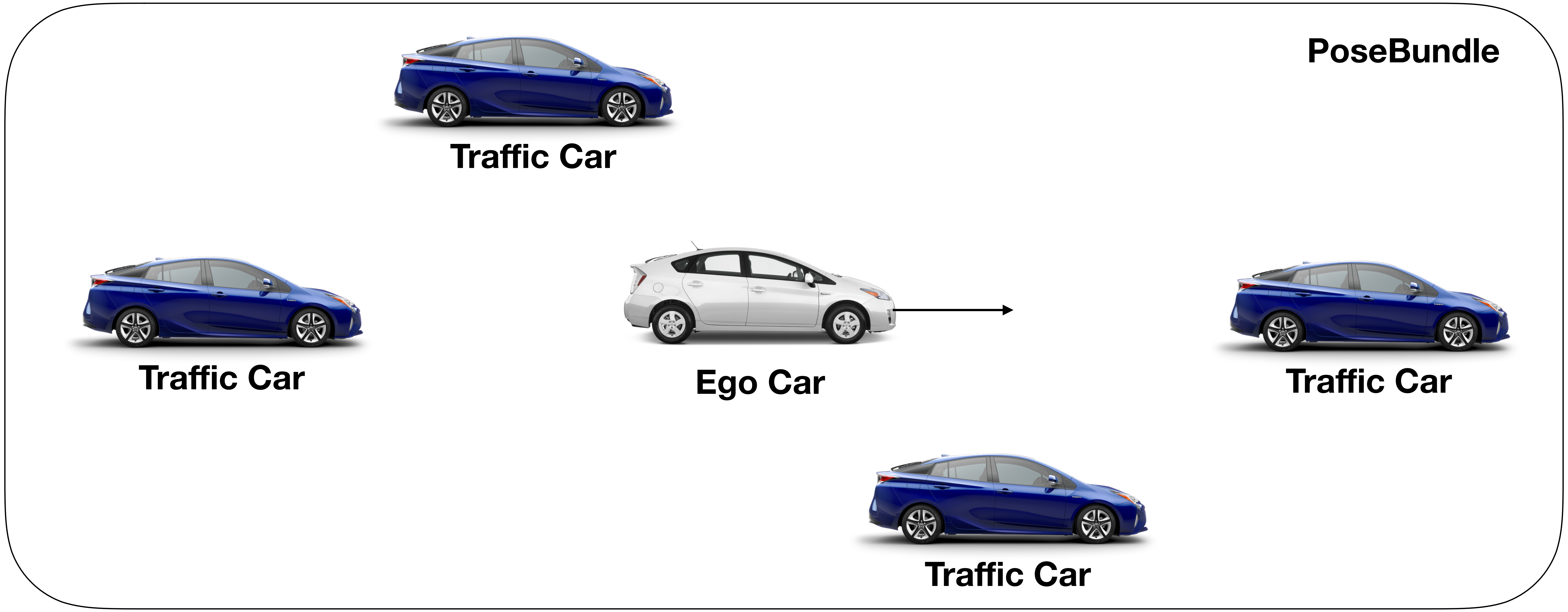
1. **PoseVector(7D) for the ego car**
2. **FrameVelocity (6D) of the ego car**
3. **PoseBundle for the traffic cars**



