Model Predictive Control

Generated by Doxygen 1.8.1.2

Sun Jun 15 2014 00:10:17

Contents

Chapter 1

lbmpc_qpoases

This is an implementation of Learning-Based Model Predictive Control (LBMPC) that uses the qpOASES dense solver.

Prerequisites

- qpOASES
- Eigen3
- CMake

Compiling examples

```
cd lbmpc_qpoases
mkdir build
cd build
cmake ..
export N_MPC_STEPS=15 # or whatever..
make
```

Creating data files

Example from documentation

(in PYTHON):

```
>> cd model-predictive-control/mpc/script
>> python lbmpc_control_design.py
```

Prerequisites

- python-control
- Slycot
- PyYAML

- CVXOPT for ATLAS installation: (http://sciruby.com/docs/installation/atlas.html)

Quadrotor example

(in MATLAB):

```
>> cd lbmpc/matlab/qr_example
>> Init
```

How to run examples

Example from documentation

cd lbmpc_lssol build/bin/example0 matlab/example0/ConstrParam.bin

Quadrotor example

cd lbmpc_lssol build/bin/qr_example matlab/qr_example/quad.bin

Back to home

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

Chapter 3

Namespace Index

3.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 4

Class Index

4.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 5

Class Index

5.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

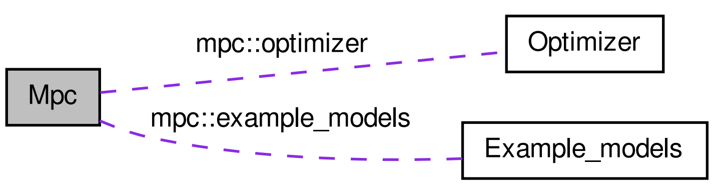
To solve each of these optimal control problems the function mpc::LBMPC::initMPC initialized the control problem.
The resulting optimization problem is then solved by a (predefined) minimization routine.
Then the first value of the computed control is implemented and the optimization horizon is shifted forward in time.
This allows the procedure to be applied iteratively and computes a (suboptimal) infinite horizon control.
Note that the function mpc::LBMPC::updatedMPC can be used to computer a control signal for the next time-step.

Chapter 6

Module Documentation

6.1 Mpc

Collaboration diagram for Mpc:



Namespaces

- namespace mpc

  Model Predictives Control interfaces and implementations.
- namespace mpc::optimizer

  Optimizer interfaces and implementations.

Classes

- class mpc::example_models::TanksSystemSimulator

  This class provides methods to simulate a example model of tanks system defined by a class mpc::model::Model object

$$\dot{x}(t) = Ax(t) + Bu(t)$$

$$y(t) = Cx(t)$$

  on a fixed prediction horizon interval $[t_0, t_N]$ *with initial value* $x(t_0, x_0) = x_0$ *and given control* $u(\cdot, x_0)$. *That is, for a given class mpc::model::Model object and a given control* u *the simulator can solve the differential or difference equation forward in time.*

- class mpc::ModelPredictiveControl

  This class serves as a base class in order to expand the functionality of the library and implement different sorts of MPC algorithms. The methods defined here are conceived in the simplest way possible to allow different implementations in the derived classes.

  .

- class mpc::STDMPC

  Class for solving the explicit model predictive control problem
  The aim of this class is to solve the explicit model predictive control of the following form:

$$\begin{aligned}
\min_{\mathbf{u}_{k_0},\cdots,\mathbf{u}_{k_0+N-1}} J_N(x,u) &= \frac{1}{2}(\mathbf{x}_{k_0+N}-\mathbf{x}_{ref})^T\mathbf{P}(\mathbf{x}_{k_0+N}-\mathbf{x}_{ref})+\frac{1}{2}\sum_{k=k_0}^{k_0+N-1}(\mathbf{x}_k-\mathbf{x}_{ref})^T\mathbf{Q}(\mathbf{x}_k-\mathbf{x}_{ref})+(\mathbf{u}_k-\mathbf{u}_{ref})^T\mathbf{R}(\mathbf{u}_k-\mathbf{u}_{ref}) \\
\mathbf{x}_{k_0} &= \omega_0(k_0) \\
\mathbf{x}_{k+1} &= \mathbf{A}\mathbf{x}_k+\mathbf{B}\mathbf{u}_k \qquad \forall k\in[k_0,N] \\
\bar{x} &\leq \mathbf{M}\mathbf{x}_k \qquad \forall k\in[k_0,N] \\
\bar{u} &\leq \mathbf{N}\mathbf{u}_k \qquad \forall k\in[k_0,N]
\end{aligned}$$

  To solve each of these optimal control problems the function mpc::STDMPC::initMPC initialized the control problem. The resulting optimization problem is then solved by a (predefined) minimization routine.
  Then the first value of the computed control is implemented and the optimization horizon is shifted forward in time. This allows the procedure to be applied iteratively and computes a (suboptimal) infinite horizon control.
  Note that the function mpc::STDMPC::updateMPC() can be used to compute a control signal for the next time-step.

  .

- class mpc::optimizer::qpOASES

  Class to interface the qpOASES library This class gives an interface with qpOASES library in order to implement a quadratic program using online active set strategy for MPC controller. qpOASES solve a convex optimization class of the following form

$$\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x}+\mathbf{x}^T\mathbf{g}(\mathbf{x_0})$$

  suject to

$$\begin{aligned}
lbG(\mathbf{x_0}) \leq \quad &\mathbf{G}\mathbf{x} &\leq ubG(\mathbf{x_0}) \\
lb(\mathbf{x_0}) \leq \quad &\mathbf{x} &\leq ub(\mathbf{x_0})
\end{aligned}$$

  .

Functions

- mpc::example_models::TanksSystemSimulator::TanksSystemSimulator ()

  Constructor function.

- mpc::example_models::TanksSystemSimulator::∼TanksSystemSimulator ()

  Destructor function.

- double ∗ mpc::example_models::TanksSystemSimulator::simulatePlant (double ∗state_vect, double ∗input_vect, double sampling_time)

  Function used to simulate the specified plant.

- mpc::ModelPredictiveControl::ModelPredictiveControl ()

  Constructor function.

- mpc::ModelPredictiveControl::∼ModelPredictiveControl ()

  Destructor function.

- virtual bool mpc::ModelPredictiveControl::resetMPC (mpc::model::Model ∗model, mpc::optimizer::Optimizer ∗optimizer, mpc::model::Simulator ∗simulator)=0

Function to specify and set the settings of all the components within the MPC problem. The mpc::ModelPredictiveControl class can change individual parts of the MPC problem; such as the model (mpc::model::Model and derived classes), the optimizer (mpc::optimizer::Optimizer and derived classes) and, if used, the plant simulator (mpc::model::Simulator and derived classes) in order to allow different combinations of these parts when solving.

- virtual bool mpc::ModelPredictiveControl::initMPC ()=0

  Function to initialize the calculation of the MPC algorithm. The function reads all required parameters from ROS' parameter server that has been previously loaded from a configuration YAML file, and performs all the initial calculations of variables to be used in the optimization problem.

- virtual void mpc::ModelPredictiveControl::updateMPC (double ∗x_measured, double ∗x_reference)=0

  Function to update the MPC algorithm for the next iteration. The parameters defined and calculated in mpc::ModelPredictiveControl::initMPC() are used together with the methods taken from the MPC class components (mpc::model::Model, mpc::optimizer::Optimizer and mpc::model::Simulator) to find a solution to the optimization problem. This is where the different variants of MPC algorithms can be implemented in a source file from a derived class.

- virtual double ∗ mpc::ModelPredictiveControl::getControlSignal () const

  Function to get the control signal generated for the MPC. As the MPC algorithm states, the optimization process yields the control signals for a range of times defined by the prediction horizon, but only the current control signal is applied to the plant. This function returns the control signal for the current time.

- virtual void mpc::ModelPredictiveControl::writeToDisc ()

  Function to write the data of the MPC in a text file.

- mpc::STDMPC::STDMPC (ros::NodeHandle node_handle)

  Constructor function.

- mpc::STDMPC::∼STDMPC ()

  Destructor function.

- virtual bool mpc::STDMPC::resetMPC (mpc::model::Model ∗model, mpc::optimizer::Optimizer ∗optimizer, mpc::model::Simulator ∗simulator)

  Function to specify and set the settings of all the components within the MPC problem. The mpc::ModelPredictiveControl class can change individual parts of the MPC problem; such as the model (mpc::model::Model and derived classes), the optimizer (mpc::optimizer::Optimizer and derived classes) and, if used, the plant simulator (mpc::model::Simulator and derived classes) in order to allow different combinations of these parts when solving.

- virtual bool mpc::STDMPC::initMPC ()

  Function to initialize the calculation of the MPC algorithm. The function reads all required parameters from ROS' parameter server that has been previously loaded from a configuration YAML file, and performs all the initial calculations of variables to be used in the optimization problem.

- virtual void mpc::STDMPC::updateMPC (double ∗x_measured, double ∗x_reference)

  Function to solve the optimization problem formulated in the MPC.

- mpc::optimizer::qpOASES::qpOASES (ros::NodeHandle node_handle)

  Constructor function.

- mpc::optimizer::qpOASES::∼qpOASES ()

  Destructor function.

- virtual bool mpc::optimizer::qpOASES::init ()

  Function to define the initialization of qpOASES optimizer.

- virtual bool mpc::optimizer::qpOASES::computeOpt (double ∗H, double ∗g, double ∗G, double ∗lb, double ∗ub, double ∗lbG, double ∗ubG, double cputime)

  Function to solve the optimization problem formulated in the MPC.

- double ∗ mpc::optimizer::qpOASES::getOptimalSolution ()

  Get the vector of optimal solutions calculated by qpOASES.

Variables

- mpc::model::Model ∗ mpc::ModelPredictiveControl::model_

  Pointer of dynamic model of the system.
- mpc::model::Simulator ∗ mpc::ModelPredictiveControl::simulator_

  Pointer of the simulator of the system.
- mpc::optimizer::Optimizer ∗ mpc::ModelPredictiveControl::optimizer_

  Pointer of the optimizer of the MPC.
- int mpc::ModelPredictiveControl::states_

  Number of states of the dynamic model.
- int mpc::ModelPredictiveControl::inputs_

  Number of inputs of the dynamic model.
- int mpc::ModelPredictiveControl::outputs_

  Number of outputs of the dynamic model.
- int mpc::ModelPredictiveControl::horizon_

  Horizon of prediction of the dynamic model.
- int mpc::ModelPredictiveControl::variables_

  Number of variables, i.e inputs ∗ horizon.
- int mpc::ModelPredictiveControl::constraints_

  Number of constraints.
- double ∗ mpc::ModelPredictiveControl::operation_states_

  Vector of the operation points for the states in case of a LTI model.
- double ∗ mpc::ModelPredictiveControl::operation_inputs_

  Vector of the operation points for the inputs in case of a LTI model.
- int mpc::ModelPredictiveControl::infeasibility_counter_

  Infeasibility counter in the solution.
- int mpc::ModelPredictiveControl::infeasibility_hack_counter_max_

  Maximun value of the infeasibility counter.
- Eigen::MatrixXd mpc::ModelPredictiveControl::Q_

  States error weight matrix.
- Eigen::MatrixXd mpc::ModelPredictiveControl::P_

  Terminal states error weight matrix.
- Eigen::MatrixXd mpc::ModelPredictiveControl::R_

  Input error weight matrix.
- double ∗ mpc::ModelPredictiveControl::mpc_solution_

  Vector of the MPC solution.
- Eigen::MatrixXd mpc::ModelPredictiveControl::u_reference_

  Stationary control signal for the reference state vector.
- double ∗ mpc::ModelPredictiveControl::control_signal_

  Control signal computes for MPC.
- std::vector< int > mpc::ModelPredictiveControl::t_

  Data of the time vector of the system.
- std::vector< std::vector
  < double > > mpc::ModelPredictiveControl::x_

  Data of the state vector of the system.
- std::vector< std::vector
  < double > > mpc::ModelPredictiveControl::xref_

  Data of the reference state vector of the system.

- std::vector< std::vector
  < double > > mpc::ModelPredictiveControl::u_

  Data of the control signal vector of the system.
- std::string mpc::ModelPredictiveControl::path_name_

  Path where the data will be save.
- std::string mpc::ModelPredictiveControl::data_name_

  Name of the file where the data will be save.
- bool mpc::ModelPredictiveControl::enable_record_

  Label that indicates if it will be save the data.
- double ∗ mpc::optimizer::qpOASES::optimal_solution_

  Optimal solution obtained with the implementation of qpOASES.

6.1.1 Detailed Description

6.1.2 Function Documentation

6.1.2.1 bool mpc::optimizer::qpOASES::computeOpt (double ∗ *H*, double ∗ *g*, double ∗ *G*, double ∗ *lb*, double ∗ *ub*, double ∗ *lbG*, double ∗ *ubG*, double *cputime*) [virtual]

Function to solve the optimization problem formulated in the MPC.

Parameters

double∗	H Hessian matrix
double∗	g Gradient vector
double∗	G Constraint matrix
double∗	lb Low bound vector
double∗	ub Upper bound vector
double∗	lbG Low constraint vector
double∗	lbG Upper constraint vector
double	cputime CPU-time for computing the optimization

Returns

bool Label that indicates if the computation of the optimization is successful

Implements mpc::optimizer::Optimizer.

Definition at line 56 of file qpOASES.cpp.

```
{
        // solve first QP.
        int nWSR = nWSR_;
        double cpu_time;

        returnValue retval;
        if (!qpOASES_initialized_) {
                cpu_time = cputime;
                retval = solver_->init(H, g, G, lb, ub, lbG, ubG, nWSR, &
      cpu_time);
                if (retval == SUCCESSFUL_RETURN) {
                        ROS_INFO("qpOASES problem successfully initialized.");
                        qpOASES_initialized_ = true;
                }
        }
        else {
                cpu_time = cputime;
```

```
                retval = solver_->hotstart(H, g, G, lb, ub, lbG, ubG, nWSR, &
      cpu_time);
        }
        if (solver_->isInfeasible())
                ROS_WARN("The quadratic programming is infeasible.");

        if (retval == SUCCESSFUL_RETURN) {
                solver_->getPrimalSolution(optimal_solution_);
        }
        else if (retval == RET_MAX_NWSR_REACHED) {
                ROS_WARN("The QP couldn't solve because the maximun number of
       WSR was reached.");
                return false;
        }
        else {
                ROS_WARN("The QP couldn't find the solution.");
                return false;
        }

        //std::cout << "cputime = " << cpu_time << std::endl;
        return true;
}
```

6.1.2.2 double * mpc::ModelPredictiveControl::getControlSignal ( ) const [inline], [virtual]

Function to get the control signal generated for the MPC. As the MPC algorithm states, the optimization process yields the control signals for a range of times defined by the prediction horizon, but only the current control signal is applied to the plant. This function returns the control signal for the current time.

Returns

double∗ Control signal for the current MPC iteration.

Definition at line 158 of file model_predictive_control.h.

```
{
        for (int i = 0; i < inputs_; i++) {
                if (infeasibility_counter_ <
      infeasibility_hack_counter_max_)
                        control_signal_[i] = mpc_solution_
      [infeasibility_counter_ * inputs_ + i];
                else
                        control_signal_[i] = u_reference_
      (i);
        }

        return control_signal_;
}
```

6.1.2.3 double * mpc::optimizer::qpOASES::getOptimalSolution ( ) [virtual]

Get the vector of optimal solutions calculated by qpOASES.