IdmController

Output

1. **Acceleration of the ego car**

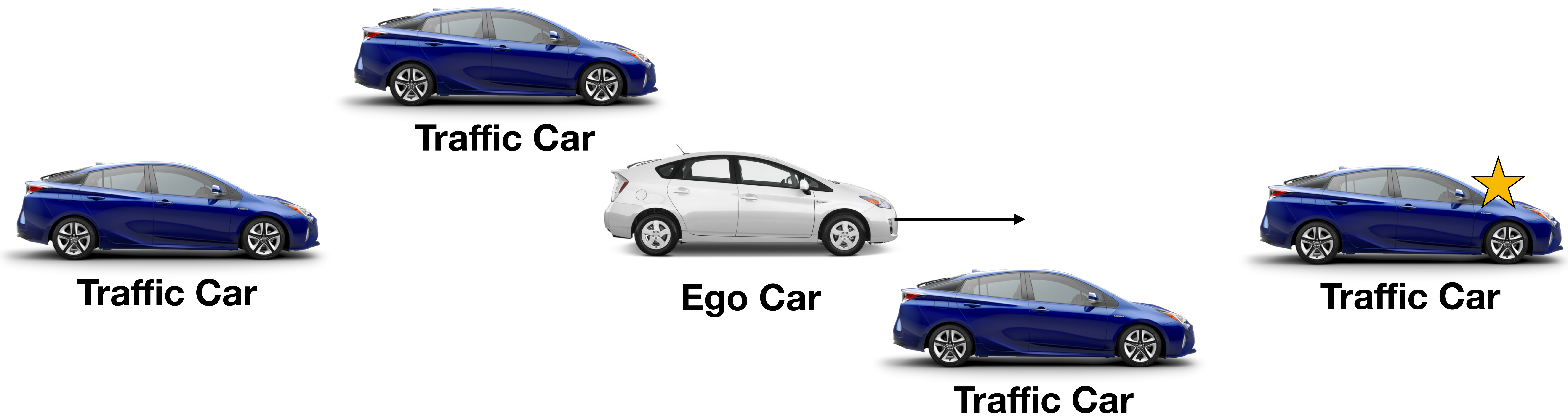


https://github.com/RobotLocomotion/drake/blob/master/automotive/idm_controller.h

IdmController

```
template <typename T>
void IdmController<T>::ImplCalcAcceleration(
    const PoseVector<T>& ego_pose, const FrameVelocity<T>& ego_velocity,
    const PoseBundle<T>& traffic_poses,
    const IdmPlannerParameters<T>& idm_params,
    const RoadPosition& ego_rp,
    systems::BasicVector<T>* command) const {
 RoadPosition ego_position = ego_rp;
  if (!ego_rp.lane) {
    const auto gp =
        GeoPositionT<T>::FromXyz(ego_pose.get_isometry().translation());
    ego_position =
        road_.ToRoadPosition(gp.MakeDouble(), nullptr, nullptr, nullptr);
  }

  // Find the single closest car ahead.
  const ClosestPose<T> lead_car_pose = PoseSelector<T>::FindSingleClosestPose(
      ego_position.lane, ego_pose, traffic_poses,
      idm_params.scan_ahead_distance(), AheadOrBehind::kAhead,
      path_or_branches_);
```

```
  const T headway_distance = lead_car_pose.distance;
  const LanePositionT<T> lane_position(T(ego_position.pos.s()),
                                       T(ego_position.pos.r()),
                                       T(ego_position.pos.h()));
  const T s_dot_ego = PoseSelector<T>::GetSigmaVelocity(
      {ego_position.lane, lane_position, ego_velocity});
  const T s_dot_lead =
      (abs(lead_car_pose.odometry.pos.s()) ==
       std::numeric_limits<T>::infinity())
          ? T(0.)
          : PoseSelector<T>::GetSigmaVelocity(lead_car_pose.odometry);

  // Saturate the net_distance at `idm_params.distance_lower_limit()` away from
  // the ego car to avoid near-singular solutions inherent to the IDM equation.
  const T actual_headway = headway_distance - idm_params.bloat_diameter();
  const T net_distance = max(actual_headway, idm_params.distance_lower_limit());
  const T closing_velocity = s_dot_ego - s_dot_lead;

  // Compute the acceleration command from the IDM equation.
  (*command)[0] = IdmPlanner<T>::Evaluate(idm_params, s_dot_ego, net_distance,
                                          closing_velocity);
}
```



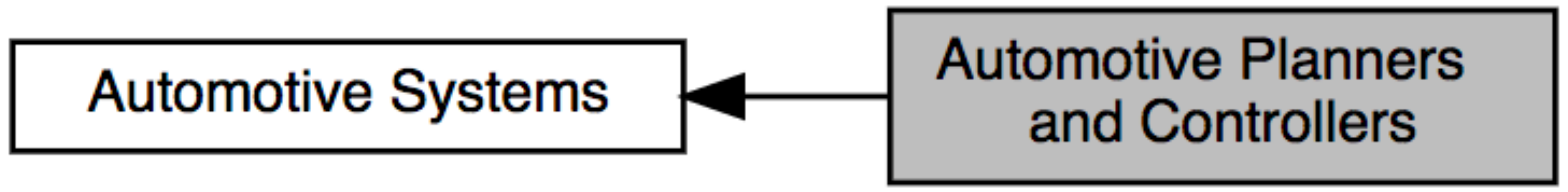
Automotive Planners and Controllers

Classes

Automotive Planners and Controllers

Modeling Dynamical Systems » Automotive Systems

Collaboration diagram for Automotive Planners and Controllers:



Classes

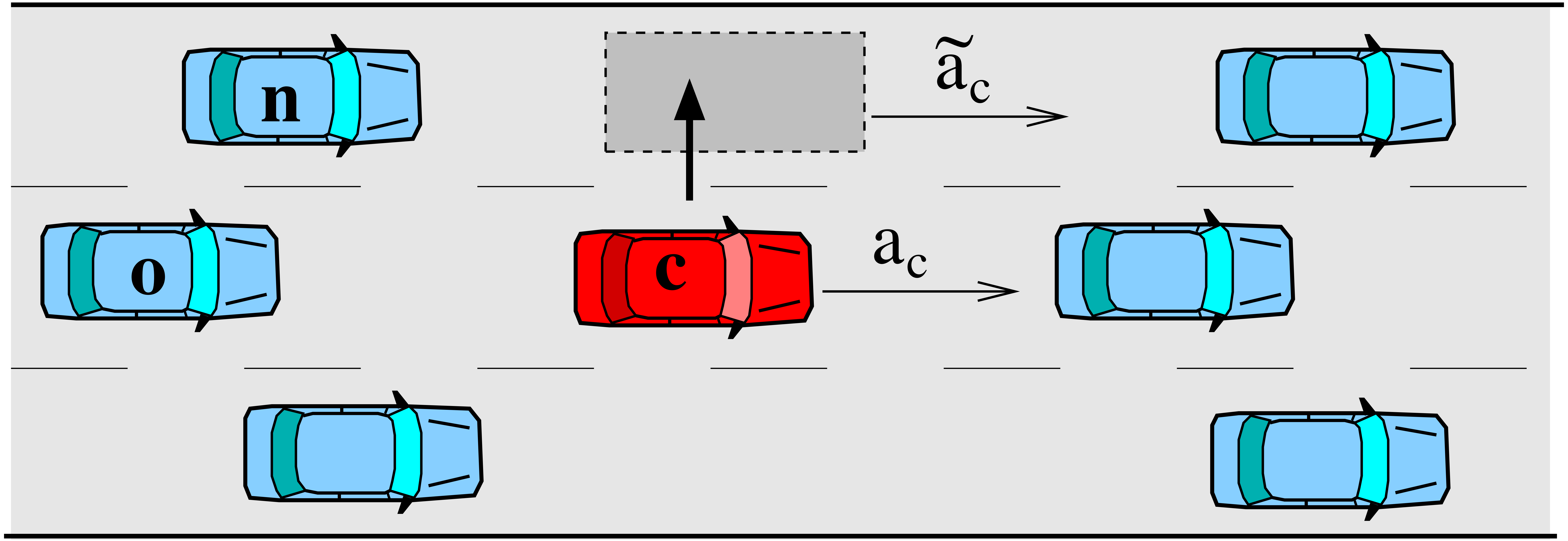
class **IdmController< T >**
IdmController implements the IDM (Intelligent Driver Model) planner, computed based only on the nearest car ahead. More...

class **MobilPlanner< T >**
MOBIL (Minimizing Overall Braking Induced by Lane Changes) [1] is a planner that minimizes braking requirement for the ego car while also minimizing (per a weighting factor) the braking requirements of any trailing cars within the ego car's immediate neighborhood. More...

class **PurePursuitController< T >**
PurePursuitController implements a pure pursuit controller. More...