Returns

double∗ Optimal solution

Implements mpc::optimizer::Optimizer.

Definition at line 95 of file qpOASES.cpp.

```
{
        return optimal_solution_;
}
```

6.1.2.4 bool mpc::optimizer::qpOASES::init ( ) [virtual]

Function to define the initialization of qpOASES optimizer.

Returns

bool Label that indicates if the initialization of the optimizer is successful

Implements mpc::optimizer::Optimizer.

Definition at line 26 of file qpOASES.cpp.

```
{
        // reading the parameters required for the solver
        if (nh_.getParam("/mpc/optimizer/number_constraints", constraints_
      )) {
                ROS_INFO("Got param: number of constraints = %d", constraints_
      );
        }

        nh_.param<int>("/mpc/optimizer/working_set_recalculations", nWSR_, 10);
        ROS_INFO("Got param: number of working set recalculations = %d", nWSR_)
      ;

        if (variables_ == 0 || constraints_ == 0 ||
      horizon_ == 0)
                return false;

        qpOASES_initialized_ = false;
        solver_ = new SQProblem(variables_, constraints_
      * horizon_);
        Options myOptions;
        myOptions.setToReliable();
//      myOptions.setToMPC();
//      myOptions.enableFlippingBounds = BT_TRUE;
//      myOptions.printLevel = PL_LOW;
        solver_->setOptions(myOptions);

        optimal_solution_ = new double[variables_];


        ROS_INFO("qpOASES solver class successfully initialized.");
        return true;
}
```

6.1.2.5 virtual bool mpc::ModelPredictiveControl::initMPC ( ) [pure virtual]

Function to initialize the calculation of the MPC algorithm. The function reads all required parameters from ROS' parameter server that has been previously loaded from a configuration YAML file, and performs all the initial calculations of variables to be used in the optimization problem.

Returns

Label that indicates if the MPC is initialized with success

Implemented in mpc::STDMPC, and mpc::LBMPC.

6.1.2.6 bool mpc::STDMPC::initMPC ( ) [virtual]

Function to initialize the calculation of the MPC algorithm. The function reads all required parameters from ROS' parameter server that has been previously loaded from a configuration YAML file, and performs all the initial calculations of variables to be used in the optimization problem.

Returns

Label that indicates if the MPC is initialized with success

Implements mpc::ModelPredictiveControl.

Definition at line 68 of file stdmpc.cpp.

```
{
        // Initialization of MPC solution
        mpc_solution_ = new double[variables_];
        control_signal_ = new double[inputs_];
        operation_states_ = new double[states_];
        operation_inputs_ = new double[inputs_];


        u_reference_ = Eigen::MatrixXd::Zero(inputs_, 1);
        infeasibility_counter_ = 0;
        x_.resize(states_);
        xref_.resize(states_);
        u_.resize(inputs_);


        // Initialization of state space matrices
        A_ = Eigen::MatrixXd::Zero(states_, states_);
        B_ = Eigen::MatrixXd::Zero(states_, inputs_);
        C_ = Eigen::MatrixXd::Zero(outputs_, states_);

        // Obtention of the model parameters
        if (model_->computeLinearSystem(A_, B_)) {
                ROS_INFO("Model calculated successfully.");
                std::cout << "A\n" << A_ << std::endl;
                std::cout << "B\n" << B_ << std::endl;
        }

        // Reading the weight matrices of the cost function
        Q_ = Eigen::MatrixXd::Zero(states_, states_);
        P_ = Eigen::MatrixXd::Zero(states_, states_);
        R_ = Eigen::MatrixXd::Zero(inputs_, inputs_);
        XmlRpc::XmlRpcValue Q_list, P_list, R_list;
        nh_.getParam("/mpc/optimizer/states_error_weight_matrix/data", Q_list);
        ROS_ASSERT(Q_list.getType() == XmlRpc::XmlRpcValue::TypeArray);
        ROS_ASSERT(Q_list.size() == states_ * states_);

        nh_.getParam("/mpc/optimizer/terminal_state_weight_matrix/data", P_list
);
        ROS_ASSERT(P_list.getType() == XmlRpc::XmlRpcValue::TypeArray);
        ROS_ASSERT(P_list.size() == states_ * states_);

        int z = 0;
        for (int i = 0; i < states_; i++) {
                for (int j = 0; j < states_; j++) {
                        ROS_ASSERT(Q_list[z].getType() ==
XmlRpc::XmlRpcValue::TypeDouble);
                        Q_(i, j) = static_cast<double>(Q_list[z]);

                        ROS_ASSERT(P_list[z].getType() ==
XmlRpc::XmlRpcValue::TypeDouble);
                        P_(i, j) = static_cast<double>(P_list[z]);
                        z++;
                }
        }

        nh_.getParam("/mpc/optimizer/input_error_weight_matrix/data", R_list);
        ROS_ASSERT(R_list.getType() == XmlRpc::XmlRpcValue::TypeArray);
        ROS_ASSERT(R_list.size() == inputs_ * inputs_);
        z = 0;
        for (int i = 0; i < inputs_; i++) {
                for (int j = 0; j < inputs_; j++) {
                        ROS_ASSERT(R_list[z].getType() ==
XmlRpc::XmlRpcValue::TypeDouble);
                        R_(i, j) = static_cast<double>(R_list[z]);
                        z++;
                }
        }


        // Creation of the states and inputs weight matrices for the quadratic
 program
```

```
  Q_bar_ = Eigen::MatrixXd::Zero((horizon_ + 1) * states_, (
horizon_ + 1) * states_);
  R_bar_ = Eigen::MatrixXd::Zero(horizon_ * inputs_, horizon_
 * inputs_);
  for (int i = 0; i < horizon_; i++) {
          Q_bar_.block(i * states_, i * states_, states_, states_) = Q_
;
          R_bar_.block(i * inputs_, i * inputs_, inputs_, inputs_) = R_
;
  }
  Q_bar_.block(horizon_ * states_, horizon_ * states_, states_, states_)
= P_;


  // Reading the constraint vectors
  Eigen::VectorXd lbG = Eigen::VectorXd::Zero(constraints_);
  Eigen::VectorXd ubG = Eigen::VectorXd::Zero(constraints_);
  XmlRpc::XmlRpcValue lbG_list, ubG_list;
  nh_.getParam("/mpc/optimizer/constraints/constraint_vector_low",
lbG_list);
  ROS_ASSERT(lbG_list.getType() == XmlRpc::XmlRpcValue::TypeArray);
  nh_.getParam("/mpc/optimizer/constraints/constraint_vector_upp",
ubG_list);
  ROS_ASSERT(ubG_list.getType() == XmlRpc::XmlRpcValue::TypeArray);
  if (ubG_list.size() == lbG_list.size()) {
          for (int i = 0; i < ubG_list.size(); ++i) {
                  ROS_ASSERT(lbG_list[i].getType() ==
XmlRpc::XmlRpcValue::TypeDouble);
                  lbG(i) = static_cast<double>(lbG_list[i]);

                  ROS_ASSERT(ubG_list[i].getType() ==
XmlRpc::XmlRpcValue::TypeDouble);
                  ubG(i) = static_cast<double>(ubG_list[i]);
          }

  }

  // Reading the bound vectors
  Eigen::VectorXd lb = Eigen::VectorXd::Zero(inputs_);
  Eigen::VectorXd ub = Eigen::VectorXd::Zero(inputs_);
  XmlRpc::XmlRpcValue lb_list, ub_list;
  nh_.getParam("/mpc/optimizer/constraints/bound_vector_low", lb_list);
  ROS_ASSERT(lb_list.getType() == XmlRpc::XmlRpcValue::TypeArray);
  nh_.getParam("/mpc/optimizer/constraints/bound_vector_upp", ub_list);
  ROS_ASSERT(ub_list.getType() == XmlRpc::XmlRpcValue::TypeArray);
  if (ub_list.size() == lb_list.size()) {
          for (int i = 0; i < ub_list.size(); ++i) {
                  ROS_ASSERT(lb_list[i].getType() ==
XmlRpc::XmlRpcValue::TypeDouble);
                  lb(i) = static_cast<double>(lb_list[i]);


                  ROS_ASSERT(ub_list[i].getType() ==
XmlRpc::XmlRpcValue::TypeDouble);
                  ub(i) = static_cast<double>(ub_list[i]);
          }
  }

  // Reading the state bound matrix
  Eigen::MatrixXd M = Eigen::MatrixXd::Zero(constraints_,
states_);
  XmlRpc::XmlRpcValue M_list;
  nh_.getParam("/mpc/optimizer/constraints/constraint_matrix_M", M_list);
  ROS_ASSERT(M_list.getType() == XmlRpc::XmlRpcValue::TypeArray);
  ROS_ASSERT(M_list.size() == constraints_ * states_);

  z = 0;
  for (int i = 0; i < constraints_; ++i) {
          for (int j = 0; j < states_; j++) {
                  ROS_ASSERT(M_list[i].getType() ==
XmlRpc::XmlRpcValue::TypeDouble);
                  M(i, j) = static_cast<double>(M_list[z]);
                  z++;
          }
  }


  // Creation of the extended constraint and bound vector
  lbG_bar_ = Eigen::VectorXd::Zero(constraints_ * horizon_);
  ubG_bar_ = Eigen::VectorXd::Zero(constraints_ * horizon_);
  lb_bar_ = Eigen::VectorXd::Zero(inputs_ * horizon_);
```

```
        ub_bar_ = Eigen::VectorXd::Zero(inputs_ * horizon_);
        for (int i = 0; i < horizon_; i++) {
                lbG_bar_.block(i * constraints_, 0, constraints_, 1) = lbG;
                ubG_bar_.block(i * constraints_, 0, constraints_, 1) = ubG;
                lb_bar_.block(i * inputs_, 0, inputs_, 1) = lb;
                ub_bar_.block(i * inputs_, 0, inputs_, 1) = ub;
        }

        // Creation of the extended constraint matrix G_bar_
        M_bar_ = Eigen::MatrixXd::Zero(constraints_ * horizon_, (horizon_ + 1)
      * states_);
        for (int i = 0; i < horizon_; i++) {
                for (int j = 0; j < horizon_ + 1; j++) {
                        if (i == j) {
                                M_bar_.block(i * constraints_, j * states_,
      constraints_, states_) = M;
                        }
                }
        }


        ROS_INFO("STDMPC class successfully initialized.");
        return true;
}
```

6.1.2.7 virtual bool mpc::ModelPredictiveControl::resetMPC (mpc::model::Model ∗ *model,* mpc::optimizer::Optimizer ∗ *optimizer,* mpc::model::Simulator ∗ *simulator*) [pure virtual]

Function to specify and set the settings of all the components within the MPC problem. The mpc::ModelPredictiveControl class can change individual parts of the MPC problem; such as the model (mpc::model::Model and derived classes), the optimizer (mpc::optimizer::Optimizer and derived classes) and, if used, the plant simulator (mpc::model::Simulator and derived classes) in order to allow different combinations of these parts when solving.

Parameters

mpc::model::-Model	∗model Pointer to the model of the plant to be used in the algorithm
mpc::optimizer::-Optimizer	∗optimizer Pointer to the optimization library to be used in the algorithm
mpc::model::-Simulator	∗simulator Pointer to the simulator class used to predict the states

Implemented in mpc::STDMPC, and mpc::LBMPC.

6.1.2.8 bool mpc::STDMPC::resetMPC (mpc::model::Model ∗ *model,* mpc::optimizer::Optimizer ∗ *optimizer,* mpc::model::Simulator ∗ *simulator*) [virtual]

Function to specify and set the settings of all the components within the MPC problem. The mpc::ModelPredictiveControl class can change individual parts of the MPC problem; such as the model (mpc::model::Model and derived classes), the optimizer (mpc::optimizer::Optimizer and derived classes) and, if used, the plant simulator (mpc::model::Simulator and derived classes) in order to allow different combinations of these parts when solving.

Parameters

mpc::model::-Model	∗model Pointer to the model of the plant to be used in the algorithm
mpc::optimizer::-Optimizer	∗optimizer Pointer to the optimization library to be used in the algorithm
mpc::model::-Simulator	∗simulator Pointer to the simulator class used to predict the states

Implements mpc::ModelPredictiveControl.

Definition at line 19 of file stdmpc.cpp.

```
{
        // Setting of the pointer of the model, optimizer and simulator classes
        model_ = model;
        optimizer_ = optimizer;
        simulator_ = simulator;
        time_index_ = 0;

        // Reading of the horizon value of the model predictive control
       algorithm
        nh_.param<int>("/mpc/horizon", horizon_, 30);
        ROS_INFO("Got param: horizon = %d", horizon_);

        nh_.param<int>("/mpc/infeasibility_hack_counter_max",
      infeasibility_hack_counter_max_, 1);
        ROS_INFO("Got param: infeasibility_hack_counter_max = %d",
      infeasibility_hack_counter_max_);


        // Reading the path and data name
        if (!nh_.getParam("/mpc/path_name", path_name_)) {
                ROS_WARN("The data will not save because could not found path
       name from parameter server.");
                enable_record_ = false;
        }
        if (!nh_.getParam("/mpc/data_name", data_name_)) {
                ROS_WARN("The data will not save because could not found data
       name from parameter server.");
                enable_record_ = false;
        }


        // Reading of the problem variables
        states_ = model_->getStatesNumber();
        inputs_ = model_->getInputsNumber();
        outputs_ = model_->getOutputsNumber();

        variables_ = horizon_ * inputs_;
        optimizer_->setHorizon(horizon_);
        optimizer_->setVariableNumber(variables_
      );

        if (!optimizer_->init()) {
                ROS_INFO("Could not initialize the optimizer class.");
                return false;
        }
        constraints_ = optimizer_->getConstraintNumber
      ();


        ROS_INFO("Reset successful. States = %d \n Inputs = %d \n Outputs = %d
      \n Constraints = %d \n", states_,inputs_,outputs_,constraints_
      );
        return true;
}
```

6.1.2.9 double ∗ mpc::example_models::TanksSystemSimulator::simulatePlant (double ∗ *state_vect,* double ∗ *input_vect,* double *sampling_time*) [virtual]

Function used to simulate the specified plant.

Parameters

double∗	state_vect State vector
double∗	input_vect Input vector
double	sampling_time Sampling time

Returns

double∗ New state vector

Implements mpc::model::Simulator.

Definition at line 17 of file tanks_system_simulator.cpp.

```
{
        ROS_ASSERT(sizeof(new_state_) == sizeof(current_state));

        // Solve the difference equations recursively
        double input = *current_input;
        Eigen::Map<Eigen::VectorXd> x_current(current_state, 2, 1);

        new_state_[0] = x_current(0) + sampling_time * (beta_ * input
       - cf_ * sqrt(2 * g_ * x_current(0))) / At_;
        new_state_[1] = x_current(1) + sampling_time * cf_ * (sqrt(2
      * g_ * x_current(0)) - sqrt(2 * g_ * x_current(1))) / At_;
//      new_state_[0] = 0.9992 * x_current(0) + 0.002551 * input;
//      new_state_[1] = -0.000803 * x_current(0) + 1.001 * x_current(1);


        return new_state_;
}
```

6.1.2.10 virtual void mpc::ModelPredictiveControl::updateMPC (double ∗ *x_measured,* double ∗ *x_reference*) [pure virtual]

Function to update the MPC algorithm for the next iteration. The parameters defined and calculated in mpc::ModelPredictiveControl::initMPC() are used together with the methods taken from the MPC class components (mpc::model::Model, mpc::optimizer::Optimizer and mpc::model::Simulator) to find a solution to the optimization problem. This is where the different variants of MPC algorithms can be implemented in a source file from a derived class.

Parameters

double∗	x_measured State vector
double∗	x_reference Reference vector

Implemented in mpc::STDMPC, and mpc::LBMPC.

6.1.2.11 void mpc::STDMPC::updateMPC (double ∗ *x_measured,* double ∗ *x_reference*) [virtual]

Function to solve the optimization problem formulated in the MPC.

Parameters

double∗	x_measured state vector
double∗	x_reference reference vector

Implements mpc::ModelPredictiveControl.

Definition at line 227 of file stdmpc.cpp.

```
{
        Eigen::Map<Eigen::VectorXd> x_measured_eigen(x_measured, states_
      , 1);
        Eigen::Map<Eigen::VectorXd> x_reference_eigen(x_reference, states_
      , 1);

        // Update of the model parameters
```

```
        if (model_->getModelType()) {
                model_->computeLinearSystem(A_, B_);
        }

        // Compute steady state control based on updated system matrices
        Eigen::JacobiSVD<Eigen::MatrixXd> SVD_B(B_, Eigen::ComputeThinU |
      Eigen::ComputeThinV);
        u_reference_ = SVD_B.solve(x_reference_eigen - A_ *
      x_reference_eigen);

        // Creation of the base vector
        A_pow_.push_back(Eigen::MatrixXd::Identity(states_, states_
      ));
        for (int i = 1; i < horizon_ + 1; i++) {
                Eigen::MatrixXd A_pow_i = A_pow_[i-1] * A_;
                A_pow_.push_back(A_pow_i);
        }

        // Compute the hessian matrix and gradient vector for the quadratic
       program
        A_bar_ = Eigen::MatrixXd::Zero((horizon_ + 1) * states_, states_
      );
        B_bar_ = Eigen::MatrixXd::Zero((horizon_ + 1) * states_,
      horizon_ * inputs_);
//      Eigen::MatrixXd H_x = Eigen::MatrixXd::Zero((horizon_ + 1) * states_,
       horizon_ * states_);
        Eigen::MatrixXd x_ref_bar = Eigen::MatrixXd::Zero((horizon_ + 1) *
      states_, 1);
        for (int i = 0; i < horizon_ + 1; i++) {
                for (int j = 0; j < horizon_; j++) {
                        if (i == horizon_) {
                                if (j == 0)
                                        A_bar_.block(i * states_, 0,
      states_, states_) = A_pow_[i];
                                if (j == 0)
                                        x_ref_bar.block(i * states_, 0, states_
      , 1) = x_reference_eigen;
                                if (i > j) {
                                        B_bar_.block(i * states_, j * inputs_
      , states_, inputs_) = A_pow_[i-j-1] * B_;
//                                      H_x.block(i * states_, j * states_,
       states_, states_) = A_p_BK_pow_[i-j-1];
                                }
                        }
                        else {
                                if (j == 0) {
                                        A_bar_.block(i * states_, 0, states_,
      states_) = A_pow_[i];
                                        x_ref_bar.block(i * states_, 0, states_
      , 1) = x_reference_eigen;
                                }
                                if (i > j) {
                                        B_bar_.block(i * states_, j * inputs_
      , states_, inputs_) = A_pow_[i-j-1] * B_;
//                                      H_x.block(i * states_, j * states_,
       states_, states_) = A_p_BK_pow_[i-j-1];
                                }
                        }
                }
        }
        double hessian_matrix[horizon_ * inputs_][horizon_ * inputs_
      ];
        double gradient_vector[horizon_ * inputs_];
        Eigen::Map<Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic,
       Eigen::RowMajor> > H(&hessian_matrix[0][0], horizon_ * inputs_, horizon_ *
      inputs_);
        Eigen::Map<Eigen::VectorXd> g(gradient_vector, horizon_ * inputs_
      , 1);

        // Computing the values of the Hessian matrix and Gradient vector
        H = B_bar_.transpose() * Q_bar_ * B_bar_ + R_bar_;
        g = B_bar_.transpose() * Q_bar_ * (A_bar_ * x_measured_eigen -
      x_ref_bar);

        // Transforming constraints and bounds to array
        double lbG_bar[constraints_ * horizon_];
        double ubG_bar[constraints_ * horizon_];
        double lb_bar[horizon_ * inputs_];
        double ub_bar[horizon_ * inputs_];
        Eigen::Map<Eigen::VectorXd> lbG_bar_eigen(lbG_bar, constraints_
       * horizon_, 1);
```

```
    Eigen::Map<Eigen::VectorXd> ubG_bar_eigen(ubG_bar, constraints_
   * horizon_, 1);
    Eigen::Map<Eigen::VectorXd> lb_bar_eigen(lb_bar, inputs_ *
  horizon_, 1);
    Eigen::Map<Eigen::VectorXd> ub_bar_eigen(ub_bar, inputs_ *
  horizon_, 1);
    lbG_bar_eigen = lbG_bar_ - M_bar_ * A_bar_ * x_measured_eigen;
    ubG_bar_eigen = ubG_bar_ - M_bar_ * A_bar_ * x_measured_eigen;
    lb_bar_eigen = lb_bar_;
    ub_bar_eigen = ub_bar_;


    // Mapping of the extended constraint matrix G_bar_
    double constraint_matrix[horizon_ * constraints_][horizon_
  * inputs_];
    Eigen::Map<Eigen::MatrixXd, Eigen::RowMajor> G_bar(&constraint_matrix[0
  ][0], constraints_ * horizon_, horizon_ * inputs_);
    G_bar = M_bar_ * B_bar_;


    double cputime = 0.008;//1.0;//NULL;
    bool success = false;
    success = optimizer_->computeOpt(&hessian_matrix[0]
  [0], gradient_vector, &constraint_matrix[0][0], lb_bar, ub_bar, lbG_bar, ubG_bar
  , cputime);
    if (success) {
            mpc_solution_ = optimizer_->
  getOptimalSolution();
            infeasibility_counter_ = 0;
    }
    else {
            infeasibility_counter_++;
            ROS_WARN("An optimal solution could not be obtained.");
    }

    // Save the data of the MPC
    for (int i = 0; i < states_; i++) {
            x_[i].push_back(x_measured[i]);
            xref_[i].push_back(x_reference[i]);
    }
    t_.push_back(time_index_);
    time_index_++;

    double *u = getControlSignal();
    for (int i = 0; i < inputs_; i++) {
            u_[i].push_back(u[i]);
    }
}
```

6.2 Example_models

Collaboration diagram for Example_models:

Classes

- class mpc::example_models::ArDrone

  Derived class from mpc::model::Model that represents the dynamics of Parrot's ARDrone1 quadrotor.
- class mpc::example_models::TanksSystem

  Class to define the process model of the tank system available at Simon Bolivar University's Automatic Control Lab.

Functions

- mpc::example_models::ArDrone::ArDrone ()

  Constructor function.
- mpc::example_models::ArDrone::∼ArDrone ()

  Destructor function.
- virtual void mpc::example_models::ArDrone::setLinearizationPoints (double ∗op_states)

  After the MPC completes an iteration, this function is used to set the new linearization points for a LTV model as global variables.
- virtual bool mpc::example_models::ArDrone::computeLinearSystem (Eigen::MatrixXd &A, Eigen::MatrixXd &B)

  Function to compute the dynamic model of the system.
- mpc::example_models::TanksSystem::TanksSystem ()

  Constructor function.
- mpc::example_models::TanksSystem::∼TanksSystem ()

  Destructor function.
- virtual void mpc::example_models::TanksSystem::setLinearizationPoints (double ∗op_states)

  After the MPC makes an iteration, this function is used to set the new linearization points for a LTV model into global variables.
- virtual bool mpc::example_models::TanksSystem::computeLinearSystem (Eigen::MatrixXd &A, Eigen::MatrixXd &B)

  Function to compute the dynamic model of the system.

6.2.1 Detailed Description

6.2.2 Function Documentation

6.2.2.1 mpc::example_models::ArDrone::ArDrone ( )

Constructor function.

All constants are defined in metric system units

Definition at line 8 of file ardrone.cpp.

```
{
        num_states_ = 12;
        num_inputs_ = 4;
        num_outputs_ = 12;
        op_point_states_ = new double[num_states_];
        op_point_input_ = new double[num_inputs_];

        A_ = Eigen::MatrixXd::Zero(num_states_, num_states_
      );
        B_ = Eigen::MatrixXd::Zero(num_states_, num_inputs_
      );

        time_variant_ = true;

        Ct_ = 8.17e-006;
        Cq_ = 2.17e-007;
        Ixx_ = 2.04e-003;
        Iyy_ = 1.57e-003;
        Izz_ = 3.52e-003;
        m_ = 0.4305;
        d_ = 0.35;
        ts_ = 0.0083;
        g_ = 9.81;
}
```

6.2.2.2 bool mpc::example_models::ArDrone::computeLinearSystem (Eigen::MatrixXd & *A*, Eigen::MatrixXd & *B*) [virtual]

Function to compute the dynamic model of the system.

Parameters

Eigen::MatrixXd&	A State matrix
Eigen::MatrixXd&	B Input matrix

Returns

bool Label that indicates if the computation of the matrices is successful

Implements mpc::model::Model.

Definition at line 62 of file ardrone.cpp.

```
{
        Eigen::Map<Eigen::VectorXd> x_bar(op_point_states_,
      num_states_);
        Eigen::Map<Eigen::VectorXd> u_bar(op_point_input_,
      num_inputs_);
        Eigen::MatrixXd U = Eigen::MatrixXd::Zero(num_inputs_,
      num_inputs_);

        double phi = x_bar(6);
        double theta = x_bar(7);
        double psi = x_bar(8);
        double p = x_bar(9);
        double q = x_bar(10);
        double r = x_bar(11);
        double U1 = Ct_ * (pow(u_bar(0),2) + pow(u_bar(1),2) + pow(u_bar(2),2)
      + pow(u_bar(3),2));

        A_(0,0) = 1.;
```

```
  A_(1,1) = 1.;
  A_(2,2) = 1.;
  A_(3,3) = 1.;
  A_(4,4) = 1.;
  A_(5,5) = 1.;
  A_(7,7) = 1.;
  A_(8,8) = 1.;
  A_(9,9) = 1.;
  A_(10,10) = 1.;
  A_(11,11) = 1.;
  A_(0,3) = ts_;
  A_(1,4) = ts_;
  A_(2,5) = ts_;
  A_(3,6) = ts_ * (sin(psi) * cos(phi) - cos(psi) * sin(theta) * sin(
phi)) * U1 / m_;
  A_(3,7) = ts_ * (cos(psi) * cos(theta) * cos(phi)) * U1 / m_;
  A_(3,8) = ts_ * (cos(psi) * sin(phi) - sin(psi) * sin(theta) * cos(
psi)) * U1 / m_;
  A_(4,6) = - ts_ * (sin(psi) * sin(theta) * sin(phi) + cos(psi) * cos(
phi)) * U1 / m_;
  A_(4,7) = ts_ * (sin(psi) * cos(theta) * cos(psi)) * U1 / m_;
  A_(4,8) = ts_ * (cos(psi) * sin(theta) * cos(phi) + sin(psi) * sin(
phi)) * U1 / m_;
  A_(5,6) = - ts_ * (cos(theta) * sin(phi)) * U1 / m_;
  A_(5,7) = - ts_ * (sin(theta) * cos(phi)) * U1 / m_;
  A_(6,6) = 1. + ts_ * (q * cos(phi) - r * sin(phi)) * tan(theta);
  A_(6,7) = ts_ * (q * sin(phi) + r * cos(phi)) / (cos(theta) * cos(
theta));
  A_(6,9) = ts_;
  A_(6,10) = ts_ * sin(phi) * tan(theta);
  A_(6,11) = ts_ * cos(phi) * tan(theta);
  A_(7,6) = - ts_ * (q * sin(phi) + r * cos(phi));
  A_(7,10) = ts_ * cos(phi);
  A_(7,11) = - ts_ * sin(phi);
  A_(8,6) = ts_ * (q * cos(phi) - r * sin(phi)) / cos(theta);
  A_(8,7) = ts_ * (q * sin(phi) + r * cos(phi)) * tan(theta) / cos(
theta);
  A_(8,10) = ts_ * sin(phi) / cos(theta);
  A_(8,11) = ts_ * cos(phi) / cos(theta);
  A_(9,10) = ts_ * r * (Iyy_ - Izz_) / Ixx_;
  A_(9,11) = ts_ * q * (Iyy_ - Izz_) / Ixx_;
  A_(10,9) = ts_ * r * (Izz_ - Ixx_) / Iyy_;
  A_(10,11) = ts_ * p * (Izz_ - Ixx_) / Iyy_;
  A_(11,9) = ts_ * q * (Ixx_ - Iyy_) / Izz_;
  A_(11,10) = ts_ * p * (Ixx_ - Iyy_) / Izz_;

  B_(3,0) = ts_ * (cos(psi) * sin(theta) * cos(phi) + sin(psi) * sin(
phi)) / m_;
  B_(4,0) = ts_ * (sin(psi) * sin(theta) * cos(phi) - cos(psi) * sin(
phi)) / m_;
  B_(5,0) = ts_ * cos(theta) * cos(phi) / m_;
  B_(9,1) = ts_ * d_ / Ixx_;
  B_(10,2) = ts_ * d_ / Iyy_;
  B_(11,3) = ts_ / Izz_;

  U(0,0) = 2 * Ct_ * u_bar(0);
  U(0,1) = 2 * Ct_ * u_bar(1);
  U(0,2) = 2 * Ct_ * u_bar(2);
  U(0,3) = 2 * Ct_ * u_bar(3);
  U(1,1) = - 2 * Ct_ * u_bar(1);
  U(1,3) = 2 * Ct_ * u_bar(3);
  U(2,0) = 2 * Ct_ * u_bar(0);
  U(2,2) = - 2 * Ct_ * u_bar(2);
  U(3,0) = - 2 * Cq_ * u_bar(0);
  U(3,1) = 2 * Cq_ * u_bar(1);
  U(3,2) = - 2 * Cq_ * u_bar(2);
  U(3,3) = 2 * Cq_ * u_bar(3);

  B_ = B_ * U;

  if (A.rows() != A_.rows()) {
          ROS_ERROR("The number of rows of the destination matrix
 variable and the model matrix A is different!");
          return false;
  }
  else if (A.cols() != A_.cols()) {
          ROS_ERROR("The number of columns of the destination matrix
 variable and the model matrix A is different!");
          return false;
  }
  else
```

```
            A = A_;


    if (B.rows() != B_.rows()) {
            ROS_ERROR("The number of rows of the destination matrix
    variable and the model matrix B is different!");
            return false;
    }
    else if (B.cols() != B_.cols()) {
            ROS_ERROR("The number of columns of the destination matrix
    variable and the model matrix B is different!");
            return false;
    }
    else
            B = B_;

    return true;
}
```

6.2.2.3 bool mpc::example_models::TanksSystem::computeLinearSystem (Eigen::MatrixXd & *A*, Eigen::MatrixXd & *B*) [virtual]

Function to compute the dynamic model of the system.