Automotive Planners and Controllers

MOBIL (Minimizing Overall Braking Induced by Lane Changes)



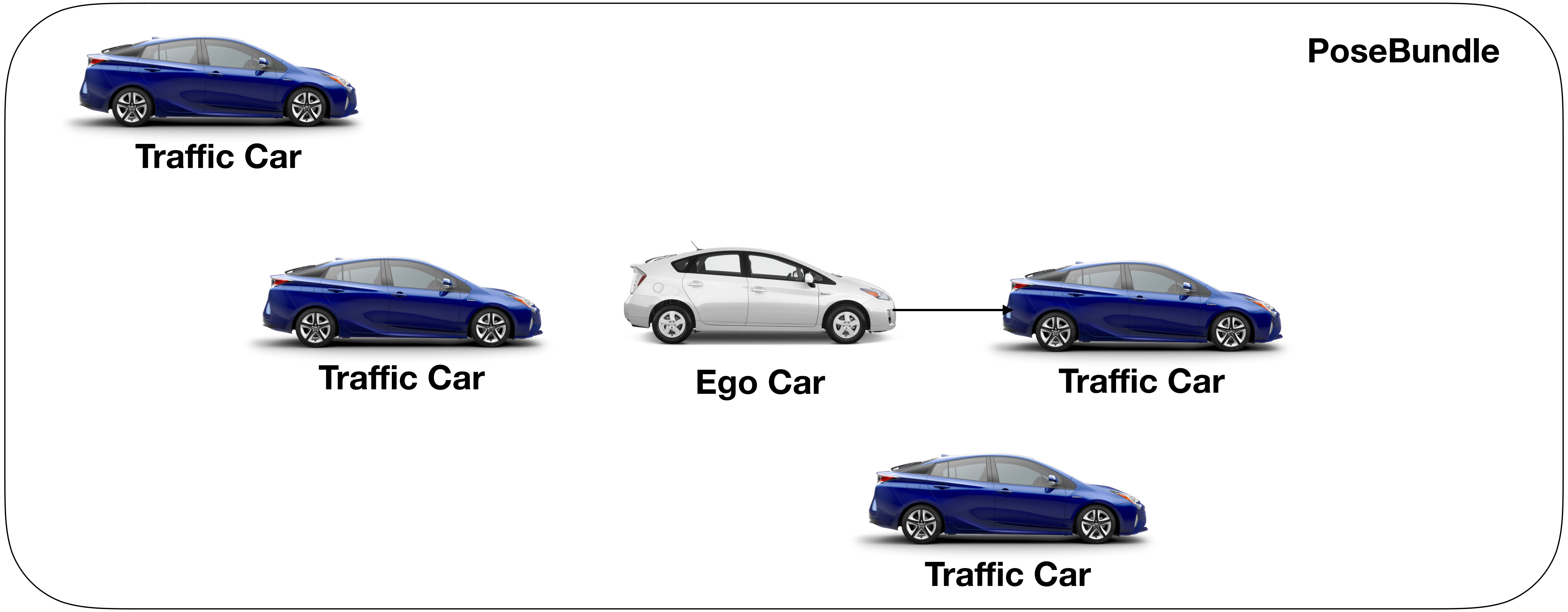
Minimizing Breaking Requirement:

- **for the Ego Car**
- **of Any Trailing Cars within the Ego Car's Neighborhood**