Parameters

Eigen::MatrixXd&	A State matrix
Eigen::MatrixXd&	B Input matrix

Returns

bool Label that indicates if the computation of the matrices is successful

Implements mpc::model::Model.

Definition at line 19 of file tanks_system.cpp.

```
{
        A_ = Eigen::MatrixXd::Zero(num_states_, num_states_
    );
        B_ = Eigen::MatrixXd::Zero(num_states_, num_inputs_
    );

        // A matrix
        A_(0,0) = 0.9992;
        A_(0,1) = 0.0000;
        A_(1,0) = -0.000803;
        A_(1,1) = 1.001;

        // B matrix
        B_(0,0) = 0.002551;
        B_(1,0) = 0.0000;

        if (A.rows() != A_.rows()) {
                ROS_ERROR("The number of rows of the destination matrix
    variable and the model matrix A is different!");
                return false;
        }
        else if (A.cols() != A_.cols()) {
                ROS_ERROR("The number of columns of the destination matrix
    variable and the model matrix A is different!");
                return false;
        }
        else
                A = A_;


        if (B.rows() != B_.rows()) {
                ROS_ERROR("The number of rows of the destination matrix
```

```
     variable and the model matrix B is different!");
              return false;
      }
      else if (B.cols() != B_.cols()) {
              ROS_ERROR("The number of columns of the destination matrix
     variable and the model matrix B is different!");
              return false;
      }
      else
              B = B_;

      return true;
}
```

6.2.2.4 void mpc::example_models::TanksSystem::setLinearizationPoints (double * *op_states*) [virtual]

After the MPC makes an iteration, this function is used to set the new linearization points for a LTV model into global variables.

Parameters

double∗	op_states new linearization point for the state vector

Implements mpc::model::Model.

Definition at line 17 of file tanks_system.cpp.

```
{ }
```

6.2.2.5 void mpc::example_models::ArDrone::setLinearizationPoints (double * *op_states*) [virtual]

After the MPC completes an iteration, this function is used to set the new linearization points for a LTV model as global variables.

Parameters

double∗	op_states new linearization point for the state vector

Implements mpc::model::Model.

Definition at line 34 of file ardrone.cpp.

```
{
        for (int i = 0; i < num_states_; i++) {
                op_point_states_[i] = op_states[i];
        }

        Eigen::MatrixXd M = Eigen::MatrixXd::Zero(num_inputs_,
      num_inputs_);
        M << 1., 1., 1., 1., 0., -1., 0., 1., 1., 0., -1., 0., -1., 1., -1., 1.
      ;
        Eigen::VectorXd f_bar = Eigen::MatrixXd::Zero(num_inputs_, 1
      );

        Eigen::Map<Eigen::VectorXd> u_bar(op_point_input_,
      num_inputs_);


        double phi = op_point_states_[6];
        double theta = op_point_states_[7];
        double p = op_point_states_[9];
        double q = op_point_states_[10];
        double r = op_point_states_[11];
```

```
        f_bar(0) = g_ * m_ / (Ct_ * cos(phi) * cos(theta));
        f_bar(1) = (Izz_ - Iyy_) * q * r / (Ct_ * d_);
        f_bar(2) = (Izz_ - Ixx_) * p * r / (Ct_ * d_);
        f_bar(3) = (Iyy_ - Ixx_) * p * q / Cq_;
        u_bar = M.inverse() * f_bar;
        u_bar = u_bar.cwiseSqrt();
}
```

6.3 Model

Namespaces

- namespace mpc::model

  Model interfaces and implementations.

Classes

- class mpc::model::Model

  This is the abstract class used to create and define different process models, in a state space representation. The models can be defined as Linear Time Invariant (LTI) such as

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) \end{aligned}$$

  or Linear Time Variant (LTV) such as

$$\begin{aligned} \dot{x}(t) &= A(t)x(t) + B(t)u(t) \\ y(t) &= C(t)x(t) \end{aligned}$$

  A boolean member variable defines the type of model used. The C matrix is not considered in the computation of the system matrices because this matrix is not used by the MPC algorithm, in an effort to reduce computation time.

Functions

- mpc::model::Model::Model ()

  Constructor function.
- mpc::model::Model::∼Model ()

  Destructor function.
- virtual void mpc::model::Model::setLinearizationPoints (double ∗op_states)=0

  After the MPC makes an iteration, this function is used to set the current state as the new linearization points for a LTV model into global variables.
- virtual bool mpc::model::Model::computeLinearSystem (Eigen::MatrixXd &A, Eigen::MatrixXd &B)=0

  Function to compute the matrices for a Linear model process. If the model is a LTI model, the function can be just defined to set the values of the model matrices.
- virtual int mpc::model::Model::getStatesNumber () const

  Get the states number of the dynamic model.
- virtual int mpc::model::Model::getInputsNumber () const

  Get the inputs number of the dynamic model.
- virtual int mpc::model::Model::getOutputsNumber () const

  Get the outputs number of the dynamic model.
- virtual bool **mpc::model::Model::setStates** (const double ∗states) const
- virtual bool **mpc::model::Model::setInputs** (const double ∗inputs) const
- virtual bool mpc::model::Model::getModelType () const

  Function to identify if the model is LTI or LTV.
- virtual double ∗ mpc::model::Model::getOperationPointsStates () const

  Function that returns the current value of the operation points for the states.
- virtual double ∗ mpc::model::Model::getOperationPointsInputs () const

  Function that returns the current value of the operation points for the inputs.

Variables

- Eigen::MatrixXd mpc::model::Model::A_

  State matrix of the dynamic model.
- Eigen::MatrixXd mpc::model::Model::B_

  Input matrix of the dynamic model.
- int mpc::model::Model::num_states_

  Number of states of the dynamic model.
- int mpc::model::Model::num_inputs_

  Number of inputs of the dynamic model.
- int mpc::model::Model::num_outputs_

  Number of outputs of the dynamic model.
- double * mpc::model::Model::op_point_states_
- double * mpc::model::Model::op_point_input_
- bool mpc::model::Model::time_variant_

  Boolean to check if the model is time variant or not (True = LTV)

6.3.1 Detailed Description

6.3.2 Function Documentation

6.3.2.1 virtual bool mpc::model::Model::computeLinearSystem (Eigen::MatrixXd & *A*, Eigen::MatrixXd & *B*) [pure virtual]

Function to compute the matrices for a Linear model process. If the model is a LTI model, the function can be just defined to set the values of the model matrices.

Parameters

Eigen::MatrixXd&	A State or System matrix
Eigen::MatrixXd&	B Input matrix

Returns

bool Label that indicates if the computation of the matrices is successful

Implemented in mpc::example_models::ArDrone, mpc::example_models::TanksSystem, and mpc::example_models::ArDroneHovering.

6.3.2.2 bool mpc::model::Model::getModelType ( ) const [inline], [virtual]

Function to identify if the model is LTI or LTV.

Returns

bool True if the model is LTV

Definition at line 146 of file model.h.

```
{
        return time_variant_;
}
```

6.3.2.3 virtual void mpc::model::Model::setLinearizationPoints (double * *op_states*) [pure virtual]

After the MPC makes an iteration, this function is used to set the current state as the new linearization points for a LTV model into global variables.

Parameters

*double**	op_states new linearization point for the state vector

Implemented in mpc::example_models::ArDrone, mpc::example_models::TanksSystem, and mpc::example_models::-ArDroneHovering.

6.3.3 Variable Documentation

6.3.3.1 double* mpc::model::Model::op_point_input_ [protected]

Pointer to the array of the input operation points

Definition at line 102 of file model.h.

6.3.3.2 double* mpc::model::Model::op_point_states_ [protected]

Pointer to the array of the states operation points

Definition at line 99 of file model.h.

6.4 Optimizer

Collaboration diagram for Optimizer:

Namespaces

- namespace mpc::optimizer

    Optimizer interfaces and implementations.

Classes

- class mpc::optimizer::Optimizer

    Abstract class to define the optimization algorithm for Model Predictive Control. This class acts as an interface to use a defined optimization solver software as a part of this library in order to provide different solver options for the end user to solve the basic optimization problem that rises in MPC. As more solvers are adapted to this library with this class, more options to try different optimization methods are available to select the most suitable one depending on each case. The basic

$$\begin{aligned} \textit{Minimize } F(x) & & \\ \textit{subject to } G(x) &= 0 \\ H(x) &\geq 0 \end{aligned}$$

    As more solvers are adapted to this library with this class, more options to try different optimization methods are available to select the most suitable one depending on each case.

Functions

- mpc::optimizer::Optimizer::Optimizer ()

    Constructor function.
- mpc::optimizer::Optimizer::∼Optimizer ()

    Destructor function.
- virtual bool mpc::optimizer::Optimizer::init ()=0

    Function to perform the initialization of optimizer, if this applies.
- virtual bool mpc::optimizer::Optimizer::computeOpt (double ∗H, double ∗g, double ∗G, double ∗lb, double ∗ub, double ∗lbG, double ∗ubG, double cputime)=0

    Function to compute the optimization algorithm associated to the MPC problem.
- virtual double ∗ mpc::optimizer::Optimizer::getOptimalSolution ()=0

    Get the vector of optimal or sub-optimal solutions calculated by the mpc::optimizer::Optimizer::computeOpt() function (optimality of the function is defined by the solver that is adapted).
- virtual int mpc::optimizer::Optimizer::getConstraintNumber () const

Get the number of constraints.

- virtual int mpc::optimizer::Optimizer::getVariableNumber () const

  Get the number of variables, i.e inputs ∗ horizon.
- virtual void mpc::optimizer::Optimizer::setHorizon (int horizon)

  Set the horizon of the MPC.
- virtual void mpc::optimizer::Optimizer::setVariableNumber (int variables)

  Set the number of variables, i.e inputs ∗ horizon.

Variables

- int mpc::optimizer::Optimizer::variables_

  Number of variables, i.e inputs ∗ horizon.
- int mpc::optimizer::Optimizer::constraints_

  Number of constraints.
- int mpc::optimizer::Optimizer::horizon_

  Horizon of MPC.

6.4.1 Detailed Description

6.4.2 Function Documentation

6.4.2.1 virtual bool mpc::optimizer::Optimizer::computeOpt (double ∗ *H,* double ∗ *g,* double ∗ *G,* double ∗ *lb,* double ∗ *ub,* double ∗ *lbG,* double ∗ *ubG,* double *cputime*) [pure virtual]

Function to compute the optimization algorithm associated to the MPC problem.

Parameters

double∗	H Hessian matrix
double∗	g Gradient vector
double∗	G Constraint matrix
double∗	lb Low bound vector
double∗	ub Upper bound vector
double∗	lbG Low constraint vector
double∗	lbG Upper constraint vector
double	cputime CPU-time for computing the optimization. If NULL, it provides on output the actual calculation time of the optimization problem.

Returns

bool Label that indicates if the computation of the optimization is successful

Implemented in mpc::optimizer::qpOASES.

6.4.2.2 int mpc::optimizer::Optimizer::getConstraintNumber ( ) const [inline], [virtual]

Get the number of constraints.

Returns

int Number of constraints

Definition at line 109 of file optimizer.h.

```
{
        return constraints_;
}
```

6.4.2.3 virtual double∗ mpc::optimizer::Optimizer::getOptimalSolution ( ) [pure virtual]

Get the vector of optimal or sub-optimal solutions calculated by the mpc::optimizer::Optimizer::computeOpt() function (optimality of the function is defined by the solver that is adapted).

Returns

double∗ Optimal solution

Implemented in mpc::optimizer::qpOASES.

6.4.2.4 int mpc::optimizer::Optimizer::getVariableNumber ( ) const [inline], [virtual]

Get the number of variables, i.e inputs ∗ horizon.

Returns

int Number of variables

Definition at line 114 of file optimizer.h.

```
{
        return variables_;
}
```

6.4.2.5 virtual bool mpc::optimizer::Optimizer::init ( ) [pure virtual]

Function to perform the initialization of optimizer, if this applies.

Returns

Label that indicates if the initialization of the optimizer is successful

Implemented in mpc::optimizer::qpOASES.

Chapter 7

Namespace Documentation

7.1 mpc Namespace Reference

Model Predictives Control interfaces and implementations.

Namespaces

- namespace model

  Model interfaces and implementations.
- namespace optimizer

  Optimizer interfaces and implementations.

Classes

- class LBMPC

  Class for solving the learning-based model predictive control problem
  The aim of this class is to solve the learning-based model predictive control of the following form:

$$
\begin{aligned}
\min_{\mathbf{u}_{k_0},\cdots,\mathbf{u}_{k_0+N-1}} J_N(x,u) &= \frac{1}{2}(\tilde{\mathbf{x}}_{k_0+N}-\bar{\mathbf{x}}_{ref})^T\mathbf{P}(\tilde{\mathbf{x}}_{k_0+N}-\bar{\mathbf{x}}_{ref})+\frac{1}{2}\sum_{k=k_0}^{k_0+N-1}(\tilde{\mathbf{x}}_k-\bar{\mathbf{x}}_{ref})^T\mathbf{Q}(\tilde{\mathbf{x}}_k-\bar{\mathbf{x}}_{ref})+(\tilde{\mathbf{u}}_k-\bar{\mathbf{u}}_{ref})^T\mathbf{R}(\tilde{\mathbf{u}}_k-\bar{\mathbf{u}}_{ref})\\
\tilde{\mathbf{x}}_{k_0} &= \bar{\mathbf{x}}_{k_0}=\hat{\mathbf{x}}_{k_0}\\
\tilde{\mathbf{x}}_{k+1} &= (\mathbf{A}+\mathbf{F})\tilde{\mathbf{x}}_k+(\mathbf{B}+\mathbf{H})\check{\mathbf{u}}_k+\mathbf{k}+\mathbf{z} \qquad \forall k\in[k_0,N]\\
\bar{\mathbf{x}}_{k+1} &= \mathbf{A}\bar{\mathbf{x}}_k+\mathbf{B}\check{\mathbf{u}}_k+\mathbf{k} \qquad \forall k\in[k_0,N]\\
\check{\mathbf{u}}_k &= \mathbf{K}\bar{\mathbf{x}}_j+\mathbf{c}_k \qquad \forall k\in[k_0,N]\\
\bar{\mathbf{x}}_{\mathbf{k}} &\in \mathbf{X} \qquad \forall k\in[k_0,N]\\
\check{\mathbf{u}}_{\mathbf{k}} &\in \mathbf{U} \qquad \forall k\in[k_0,N]\\
\bar{\mathbf{x}}_{\mathbf{k}} &\in \mathbf{X}\ominus\mathbf{D} \qquad \forall k\in[k_0,N]\\
(\bar{\mathbf{x}}_{\mathbf{k}},\xi) &\in \omega \qquad \forall k\in[k_0,N]
\end{aligned}
$$

  To solve each of these optimal control problems the function mpc::LBMPC::initMPC initialized the control problem. The resulting optimization problem is then solved by a (predefined) minimization routine.
  Then the first value of the computed control is implemented and the optimization horizon is shifted forward in time. This allows the procedure to be applied iteratively and computes a (suboptimal) infinite horizon control.
  Note that the function mpc::LBMPC::updatedMPC can be used to computer a control signal for the next time-step.
  .

- class ModelPredictiveControl

    This class serves as a base class in order to expand the functionality of the library and implement different sorts of MPC algorithms. The methods defined here are conceived in the simplest way possible to allow different implementations in the derived classes.

    .

- class STDMPC

    Class for solving the explicit model predictive control problem
    The aim of this class is to solve the explicit model predictive control of the following form:

$$
\begin{aligned}
\min_{\mathbf{u}_{k_0},\cdots,\mathbf{u}_{k_0+N-1}} J_N(x,u) &= \frac{1}{2}(\mathbf{x}_{k_0+N}-\mathbf{x}_{ref})^T\mathbf{P}(\mathbf{x}_{k_0+N}-\mathbf{x}_{ref})+\frac{1}{2}\sum_{k=k_0}^{k_0+N-1}(\mathbf{x}_k-\mathbf{x}_{ref})^T\mathbf{Q}(\mathbf{x}_k-\mathbf{x}_{ref})+(\mathbf{u}_k-\mathbf{u}_{ref})^T\mathbf{R}(\mathbf{u}_k-\mathbf{u}_{ref}) \\
\mathbf{x}_{k_0} &= \omega_0(k_0) \\
\mathbf{x}_{k+1} &= \mathbf{A}\mathbf{x}_k+\mathbf{B}\mathbf{u}_k \qquad \forall k\in[k_0,N] \\
\bar{x} &\leq \mathbf{M}\mathbf{x}_k \qquad \forall k\in[k_0,N] \\
\bar{u} &\leq \mathbf{N}\mathbf{u}_k \qquad \forall k\in[k_0,N]
\end{aligned}
$$

    To solve each of these optimal control problems the function mpc::STDMPC::initMPC initialized the control problem. The resulting optimization problem is then solved by a (predefined) minimization routine.
    Then the first value of the computed control is implemented and the optimization horizon is shifted forward in time. This allows the procedure to be applied iteratively and computes a (suboptimal) infinite horizon control.
    Note that the function mpc::STDMPC::updateMPC() can be used to compute a control signal for the next time-step.

    .

7.1.1 Detailed Description

Model Predictives Control interfaces and implementations.

7.2 mpc::model Namespace Reference

Model interfaces and implementations.

Classes

- class Model

    This is the abstract class used to create and define different process models, in a state space representation. The models can be defined as Linear Time Invariant (LTI) such as

$$
\begin{aligned}
\dot{x}(t) &= Ax(t)+Bu(t) \\
y(t) &= Cx(t)
\end{aligned}
$$

    or Linear Time Variant (LTV) such as

$$
\begin{aligned}
\dot{x}(t) &= A(t)x(t)+B(t)u(t) \\
y(t) &= C(t)x(t)
\end{aligned}
$$

    A boolean member variable defines the type of model used. The C matrix is not considered in the computation of the system matrices because this matrix is not used by the MPC algorithm, in an effort to reduce computation time.

- class Simulator

This class provides methods to simulate a given model of a process defined by a class mpc::model::Model object This class provides methods to simulate a given model

$$\dot{x}(t) = Ax(t) + Bu(t)$$

$$y(t) = Cx(t)$$

on a fixed prediction horizon interval $[t_0, t_N]$ with initial value $x(t_0, x_0) = x_0$ and given control $u(\cdot, x_0)$. That is, for a given class mpc::model::Model object and a given control u the simulator can solve the differential or difference equation forward in time.

7.2.1 Detailed Description

Model interfaces and implementations.

7.3 mpc.msg._MPCState Namespace Reference

Classes

- class MPCState

Variables

- int **python3** = 0x03000000
- **_struct_I** = genpy.struct_I
- tuple **_struct_3I** = struct.Struct("<3I")

7.3.1 Detailed Description

```
autogenerated by genpy from mpc/MPCState.msg. Do not edit.
```

7.4 mpc::optimizer Namespace Reference

Optimizer interfaces and implementations.

Classes

- class Optimizer

  Abstract class to define the optimization algorithm for Model Predictive Control. This class acts as an interface to use a defined optimization solver software as a part of this library in order to provide different solver options for the end user to solve the basic optimization problem that rises in MPC. As more solvers are adapted to this library with this class, more options to try different optimization methods are available to select the most suitable one depending on each case. The basic

$$\begin{aligned} \textit{Minimize } F(x) & \\ \textit{subject to } G(x) &= 0 \\ H(x) &\geq 0 \end{aligned}$$

  As more solvers are adapted to this library with this class, more options to try different optimization methods are available to select the most suitable one depending on each case.

- class qpOASES

  Class to interface the qpOASES library This class gives an interface with qpOASES library in order to implement a quadratic program using online active set strategy for MPC controller. qpOASES solve a convex optimization class of the following form

$$\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T \mathbf{H}\mathbf{x} + \mathbf{x}^T \mathbf{g}(\mathbf{x_0})$$

  suject to

$$\begin{aligned} lbG(\mathbf{x_0}) \leq \quad & \mathbf{Gx} & \leq ubG(\mathbf{x_0}) \\ lb(\mathbf{x_0}) \leq \quad & \mathbf{x} & \leq ub(\mathbf{x_0}) \end{aligned}$$

  .

7.4.1 Detailed Description

Optimizer interfaces and implementations.

Chapter 8

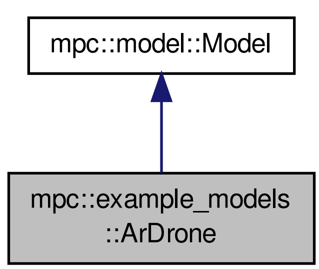
Class Documentation

8.1 mpc::example_models::ArDrone Class Reference

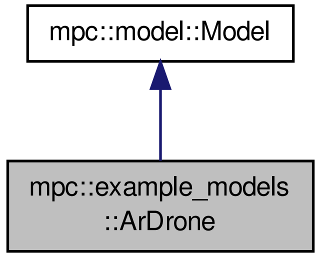
Derived class from mpc::model::Model that represents the dynamics of Parrot's ARDrone1 quadrotor.

`#include <ardrone.h>`

Inheritance diagram for mpc::example_models::ArDrone:



Collaboration diagram for mpc::example_models::ArDrone:



Public Member Functions

- ArDrone ()
  Constructor function.
- ∼ArDrone ()
  Destructor function.
- virtual void setLinearizationPoints (double ∗op_states)
  After the MPC completes an iteration, this function is used to set the new linearization points for a LTV model as global variables.
- virtual bool computeLinearSystem (Eigen::MatrixXd &A, Eigen::MatrixXd &B)
  Function to compute the dynamic model of the system.

Additional Inherited Members

8.1.1 Detailed Description

Derived class from mpc::model::Model that represents the dynamics of Parrot's ARDrone1 quadrotor.

Definition at line 25 of file ardrone.h.

The documentation for this class was generated from the following files:

- /home/rene/ros_workspace/model-predictive-control/mpc/include/mpc/example_models/ardrone.h
- /home/rene/ros_workspace/model-predictive-control/mpc/src/example_models/ardrone.cpp

8.2 mpc::example_models::ArDroneHovering Class Reference

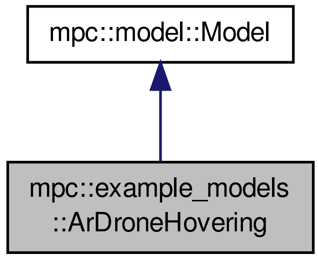
Class to define the example model, tanks system, of the process and the optimal control problem to be solved This class gives an definition of an example model, tanks systems, of process model and the optimal control problem which shall be considered. The model itself is defined via its dynamic

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) \end{aligned}$$

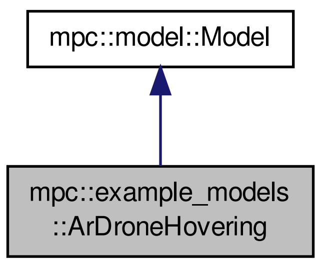
on the optimization horizon $[t_0, N]$ with initial value $x(t_0, x_0) = x_0$ over an optimization criterion.

`#include <ardrone_hovering.h>`

Inheritance diagram for mpc::example_models::ArDroneHovering:



Collaboration diagram for mpc::example_models::ArDroneHovering:



Public Member Functions

- ArDroneHovering ()

  Constructor function.
- ∼ArDroneHovering ()

  Destructor function.
- virtual void setLinearizationPoints (double ∗op_states)

  After the MPC makes an iteration, this function is used to set the new linearization points for a LTV model into global variables for the STDMPC class.
- virtual bool computeLinearSystem (Eigen::MatrixXd &A, Eigen::MatrixXd &B)

  Function to compute the dynamic model of the system.

Additional Inherited Members

8.2.1 Detailed Description

Class to define the example model, tanks system, of the process and the optimal control problem to be solved This class gives an definition of an example model, tanks systems, of process model and the optimal control problem which shall be considered. The model itself is defined via its dynamic

$$\dot{x}(t) = Ax(t) + Bu(t)$$
$$y(t) = Cx(t)$$

on the optimization horizon $[t_0, N]$ with initial value $x(t_0, x_0) = x_0$ over an optimization criterion.

Definition at line 22 of file ardrone_hovering.h.

8.2.2 Constructor & Destructor Documentation

8.2.2.1 mpc::example_models::ArDroneHovering::ArDroneHovering ( )

Constructor function.

All constants are defined in metric system units

Definition at line 8 of file ardrone_hovering.cpp.

```
{
        num_states_ = 10;
        num_inputs_ = 4;
        num_outputs_ = 10;
        op_point_states_ = new double[num_states_];
        op_point_input_ = new double[num_inputs_];

        A_ = Eigen::MatrixXd::Zero(num_states_, num_states_
      );
        B_ = Eigen::MatrixXd::Zero(num_states_, num_inputs_
      );

        time_variant_ = false;

        ts_ = 0.0083;
        c1_ = 10.324;//0.58;
        c2_ = 0.58;//17.8;
        c3_ = 350.;//10.;
        c4_ = 10.;//35.;
        c5_ = 250.;//10.;
        c6_ = 10.;//25.;
        c7_ = 1.4;//1.4;
        c8_ = 1.4;//1.0;
}
```

8.2.3 Member Function Documentation

8.2.3.1 bool mpc::example_models::ArDroneHovering::computeLinearSystem (Eigen::MatrixXd & *A*, Eigen::MatrixXd & *B*) [virtual]

Function to compute the dynamic model of the system.

Parameters

Eigen::MatrixXd&	A State matrix
Eigen::MatrixXd&	B Input matrix

Returns

bool Label that indicates if the computation of the matrices is successful

Implements mpc::model::Model.

Definition at line 62 of file ardrone_hovering.cpp.

```
{
        Eigen::Map<Eigen::VectorXd> x_bar(op_point_states_,
      num_states_);
        Eigen::Map<Eigen::VectorXd> u_bar(op_point_input_,
      num_inputs_);
        Eigen::MatrixXd U = Eigen::MatrixXd::Zero(num_inputs_,
      num_inputs_);

        double phi = x_bar(6);
        double theta = x_bar(7);
        double psi = x_bar(8);

        A_(0,0) = 1.;
        A_(0,3) = ts_;
        A_(1,1) = 1.;
        A_(1,4) = ts_;
        A_(2,2) = 1.;
        A_(2,5) = ts_;
        A_(3,3) = 1. - ts_ * c2_;
        A_(3,6) = ts_ * c1_ * cos(psi) * cos(phi) * cos(theta);
        A_(3,7) = - ts_ * c1_ * (cos(psi) * sin(phi) * sin(theta) + sin(psi)
      * cos(theta));
        A_(3,8) = - ts_ * c1_ * (sin(psi) * sin(phi) * cos(theta) + cos(psi)
      * sin(theta));
        A_(4,4) = 1. - ts_ * c2_;
        A_(4,6) = - ts_ * c1_ * sin(psi) * cos(phi) * cos(theta);
        A_(4,7) = ts_ * c1_ * (sin(psi) * sin(phi) * sin(theta) - cos(psi) *
      cos(theta));
        A_(4,8) = ts_ * c1_ * (-cos(psi) * sin(phi) * cos(theta) + sin(psi) *
       sin(theta));
        A_(5,5) = 1. - ts_ * c8_;
        A_(6,6) = 1. - ts_ * c4_;
        A_(7,7) = 1. - ts_ * c4_;
        A_(8,8) = 1.;
        A_(8,9) = - ts_ * c6_;
        A_(9,9) = 1.;

        B_(5,2) = ts_ * c7_;
        B_(6,0) = ts_ * c3_;
        B_(7,1) = ts_ * c3_;
        B_(9,3) = ts_ * c5_;

        if (A.rows() != A_.rows()) {
                ROS_ERROR("The number of rows of the destination matrix
       variable and the model matrix A is different!");
                return false;
        }
        else if (A.cols() != A_.cols()) {
                ROS_ERROR("The number of columns of the destination matrix
       variable and the model matrix A is different!");
                return false;
        }
        else
                A = A_;


        if (B.rows() != B_.rows()) {
                ROS_ERROR("The number of rows of the destination matrix
       variable and the model matrix B is different!");
                return false;
        }
        else if (B.cols() != B_.cols()) {
                ROS_ERROR("The number of columns of the destination matrix
       variable and the model matrix B is different!");
                return false;
        }
        else
                B = B_;

        return true;
```

```
}
```

8.2.3.2 void mpc::example_models::ArDroneHovering::setLinearizationPoints (double * *op_states*) [virtual]

After the MPC makes an iteration, this function is used to set the new linearization points for a LTV model into global variables for the STDMPC class.

Parameters

*double**	op_states new linearization point for the state vector

Implements mpc::model::Model.

Definition at line 34 of file ardrone_hovering.cpp.

```
{
        for (int i = 0; i < num_states_; i++) {
                op_point_states_[i] = op_states[i];
        }

/*      Eigen::MatrixXd M = Eigen::MatrixXd::Zero(num_inputs_, num_inputs_);
        M << 1., 1., 1., 1., 0., -1., 0., 1., 1., 0., -1., 0., -1., 1., -1.,
       1.;
        Eigen::VectorXd f_bar = Eigen::MatrixXd::Zero(num_inputs_, 1);

        Eigen::Map<Eigen::VectorXd> u_bar(op_point_input_, num_inputs_);


        double phi = op_point_states_[6];
        double theta = op_point_states_[7];
        double p = op_point_states_[9];
        double q = op_point_states_[10];
        double r = op_point_states_[11];

        f_bar(0) = g_ * m_ / (Ct_ * cos(phi) * cos(theta));
        f_bar(1) = (Izz_ - Iyy_) * q * r / (Ct_ * d_);
        f_bar(2) = (Izz_ - Ixx_) * p * r / (Ct_ * d_);
        f_bar(3) = (Iyy_ - Ixx_) * p * q / Cq_;
        u_bar = M.inverse() * f_bar;
        u_bar = u_bar.cwiseSqrt();*/
}
```

The documentation for this class was generated from the following files:

- /home/rene/ros_workspace/model-predictive-control/mpc/include/mpc/example_models/ardrone_hovering.h
- /home/rene/ros_workspace/model-predictive-control/mpc/src/example_models/ardrone_hovering.cpp

8.3 mpc::example_models::ArDroneSimulator Class Reference

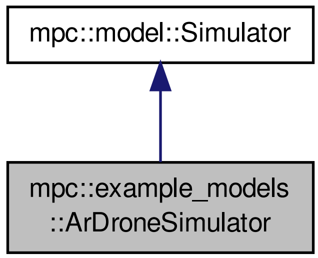
This class provides methods to simulate the Parrot ARDrone1 quadrotor defined as the following non-linear system

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

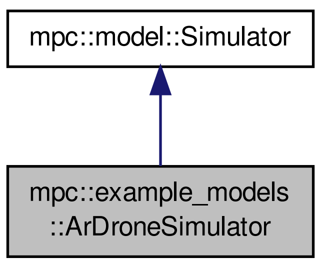
with initial value $x(t=0) = x_0$ and given control input for the given sample $u(\cdot, x_0)$ using a Euler backward integration method.