MobilPlanner

Input

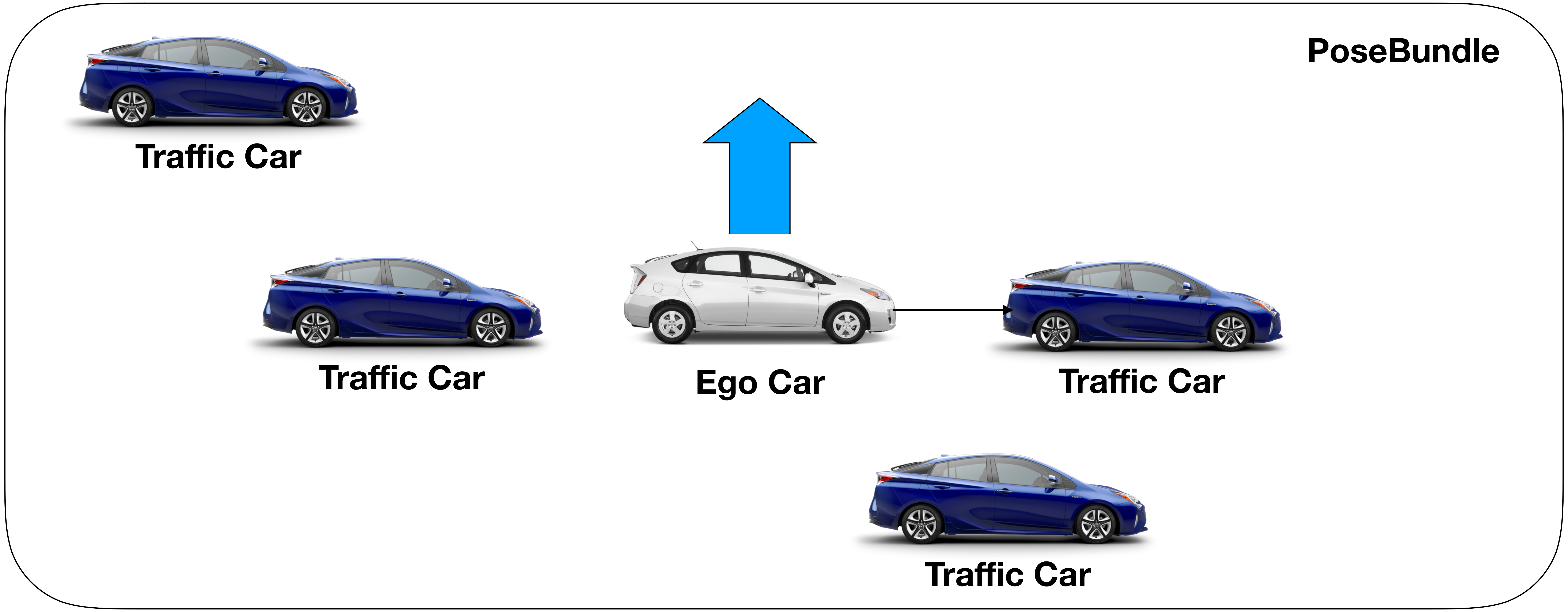
1. **PoseVector for the Ego Car**
2. **FrameVelocity for the Ego Car**
3. **Ego Car's Commanded Acceleration**
4. **PoseBundle for the traffic cars**