`#include <ardrone_simulator.h>`

Inheritance diagram for mpc::example_models::ArDroneSimulator:



Collaboration diagram for mpc::example_models::ArDroneSimulator:



Public Member Functions

- ArDroneSimulator ()

  Constructor function.
- ∼ArDroneSimulator ()

  Destructor function.
- double ∗ simulatePlant (double ∗current_state, double ∗current_input, double sampling_time)

  Function used to calculate the simulated output of the quadrotor for each time sample. In this simulator, the non linear model of the quadrotor system is implemented in this function.

Additional Inherited Members

8.3.1 Detailed Description

This class provides methods to simulate the Parrot ARDrone1 quadrotor defined as the following non-linear system

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

with initial value $x(t = 0) = x_0$ and given control input for the given sample $u(\cdot, x_0)$ using a Euler backward integration method.

This class provides methods to simulate the Parrot ARDrone1 quadrotor.

Definition at line 18 of file ardrone_simulator.h.

8.3.2 Member Function Documentation

8.3.2.1 double ∗ mpc::example_models::ArDroneSimulator::simulatePlant (double ∗ *current_state*, double ∗ *current_input*, double *sampling_time*) [virtual]

Function used to calculate the simulated output of the quadrotor for each time sample. In this simulator, the non linear model of the quadrotor system is implemented in this function.

Parameters

double∗	current_state State vector for the system in time k.
double∗	input_vect Input vector for the system in time k.
double	sampling_time Sampling time chosen for the simulation.

Returns

double∗ Array containing the state vector for the system in time k.

Implements mpc::model::Simulator.

Definition at line 27 of file ardrone_simulator.cpp.

```
{
        double ts;
        ts = sampling_time;

        /* Creating random noise to the simulator
           Noise will be added to the position and orientation states and from
       there, propagated to the others. */

        // Seeding the random number generator
        srand((unsigned)time(NULL));

        double noiseX = ((double)rand()/(double)RAND_MAX)*0.001;
        double noiseY = ((double)rand()/(double)RAND_MAX)*0.001;
        double noiseZ = ((double)rand()/(double)RAND_MAX)*0.001;


        double noiseRoll = ((double)rand()/(double)RAND_MAX)*0.001;
        double noisePitch = ((double)rand()/(double)RAND_MAX)*0.001;
        double noiseYaw = ((double)rand()/(double)RAND_MAX)*0.001;

        double noiseU = ((double)rand()/(double)RAND_MAX)*0.001;
        double noiseV = ((double)rand()/(double)RAND_MAX)*0.001;
        double noiseW = ((double)rand()/(double)RAND_MAX)*0.001;

        double noiseP = ((double)rand()/(double)RAND_MAX)*0.001;
        double noiseQ = ((double)rand()/(double)RAND_MAX)*0.001;
        double noiseR = ((double)rand()/(double)RAND_MAX)*0.001;

        // Map into Eigen objects for easier manipulation
        Eigen::Map<Eigen::VectorXd> x_current(current_state, 12, 1);
```

```
    Eigen::Map<Eigen::VectorXd> u_current(current_input, 4, 1);
    Eigen::VectorXd x_new = Eigen::MatrixXd::Zero(number_of_states_, 1);

    double phi = x_current(6);
    double theta = x_current(7);
    double psi = x_current(8);
    double p = x_current(9);
    double q = x_current(10);
    double r = x_current(11);
    double U1 = Ct_ * (pow(u_current(0),2) + pow(u_current(1),2) + pow(
  u_current(2),2) + pow(u_current(3),2));
    double U2 = Ct_ * (- pow(u_current(1),2) + pow(u_current(3),2));
    double U3 = Ct_ * (pow(u_current(0),2) - pow(u_current(2),2));
    double U4 = Cq_ * (-pow(u_current(0),2) + pow(u_current(1),2) - pow(
  u_current(2),2) + pow(u_current(3),2));


    // Solve the difference equations recursively
    x_new(0) = x_current(0) + ts * (x_current(3)) + noiseX;
    x_new(1) = x_current(1) + ts * (x_current(4)) + noiseY;
    x_new(2) = x_current(2) + ts * (x_current(5)) + noiseZ;
    x_new(3) = x_current(3) + (ts / m_) * (cos(psi) * sin(theta) * cos(phi)
   + sin(psi) * sin(phi)) * U1 + noiseRoll;
    x_new(4) = x_current(4) + (ts / m_) * (sin(psi) * sin(theta) * cos(phi)
   - cos(psi) * sin(phi)) * U1 + noisePitch;
    x_new(5) = x_current(5) + (ts / m_) * (-m_ * g_ + cos(theta) * cos(phi)
   * U1) + noiseYaw;
    x_new(6) = x_current(6) + ts * (p + q * sin(phi) * tan(theta) + r * cos
  (phi) * tan(theta)) + noiseU;
    x_new(7) = x_current(7) + ts * (q * cos(phi) - r * sin(phi)) + noiseV;
    x_new(8) = x_current(8) + ts * (q * sin(phi) + r * cos(phi)) / cos(
  theta) + noiseW;
    x_new(9) = x_current(9) + ts * ((Iyy_ - Izz_) * q * r / Ixx_ + (d_ /
  Ixx_) * U2) + noiseP;
    x_new(10) = x_current(10) + ts * ((Izz_ - Ixx_) * p * r / Iyy_ + (d_ /
  Iyy_) * U3) + noiseQ;
    x_new(11) = x_current(11) + ts * ((Ixx_ - Iyy_) * p * q / Izz_ + (1 /
  Izz_) * U4) + noiseR;


    x_current = x_new;
    return current_state;
}
```

The documentation for this class was generated from the following files:

- /home/rene/ros_workspace/model-predictive-control/mpc/include/mpc/example_models/ardrone_simulator.h
- /home/rene/ros_workspace/model-predictive-control/mpc/src/example_models/ardrone_simulator.cpp

8.4 mpc::LBMPC Class Reference

Class for solving the learning-based model predictive control problem

The aim of this class is to solve the learning-based model predictive control of the following form:

$$
\begin{aligned}
\min_{\mathbf{u}_{k_0},\cdots,\mathbf{u}_{k_0+N-1}} J_N(x,u) &= \frac{1}{2}(\tilde{\mathbf{x}}_{k_0+N}-\bar{\mathbf{x}}_{ref})^T\mathbf{P}(\tilde{\mathbf{x}}_{k_0+N}-\bar{\mathbf{x}}_{ref})+\frac{1}{2}\sum_{k=k_0}^{k_0+N-1}(\tilde{\mathbf{x}}_k-\bar{\mathbf{x}}_{ref})^T\mathbf{Q}(\tilde{\mathbf{x}}_k-\bar{\mathbf{x}}_{ref})+(\tilde{\mathbf{u}}_k-\bar{\mathbf{u}}_{ref})^T\mathbf{R}(\tilde{\mathbf{u}}_k-\bar{\mathbf{u}}_{ref}) \\
\tilde{\mathbf{x}}_{k_0} &= \bar{\mathbf{x}}_{k_0}=\hat{\mathbf{x}}_{k_0} \\
\tilde{\mathbf{x}}_{k+1} &= (\mathbf{A}+\mathbf{F})\tilde{\mathbf{x}}_k+(\mathbf{B}+\mathbf{H})\check{\mathbf{u}}_k+\mathbf{k}+\mathbf{z} \qquad \forall k\in[k_0,N] \\
\bar{\mathbf{x}}_{k+1} &= \mathbf{A}\bar{\mathbf{x}}_k+\mathbf{B}\check{\mathbf{u}}_k+\mathbf{k} \qquad \forall k\in[k_0,N] \\
\check{\mathbf{u}}_k &= \mathbf{K}\bar{\mathbf{x}}_j+\mathbf{c}_k \qquad \forall k\in[k_0,N] \\
\bar{\mathbf{x}}_{\mathbf{k}} &\in \mathbf{X} \qquad \forall k\in[k_0,N] \\
\check{\mathbf{u}}_{\mathbf{k}} &\in \mathbf{U} \qquad \forall k\in[k_0,N] \\
\bar{\mathbf{x}}_{\mathbf{k}} &\in \mathbf{X}\ominus\mathbf{D} \qquad \forall k\in[k_0,N] \\
(\bar{\mathbf{x}}_{\mathbf{k}},\xi) &\in \omega \qquad \forall k\in[k_0,N]
\end{aligned}
$$

To solve each of these optimal control problems the function mpc::LBMPC::initMPC initialized the control problem. The resulting optimization problem is then solved by a (predefined) minimization routine.

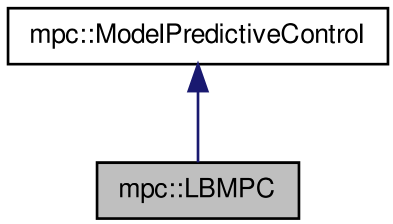
Then the first value of the computed control is implemented and the optimization horizon is shifted forward in time. This allows the procedure to be applied iteratively and computes a (suboptimal) infinite horizon control.

Note that the function mpc::LBMPC::updatedMPC can be used to computer a control signal for the next time-step.

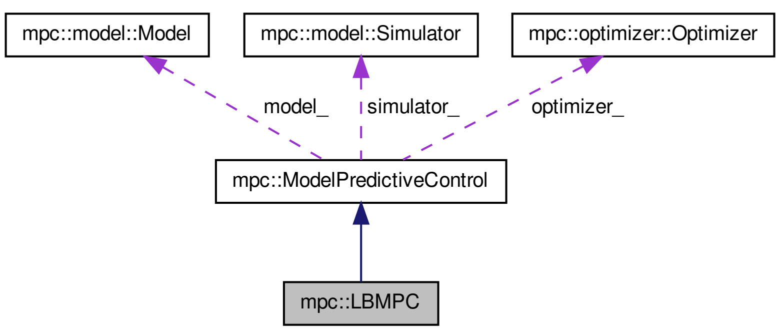
.

```
#include <lbmpc.h>
```

Inheritance diagram for mpc::LBMPC:



Collaboration diagram for mpc::LBMPC:



Public Member Functions

- LBMPC (ros::NodeHandle node)

  Constructorb function.
- ∼LBMPC ()

Destructor function.

- virtual bool resetMPC (mpc::model::Model ∗model, mpc::optimizer::Optimizer ∗optimizer, mpc::model::Simulator ∗simulator)

  Function to specify and set the settings of all the components within the MPC problem. The mpc::ModelPredictiveControl class can change individual parts of the MPC problem; such as the model (mpc::model::Model and derived classes), the optimizer (mpc::optimizer::Optimizer and derived classes) and, if used, the plant simulator (mpc::model::Simulator and derived classes) in order to allow different combinations of these parts when solving.

- virtual bool initMPC ()

  Function to initialize the calculation of the MPC algorithm. The function reads all required parameters from ROS' parameter server that has been previously loaded from a configuration YAML file, and performs all the initial calculations of variables to be used in the optimization problem.

- virtual void updateMPC (double ∗x_measured, double ∗x_reference)

  Function to solve the optimization problem formulated in the MPC.

Additional Inherited Members

8.4.1 Detailed Description

Class for solving the learning-based model predictive control problem

The aim of this class is to solve the learning-based model predictive control of the following form:

$$\begin{aligned}
\min_{\mathbf{u}_{k_0},\cdots,\mathbf{u}_{k_0+N-1}} J_N(x,u) &= \frac{1}{2}(\tilde{\mathbf{x}}_{k_0+N}-\bar{\mathbf{x}}_{ref})^T\mathbf{P}(\tilde{\mathbf{x}}_{k_0+N}-\bar{\mathbf{x}}_{ref})+\frac{1}{2}\sum_{k=k_0}^{k_0+N-1}(\tilde{\mathbf{x}}_k-\bar{\mathbf{x}}_{ref})^T\mathbf{Q}(\tilde{\mathbf{x}}_k-\bar{\mathbf{x}}_{ref})+(\tilde{\mathbf{u}}_k-\bar{\mathbf{u}}_{ref})^T\mathbf{R}(\tilde{\mathbf{u}}_k-\bar{\mathbf{u}}_{ref}) \\
\tilde{\mathbf{x}}_{k_0} &= \bar{\mathbf{x}}_{k_0}=\hat{\mathbf{x}}_{k_0} \\
\tilde{\mathbf{x}}_{k+1} &= (\mathbf{A}+\mathbf{F})\tilde{\mathbf{x}}_k+(\mathbf{B}+\mathbf{H})\check{\mathbf{u}}_k+\mathbf{k}+\mathbf{z} \qquad \forall k\in[k_0,N] \\
\bar{\mathbf{x}}_{k+1} &= \mathbf{A}\bar{\mathbf{x}}_k+\mathbf{B}\check{\mathbf{u}}_k+\mathbf{k} \qquad \forall k\in[k_0,N] \\
\check{\mathbf{u}}_k &= \mathbf{K}\bar{\mathbf{x}}_j+\mathbf{c}_k \qquad \forall k\in[k_0,N] \\
\bar{\mathbf{x}}_\mathbf{k} &\in \mathbf{X} \qquad \forall k\in[k_0,N] \\
\check{\mathbf{u}}_\mathbf{k} &\in \mathbf{U} \qquad \forall k\in[k_0,N] \\
\bar{\mathbf{x}}_\mathbf{k} &\in \mathbf{X}\ominus\mathbf{D} \qquad \forall k\in[k_0,N] \\
(\bar{\mathbf{x}}_\mathbf{k},\xi) &\in \omega \qquad \forall k\in[k_0,N]
\end{aligned}$$

To solve each of these optimal control problems the function mpc::LBMPC::initMPC initialized the control problem. The resulting optimization problem is then solved by a (predefined) minimization routine.

Then the first value of the computed control is implemented and the optimization horizon is shifted forward in time. This allows the procedure to be applied iteratively and computes a (suboptimal) infinite horizon control.

Note that the function mpc::LBMPC::updatedMPC can be used to computer a control signal for the next time-step.

.

Definition at line 32 of file lbmpc.h.

8.4.2 Constructor & Destructor Documentation

8.4.2.1 mpc::LBMPC::LBMPC (ros::NodeHandle *node*)

Constructorb function.

Parameters

ros::NodeHandle	node Node handle

Definition at line 5 of file lbmpc.cpp.

```
                                  : nh_(node)
{
        model_ = 0;
        optimizer_ = 0;
        simulator_ = 0;
        enable_record_ = true;
}
```

8.4.3 Member Function Documentation

8.4.3.1 bool mpc::LBMPC::initMPC ( ) [virtual]

Function to initialize the calculation of the MPC algorithm. The function reads all required parameters from ROS' parameter server that has been previously loaded from a configuration YAML file, and performs all the initial calculations of variables to be used in the optimization problem.

Returns

Label that indicates if the MPC is initialized with success

Implements mpc::ModelPredictiveControl.