MobilPlanner

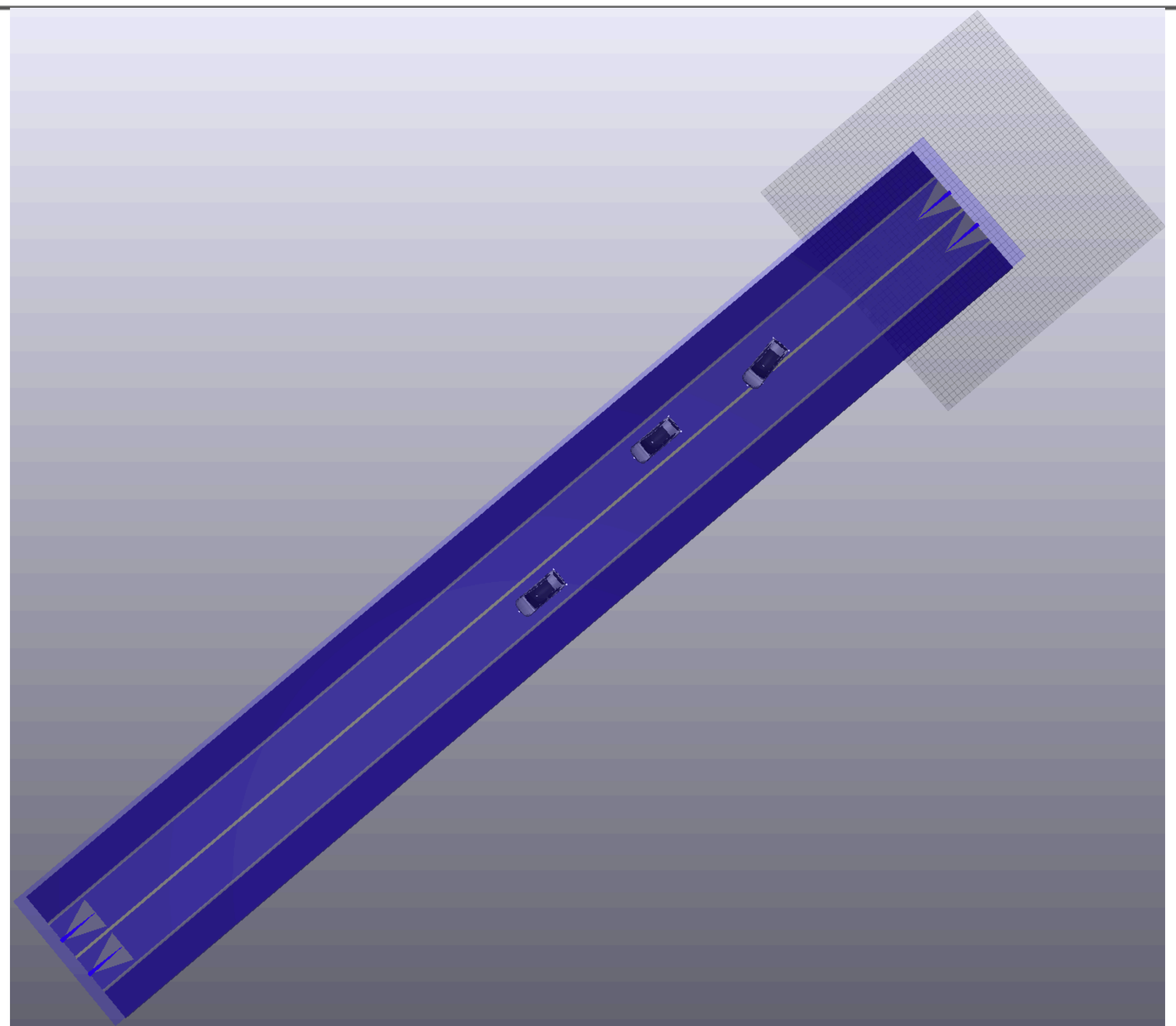
Output

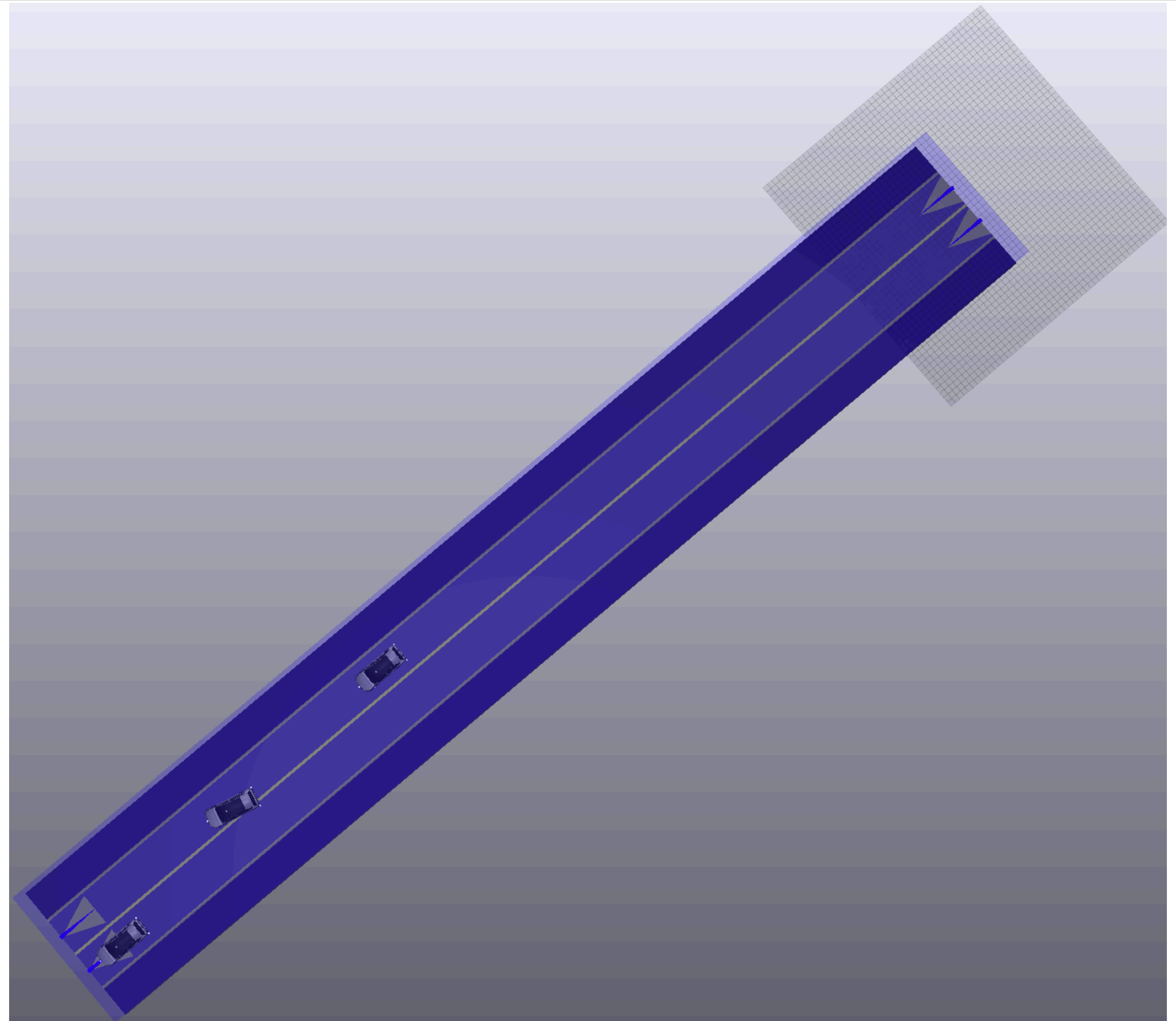
1. **LaneDirection**



https://github.com/RobotLocomotion/drake/blob/master/automotive/mobil_planner.h

Automotive Demo

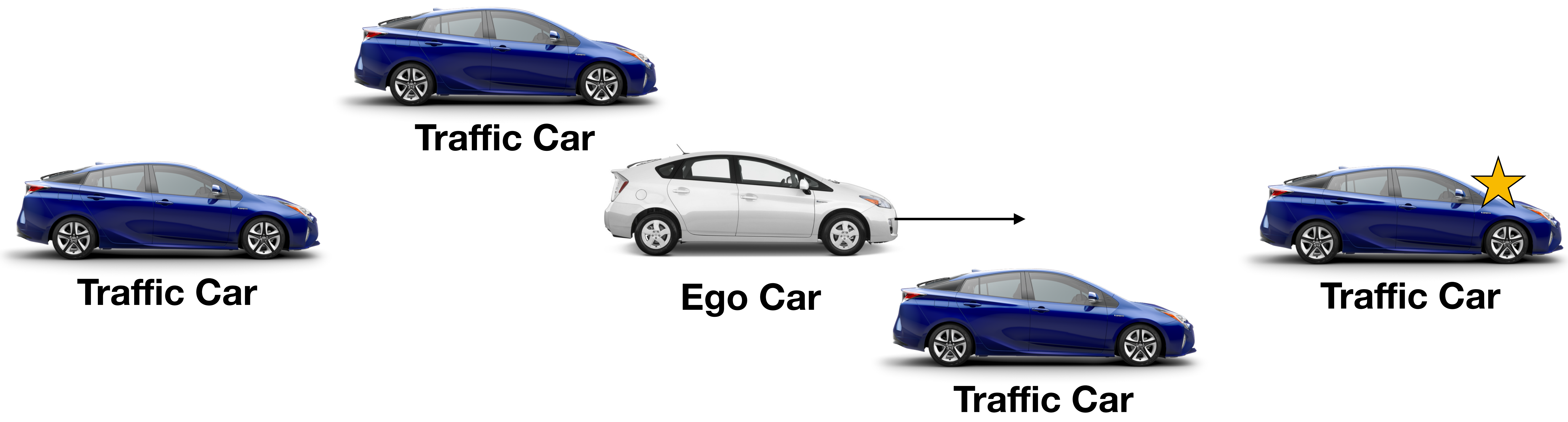




Q: Sensors?

```
template <typename T>
void IdmController<T>::ImplCalcAcceleration(
    const PoseVector<T>& ego_pose, const FrameVelocity<T>& ego_velocity,
    const PoseBundle<T>& traffic_poses,
    const IdmPlannerParameters<T>& idm_params,
    const RoadPosition& ego_rp,
    systems::BasicVector<T>* command) const {
 RoadPosition ego_position = ego_rp;
  if (!ego_rp.lane) {
    const auto gp =
        GeoPositionT<T>::FromXyz(ego_pose.get_isometry().translation());
    ego_position =
        road_.ToRoadPosition(gp.MakeDouble(), nullptr, nullptr, nullptr);
  }

  // Find the single closest car ahead.
  const ClosestPose<T> lead_car_pose = PoseSelector<T>::FindSingleClosestPose(
      ego_position.lane, ego_pose, traffic_poses,
      idm_params.scan_ahead_distance(), AheadOrBehind::kAhead,
      path_or_branches_);
```

```
  const T headway_distance = lead_car_pose.distance;
  const LanePositionT<T> lane_position(T(ego_position.pos.s()),
                                       T(ego_position.pos.r()),
                                       T(ego_position.pos.h()));
  const T s_dot_ego = PoseSelector<T>::GetSigmaVelocity(
      {ego_position.lane, lane_position, ego_velocity});
  const T s_dot_lead =
      (abs(lead_car_pose.odometry.pos.s()) ==
       std::numeric_limits<T>::infinity())
          ? T(0.)
          : PoseSelector<T>::GetSigmaVelocity(lead_car_pose.odometry);

  // Saturate the net_distance at `idm_params.distance_lower_limit()` away from
  // the ego car to avoid near-singular solutions inherent to the IDM equation.
  const T actual_headway = headway_distance - idm_params.bloat_diameter();
  const T net_distance = max(actual_headway, idm_params.distance_lower_limit());
  const T closing_velocity = s_dot_ego - s_dot_lead;

  // Compute the acceleration command from the IDM equation.
  (*command)[0] = IdmPlanner<T>::Evaluate(idm_params, s_dot_ego, net_distance,
                                          closing_velocity);
}
```



Q: Python Bindings?

Currently, we expose a subset of C++ APIs:

- **SimpleCar**
- **IdmController**
- **PurePursuitController**
- **DrivingCommand**
- **...**

https://github.com/RobotLocomotion/drake/blob/master/bindings/pydrake/automotive_py.cc

More Questions?

The following people at TRI helped me make this presentation. Thank you all!

- **Alejandro Castro**
- **Jonathan Decastro**
- **Evan Drumwright**
- **Liang Fok**
- **Naveen Kuppuswamy**
- **Michael Sherman**
- **Prof. Russ Tedrake**

Some Tips

Find a bug? Have a feature-request?

-> File an issue at https://github.com/RobotLocomotion/drake/issues/new

Questions?

-> Ask a question at https://stackoverflow.com/questions/ask with ***drake*** tag.