Definition at line 65 of file lbmpc.cpp.

```
{
        // Initialization of MPC solution
        mpc_solution_ = new double[variables_];
        control_signal_ = new double[inputs_];
        u_reference_ = Eigen::MatrixXd::Zero(inputs_, 1);
        infeasibility_counter_ = 0;
        x_.resize(states_);
        xref_.resize(states_);
        u_.resize(inputs_);


        // Get the nominal dynamic model matrices
        A_nominal_ = Eigen::MatrixXd::Zero(states_, states_);
        A_estimated_ = Eigen::MatrixXd::Zero(states_, states_);

        B_nominal_ = Eigen::MatrixXd::Zero(states_, inputs_);
        B_estimated_ = Eigen::MatrixXd::Zero(states_, inputs_);

        d_nominal_ = Eigen::MatrixXd::Zero(states_, 1);
        d_estimated_ = Eigen::MatrixXd::Zero(states_, 1);

        C_nominal_ = Eigen::MatrixXd::Zero(outputs_, states_);
        if (!model_->computeDynamicModel(A_nominal_, B_nominal_,
      C_nominal_)) {
                ROS_ERROR("Could not compute the nominal dynamic model of the
       linear system.");
                return false;
        }


        // Get the feedback gain that serves to limit the effects of model
       uncertainty
        K_ =  Eigen::MatrixXd::Zero(inputs_, states_);
        XmlRpc::XmlRpcValue feedback_gain_list;
        nh_.getParam("feedback_gain/data", feedback_gain_list);
        ROS_ASSERT(feedback_gain_list.getType() ==
      XmlRpc::XmlRpcValue::TypeArray);
        ROS_ASSERT(feedback_gain_list.size() == inputs_ * states_
      );
        int z = 0;
```

```
        for (int i = 0; i < inputs_; i++) {
                for (int j = 0; j < states_; j++) {
                        ROS_ASSERT(feedback_gain_list[z].getType() ==
      XmlRpc::XmlRpcValue::TypeDouble);
                        K_(i, j) = static_cast<double>(feedback_gain_list[z]);
                        z++;
                }
        }


        // Get the weight matrices of the cost function
        Q_ = Eigen::MatrixXd::Zero(states_, states_);
        P_ = Eigen::MatrixXd::Zero(states_, states_);
        R_ = Eigen::MatrixXd::Zero(inputs_, inputs_);

        //TODO: To get numerical values of Q, R and P matrices through the
       parameter serves
        XmlRpc::XmlRpcValue Q_list, P_list, R_list;
        nh_.getParam("optimizer/states_error_weight_matrix/data", Q_list);
        ROS_ASSERT(Q_list.getType() == XmlRpc::XmlRpcValue::TypeArray);
        ROS_ASSERT(Q_list.size() == states_ * states_);

        nh_.getParam("optimizer/terminal_state_weight_matrix/data", P_list);
        ROS_ASSERT(P_list.getType() == XmlRpc::XmlRpcValue::TypeArray);
        ROS_ASSERT(P_list.size() == states_ * states_);

        z = 0;
        for (int i = 0; i < states_; i++) {
                for (int j = 0; j < states_; j++) {
                        ROS_ASSERT(Q_list[z].getType() ==
      XmlRpc::XmlRpcValue::TypeDouble);
                        Q_(i, j) = static_cast<double>(Q_list[z]);

                        ROS_ASSERT(P_list[z].getType() ==
      XmlRpc::XmlRpcValue::TypeDouble);
                        P_(i, j) = static_cast<double>(P_list[z]);
                        z++;
                }
        }

        nh_.getParam("optimizer/input_error_weight_matrix/data", R_list);
        ROS_ASSERT(R_list.getType() == XmlRpc::XmlRpcValue::TypeArray);
        ROS_ASSERT(R_list.size() == inputs_ * inputs_);
        z = 0;
        for (int i = 0; i < inputs_; i++) {
                for (int j = 0; j < inputs_; j++) {
                        ROS_ASSERT(R_list[z].getType() ==
      XmlRpc::XmlRpcValue::TypeDouble);
                        R_(i, j) = static_cast<double>(R_list[z]);
                        z++;
                }
        }


        // Creation of the states and inputs weight matrices for the quadratic
       program
        Q_bar_ = Eigen::MatrixXd::Zero((horizon_ + 1) * states_, (
      horizon_ + 1) * states_);
        R_bar_ = Eigen::MatrixXd::Zero(horizon_ * inputs_, horizon_
       * inputs_);
        for (int i = 0; i < horizon_; i++) {
                Q_bar_.block(i * states_, i * states_, states_, states_) = Q_
      ;
                R_bar_.block(i * inputs_, i * inputs_, inputs_, inputs_) = R_
      ;
        }
        Q_bar_.block(horizon_ * states_, horizon_ * states_, states_, states_)
      = P_;


        ROS_INFO("Learning-based MPC successfully initialized");
        return true;
}
```

8.4.3.2 bool mpc::LBMPC::resetMPC (mpc::model::Model * *model*, mpc::optimizer::Optimizer * *optimizer*, mpc::model::Simulator * *simulator*) [virtual]

Function to specify and set the settings of all the components within the MPC problem. The mpc::ModelPredictiveControl class can change individual parts of the MPC problem; such as the model (mpc::model::Model and derived classes), the optimizer (mpc::optimizer::Optimizer and derived classes) and, if used, the plant simulator (mpc::model::Simulator and derived classes) in order to allow different combinations of these parts when solving.

Parameters

mpc::model::-Model	*model Pointer to the model of the plant to be used in the algorithm
mpc::optimizer::-Optimizer	*optimizer Pointer to the optimization library to be used in the algorithm
mpc::model::-Simulator	*simulator Pointer to the simulator class used to predict the states

Implements mpc::ModelPredictiveControl.

Definition at line 14 of file lbmpc.cpp.

```
{
        model_ = model;
        optimizer_ = optimizer;
        simulator_ = simulator;

        // Reading of the horizon value of the model predictive control
       algorithm
        nh_.param<int>("horizon", horizon_, 30);
        ROS_INFO("Got param: horizon = %d", horizon_);

                // Ask about the enable or disable of the learning process
        if (!nh_.getParam("enable_learning_process", enable_learning_process_))
       {
                enable_learning_process_ = false;
                ROS_WARN("Could not get the enable or disable learning process,
       therefore it is disable the learning process.");
        }


        nh_.param<int>("infeasibility_hack_counter_max",
      infeasibility_hack_counter_max_, 1);
        ROS_INFO("Got param: infeasibility_hack_counter_max = %d",
      infeasibility_hack_counter_max_);

        // Reading the path and data name
        if (!nh_.getParam("path_name", path_name_)) {
                ROS_WARN("The data will not save because could not found path
       name from parameter server.");
                enable_record_ = false;
        }
        if (!nh_.getParam("data_name", data_name_)) {
                ROS_WARN("The data will not save because could not found data
       name from parameter server.");
                enable_record_ = false;
        }

        // Get the number of states, inputs and outputs of the plant
        states_ = model_->getStatesNumber();
        inputs_ = model_->getInputsNumber();
        outputs_ = model_->getOutputsNumber();

        variables_ = horizon_ * inputs_;
        optimizer_->setHorizon(horizon_);
        optimizer_->setVariableNumber(variables_
      );

        if (!optimizer_->init()) {
                ROS_INFO("Could not initialized the optimizer class.");
                return false;
        }
        constraints_ = optimizer_->getConstraintNumber
```

```
();

        ROS_INFO("Reset successful.");
        return true;
}
```

8.4.3.3 void mpc::LBMPC::updateMPC (double * *x_measured*, double * *x_reference*) [virtual]

Function to solve the optimization problem formulated in the MPC.

Parameters

*double**	x_measured state vector
*double**	x_reference reference vector

Implements mpc::ModelPredictiveControl.

Definition at line 165 of file lbmpc.cpp.

```
{
        Eigen::Map<Eigen::VectorXd> x_measured_eigen(x_measured, states_
      , 1);
        Eigen::Map<Eigen::VectorXd> x_reference_eigen(x_reference, states_
      , 1);

        // Compute the estimated dynamic model matrices
        if (enable_learning_process_) {
                //TODO a function that determines the matrix of model learned,
       i.e. A_learned and B_learned
                Eigen::MatrixXd A_learned = Eigen::MatrixXd::Zero(states_
      , states_);
                Eigen::MatrixXd B_learned = Eigen::MatrixXd::Zero(states_
      , inputs_);
                Eigen::MatrixXd d_learned = Eigen::MatrixXd::Zero(states_
      , 1);

                A_estimated_ = A_nominal_ + A_learned;
                B_estimated_ = B_nominal_ + B_learned;
                d_estimated_ = d_nominal_ + d_learned;
        }
        else {
                A_estimated_ = A_nominal_;
                B_estimated_ = B_nominal_;
                d_estimated_ = d_nominal_;
        }

        // Compute steady state control based on updated system matrices
        Eigen::JacobiSVD<Eigen::MatrixXd> SVD_B(B_estimated_,
      Eigen::ComputeThinU | Eigen::ComputeThinV);
        Eigen::MatrixXd u_reference = SVD_B.solve(x_reference_eigen -
      A_estimated_ * x_reference_eigen - d_estimated_);


        Eigen::MatrixXd A_p_Bk = A_estimated_ + B_estimated_ * K_;
        A_p_BK_pow_.push_back(Eigen::MatrixXd::Identity(states_, states_
      ));
        for (int i = 1; i < horizon_ + 1; i++) {
                Eigen::MatrixXd A_p_BK_pow_i = A_p_BK_pow_[i-1] * A_p_Bk;
                A_p_BK_pow_.push_back(A_p_BK_pow_i);
//              std::cout << A_p_BK_pow_i << " = (A+BK)^" << i <<
       std::endl;//Work it!
        }

        // Compute the hessian matrix and gradient vector for the quadratic
       program
       Eigen::MatrixXd A_x((horizon_ + 1) * states_, states_);
       Eigen::MatrixXd B_x = Eigen::MatrixXd::Zero((horizon_ + 1) * states_
      , horizon_ * inputs_);
       Eigen::MatrixXd H_x = Eigen::MatrixXd::Zero((horizon_ + 1) * states_
      , horizon_ * states_);
```

```
    Eigen::MatrixXd A_u(horizon_ * inputs_, states_);
    Eigen::MatrixXd B_u = Eigen::MatrixXd::Zero(horizon_ * inputs_,
  horizon_ * inputs_);
    Eigen::MatrixXd H_u = Eigen::MatrixXd::Zero(horizon_ * inputs_,
  horizon_ * states_);
    Eigen::MatrixXd u_s = Eigen::MatrixXd::Zero(horizon_ * inputs_,
  1);
    Eigen::MatrixXd x_s = Eigen::MatrixXd::Zero((horizon_ + 1) * states_
  , 1);
    Eigen::MatrixXd d_bar = Eigen::MatrixXd::Zero(horizon_ * states_
  , 1);
    for (int i = 0; i < horizon_ + 1; i++) {
            for (int j = 0; j < horizon_; j++) {
                    if (i == horizon_) {
                            if (j == 0)
                                    A_x.block(i * states_, 0,
  states_, states_) = A_p_BK_pow_[i];

                            if (j == 0)
                                    x_s.block(i * states_, 0, states_, 1) =
   x_reference_eigen;

                            if (i > j) {
                                    B_x.block(i * states_, j * inputs_
  , states_, inputs_) = A_p_BK_pow_[i-j-1] * B_estimated_;
                                    H_x.block(i * states_, j * states_,
  states_, states_) = A_p_BK_pow_[i-j-1];
                            }
                    }
                    else {
                            if (j == 0) {
                                    A_x.block(i * states_, 0, states_,
  states_) = A_p_BK_pow_[i];
                                    A_u.block(i * inputs_, 0,
  inputs_, states_) = K_ * A_p_BK_pow_[i];
                                    u_s.block(i * inputs_, 0, inputs_, 1) =
   u_reference;
                                    x_s.block(i * states_, 0, states_, 1) =
   x_reference_eigen;
                            }
                            if (i == j) {
                                    B_u.block(i * inputs_, j * inputs_,
  inputs_, inputs_) = Eigen::MatrixXd::Identity(inputs_, inputs_);
                                    d_bar.block(i * states_, 0, states_, 1)
   = d_estimated_;
                            }

                            if (i > j) {
                                    B_x.block(i * states_, j * inputs_,
  states_, inputs_) = A_p_BK_pow_[i-j-1] * B_estimated_;
                                    H_x.block(i * states_, j * states_,
  states_, states_) = A_p_BK_pow_[i-j-1];
                                    B_u.block(i * inputs_, j * inputs_,
  inputs_, inputs_) = K_ * A_p_BK_pow_[i-j-1] * B_estimated_;
                                    H_u.block(i * inputs_, j * states_,
  inputs_, states_) = K_ * A_p_BK_pow_[i-j-1];
                            }
                    }
            }
    }

    double hessian_matrix[horizon_ * inputs_][horizon_ * inputs_
  ];
    double gradient_vector[horizon_ * inputs_];
    Eigen::Map<Eigen::MatrixXd> H(&hessian_matrix[0][0], horizon_ * inputs_
  , horizon_ * inputs_);
    Eigen::Map<Eigen::MatrixXd> g(gradient_vector, horizon_ * inputs_, 1);

    H = B_x.transpose() * Q_bar_ * B_x + B_u.transpose() * R_bar_ * B_u;
    g = B_x.transpose() * Q_bar_ * (A_x * x_measured_eigen + H_x * d_bar -
  x_s) + B_u.transpose() * R_bar_ * (A_u * x_measured_eigen + H_u * d_bar - u_s);

//      std::cout << A_x << " = A_x" << std::endl;//Work it!
//      std::cout << A_u << " = A_u" << std::endl;//Work it!
//      std::cout << B_x << " = B_x" << std::endl;//Work it!
//      std::cout << B_u << " = B_u" << std::endl;//Work it!
//      std::cout << d_x << " = d_x" << std::endl;
//      std::cout << d_u << " = d_u" << std::endl;
//      std::cout << u_s << " = u_s" << std::endl;//Work it!
//      std::cout << x_s << " = x_s" << std::endl;//Work it!
```

```
//      std::cout << H << " = H" << std::endl;//Work it!
//      std::cout << g << " = g" << std::endl;//Work it!

        // Solve the optimization problem
//      optimizer->computeOptimization();
}
```

The documentation for this class was generated from the following files:

- /home/rene/ros_workspace/model-predictive-control/mpc/include/mpc/mpc/lbmpc.h
- /home/rene/ros_workspace/model-predictive-control/mpc/src/mpc/lbmpc.cpp

8.5 mpc::model::Model Class Reference

This is the abstract class used to create and define different process models, in a state space representation. The models can be defined as Linear Time Invariant (LTI) such as

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) \end{aligned}$$

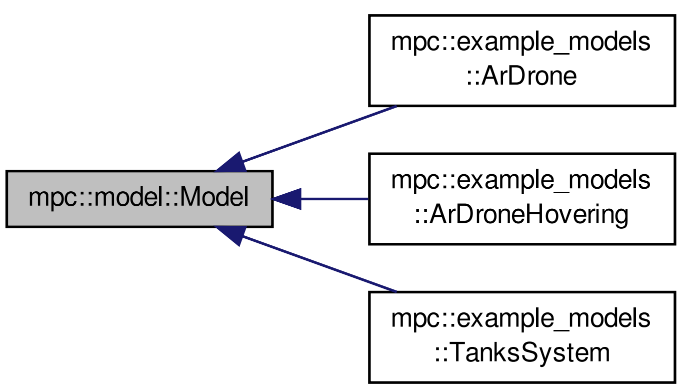
or Linear Time Variant (LTV) such as

$$\begin{aligned} \dot{x}(t) &= A(t)x(t) + B(t)u(t) \\ y(t) &= C(t)x(t) \end{aligned}$$

A boolean member variable defines the type of model used. The C matrix is not considered in the computation of the system matrices because this matrix is not used by the MPC algorithm, in an effort to reduce computation time.

`#include <model.h>`

Inheritance diagram for mpc::model::Model:



Public Member Functions

- Model ()

Constructor function.

- ∼Model ()

    Destructor function.

- virtual void setLinearizationPoints (double ∗op_states)=0

    After the MPC makes an iteration, this function is used to set the current state as the new linearization points for a LTV model into global variables.

- virtual bool computeLinearSystem (Eigen::MatrixXd &A, Eigen::MatrixXd &B)=0

    Function to compute the matrices for a Linear model process. If the model is a LTI model, the function can be just defined to set the values of the model matrices.

- virtual int getStatesNumber () const

    Get the states number of the dynamic model.

- virtual int getInputsNumber () const

    Get the inputs number of the dynamic model.

- virtual int getOutputsNumber () const

    Get the outputs number of the dynamic model.

- virtual bool **setStates** (const double ∗states) const
- virtual bool **setInputs** (const double ∗inputs) const
- virtual bool getModelType () const

    Function to identify if the model is LTI or LTV.

- virtual double ∗ getOperationPointsStates () const

    Function that returns the current value of the operation points for the states.

- virtual double ∗ getOperationPointsInputs () const

    Function that returns the current value of the operation points for the inputs.

Protected Attributes

- Eigen::MatrixXd A_

    State matrix of the dynamic model.

- Eigen::MatrixXd B_

    Input matrix of the dynamic model.

- int num_states_

    Number of states of the dynamic model.

- int num_inputs_

    Number of inputs of the dynamic model.

- int num_outputs_

    Number of outputs of the dynamic model.

- double ∗ op_point_states_
- double ∗ op_point_input_
- bool time_variant_

    Boolean to check if the model is time variant or not (True = LTV)

8.5.1 Detailed Description

This is the abstract class used to create and define different process models, in a state space representation. The models can be defined as Linear Time Invariant (LTI) such as

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) \end{aligned}$$

or Linear Time Variant (LTV) such as

$$\begin{aligned} \dot{x}(t) &= A(t)x(t) + B(t)u(t) \\ y(t) &= C(t)x(t) \end{aligned}$$

A boolean member variable defines the type of model used. The C matrix is not considered in the computation of the system matrices because this matrix is not used by the MPC algorithm, in an effort to reduce computation time.

Definition at line 34 of file model.h.

The documentation for this class was generated from the following file:

- /home/rene/ros_workspace/model-predictive-control/mpc/include/mpc/model/model.h

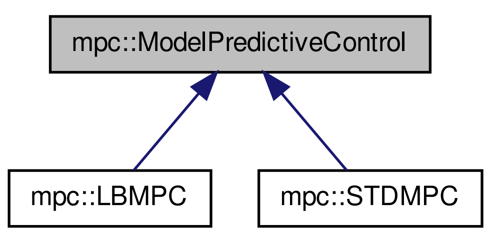
8.6 mpc::ModelPredictiveControl Class Reference

This class serves as a base class in order to expand the functionality of the library and implement different sorts of MPC algorithms. The methods defined here are conceived in the simplest way possible to allow different implementations in the derived classes.

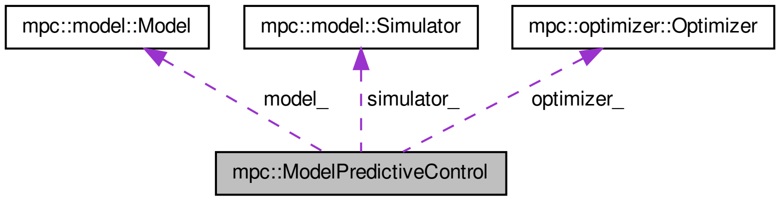
.

`#include <model_predictive_control.h>`

Inheritance diagram for mpc::ModelPredictiveControl:



Collaboration diagram for mpc::ModelPredictiveControl:



Public Member Functions

- ModelPredictiveControl ()

  Constructor function.
- ∼ModelPredictiveControl ()

  Destructor function.
- virtual bool resetMPC (mpc::model::Model ∗model, mpc::optimizer::Optimizer ∗optimizer, mpc::model::Simulator ∗simulator)=0

  Function to specify and set the settings of all the components within the MPC problem. The mpc::ModelPredictiveControl class can change individual parts of the MPC problem; such as the model (mpc::model::Model and derived classes), the optimizer (mpc::optimizer::Optimizer and derived classes) and, if used, the plant simulator (mpc::model::Simulator and derived classes) in order to allow different combinations of these parts when solving.
- virtual bool initMPC ()=0

  Function to initialize the calculation of the MPC algorithm. The function reads all required parameters from ROS' parameter server that has been previously loaded from a configuration YAML file, and performs all the initial calculations of variables to be used in the optimization problem.
- virtual void updateMPC (double ∗x_measured, double ∗x_reference)=0

  Function to update the MPC algorithm for the next iteration. The parameters defined and calculated in mpc::ModelPredictiveControl::initMPC() are used together with the methods taken from the MPC class components (mpc::model::Model, mpc::optimizer::Optimizer and mpc::model::Simulator) to find a solution to the optimization problem. This is where the different variants of MPC algorithms can be implemented in a source file from a derived class.
- virtual double ∗ getControlSignal () const

  Function to get the control signal generated for the MPC. As the MPC algorithm states, the optimization process yields the control signals for a range of times defined by the prediction horizon, but only the current control signal is applied to the plant. This function returns the control signal for the current time.
- virtual void writeToDisc ()

  Function to write the data of the MPC in a text file.

Protected Attributes

- mpc::model::Model ∗ model_

  Pointer of dynamic model of the system.
- mpc::model::Simulator ∗ simulator_

  Pointer of the simulator of the system.
- mpc::optimizer::Optimizer ∗ optimizer_

Pointer of the optimizer of the MPC.
- int states_
  Number of states of the dynamic model.
- int inputs_
  Number of inputs of the dynamic model.
- int outputs_
  Number of outputs of the dynamic model.
- int horizon_
  Horizon of prediction of the dynamic model.
- int variables_
  Number of variables, i.e inputs ∗ horizon.
- int constraints_
  Number of constraints.
- double ∗ operation_states_
  Vector of the operation points for the states in case of a LTI model.
- double ∗ operation_inputs_
  Vector of the operation points for the inputs in case of a LTI model.
- int infeasibility_counter_
  Infeasibility counter in the solution.
- int infeasibility_hack_counter_max_
  Maximun value of the infeasibility counter.
- Eigen::MatrixXd Q_
  States error weight matrix.
- Eigen::MatrixXd P_
  Terminal states error weight matrix.
- Eigen::MatrixXd R_
  Input error weight matrix.
- double ∗ mpc_solution_
  Vector of the MPC solution.
- Eigen::MatrixXd u_reference_
  Stationary control signal for the reference state vector.
- double ∗ control_signal_
  Control signal computes for MPC.
- std::vector< int > t_
  Data of the time vector of the system.
- std::vector< std::vector
  < double > > x_
  Data of the state vector of the system.
- std::vector< std::vector
  < double > > xref_
  Data of the reference state vector of the system.
- std::vector< std::vector
  < double > > u_
  Data of the control signal vector of the system.
- std::string path_name_
  Path where the data will be save.
- std::string data_name_
  Name of the file where the data will be save.
- bool enable_record_
  Label that indicates if it will be save the data.

8.6.1 Detailed Description

This class serves as a base class in order to expand the functionality of the library and implement different sorts of MPC algorithms. The methods defined here are conceived in the simplest way possible to allow different implementations in the derived classes.

.

Definition at line 24 of file model_predictive_control.h.

The documentation for this class was generated from the following file:

- /home/rene/ros_workspace/model-predictive-control/mpc/include/mpc/mpc/model_predictive_control.h

8.7 mpc.msg._MPCState.MPCState Class Reference

Public Member Functions

- def __init__
- def serialize
- def deserialize
- def serialize_numpy
- def deserialize_numpy

Public Attributes

- **header**
- **states**
- **reference_states**
- **inputs**

8.7.1 Detailed Description

Definition at line 9 of file _MPCState.py.

8.7.2 Constructor & Destructor Documentation

8.7.2.1 def mpc.msg._MPCState.MPCState.__init__ (*self*, *args*, *kwds*)

```
Constructor. Any message fields that are implicitly/explicitly
set to None will be assigned a default value. The recommend
use is keyword arguments as this is more robust to future message
changes.  You cannot mix in-order arguments and keyword arguments.

The available fields are:
   header,states,reference_states,inputs

:param args: complete set of field values, in .msg order
:param kwds: use keyword arguments corresponding to message field names
to set specific fields.
```

Definition at line 40 of file _MPCState.py.

```

  def __init__(self, *args, **kwds):
    """
    Constructor. Any message fields that are implicitly/explicitly
    set to None will be assigned a default value. The recommend
    use is keyword arguments as this is more robust to future message
    changes.  You cannot mix in-order arguments and keyword arguments.

    The available fields are:
       header,states,reference_states,inputs

    :param args: complete set of field values, in .msg order
    :param kwds: use keyword arguments corresponding to message field names
    to set specific fields.
    """
    if args or kwds:
      super(MPCState, self).__init__(*args, **kwds)
      #message fields cannot be None, assign default values for those that are
      if self.header is None:
        self.header = std_msgs.msg.Header()
      if self.states is None:
        self.states = []
      if self.reference_states is None:
        self.reference_states = []
      if self.inputs is None:
        self.inputs = []
    else:
      self.header = std_msgs.msg.Header()
      self.states = []
      self.reference_states = []
      self.inputs = []
```

8.7.3 Member Function Documentation

8.7.3.1 def mpc.msg._MPCState.MPCState.deserialize (*self*, *str*)

```
unpack serialized message in str into this message instance
:param str: byte array of serialized message, ``str``
```

Definition at line 106 of file _MPCState.py.

```
106
107   def deserialize(self, str):
108     """
109     unpack serialized message in str into this message instance
110     :param str: byte array of serialized message, ``str``
111     """
112     try:
113       if self.header is None:
114         self.header = std_msgs.msg.Header()
115       end = 0
116       _x = self
117       start = end
118       end += 12
119       (_x.header.seq, _x.header.stamp.secs, _x.header.stamp.nsecs,) =
      _struct_3I.unpack(str[start:end])
120       start = end
121       end += 4
122       (length,) = _struct_I.unpack(str[start:end])
123       start = end
124       end += length
125       if python3:
126         self.header.frame_id = str[start:end].decode('utf-8')
127       else:
128         self.header.frame_id = str[start:end]
129       start = end
130       end += 4
131       (length,) = _struct_I.unpack(str[start:end])
132       pattern = '<%sd'%length
133       start = end
134       end += struct.calcsize(pattern)
135       self.states = struct.unpack(pattern, str[start:end])
136       start = end
```

```
      end += 4
      (length,) = _struct_I.unpack(str[start:end])
      pattern = '<%sd'%length
      start = end
      end += struct.calcsize(pattern)
      self.reference_states = struct.unpack(pattern, str[start:end])
      start = end
      end += 4
      (length,) = _struct_I.unpack(str[start:end])
      pattern = '<%sd'%length
      start = end
      end += struct.calcsize(pattern)
      self.inputs = struct.unpack(pattern, str[start:end])
      return self
    except struct.error as e:
      raise genpy.DeserializationError(e) #most likely buffer underfill

```

8.7.3.2 def mpc.msg._MPCState.MPCState.deserialize_numpy (*self*, *str*, *numpy*)

```
unpack serialized message in str into this message instance using numpy for array types
:param str: byte array of serialized message, ``str``
:param numpy: numpy python module
```

Definition at line 184 of file _MPCState.py.

```
184
185   def deserialize_numpy(self, str, numpy):
186     """
187     unpack serialized message in str into this message instance using numpy for array types
188     :param str: byte array of serialized message, ``str``
189     :param numpy: numpy python module
190     """
191     try:
192       if self.header is None:
193         self.header = std_msgs.msg.Header()
194       end = 0
195       _x = self
196       start = end
197       end += 12
198       (_x.header.seq, _x.header.stamp.secs, _x.header.stamp.nsecs,) = _struct_3I.unpack(str[start:end])
199       start = end
200       end += 4
201       (length,) = _struct_I.unpack(str[start:end])
202       start = end
203       end += length
204       if python3:
205         self.header.frame_id = str[start:end].decode('utf-8')
206       else:
207         self.header.frame_id = str[start:end]
208       start = end
209       end += 4
210       (length,) = _struct_I.unpack(str[start:end])
211       pattern = '<%sd'%length
212       start = end
213       end += struct.calcsize(pattern)
214       self.states = numpy.frombuffer(str[start:end], dtype=numpy.float64, count=length)
215       start = end
216       end += 4
217       (length,) = _struct_I.unpack(str[start:end])
218       pattern = '<%sd'%length
219       start = end
220       end += struct.calcsize(pattern)
221       self.reference_states = numpy.frombuffer(str[start:end], dtype=numpy.float64, count=length)
222       start = end
223       end += 4
224       (length,) = _struct_I.unpack(str[start:end])
225       pattern = '<%sd'%length
```

```
226       start = end
227       end += struct.calcsize(pattern)
228       self.inputs = numpy.frombuffer(str[start:end], dtype=numpy.float64,
      count=length)
229       return self
230     except struct.error as e:
231       raise genpy.DeserializationError(e) #most likely buffer underfill
```

8.7.3.3 def mpc.msg._MPCState.MPCState.serialize (*self*, *buff*)

```
serialize message into buffer
:param buff: buffer, ``StringIO``
```

Definition at line 77 of file _MPCState.py.

```
77
78   def serialize(self, buff):
79     """
80     serialize message into buffer
81     :param buff: buffer, ``StringIO``
82     """
83     try:
84       _x = self
85       buff.write(_struct_3I.pack(_x.header.seq, _x.header.stamp.secs,
     _x.header.stamp.nsecs))
86       _x = self.header.frame_id
87       length = len(_x)
88       if python3 or type(_x) == unicode:
89         _x = _x.encode('utf-8')
90         length = len(_x)
91       buff.write(struct.pack('<I%ss'%length, length, _x))
92       length = len(self.states)
93       buff.write(_struct_I.pack(length))
94       pattern = '<%sd'%length
95       buff.write(struct.pack(pattern, *self.states))
96       length = len(self.reference_states)
97       buff.write(_struct_I.pack(length))
98       pattern = '<%sd'%length
99       buff.write(struct.pack(pattern, *self.reference_states))
100      length = len(self.inputs)
101      buff.write(_struct_I.pack(length))
102      pattern = '<%sd'%length
103      buff.write(struct.pack(pattern, *self.inputs))
104    except struct.error as se: self._check_types(se)
105    except TypeError as te: self._check_types(te)
```

8.7.3.4 def mpc.msg._MPCState.MPCState.serialize_numpy (*self*, *buff*, *numpy*)

```
serialize message with numpy array types into buffer
:param buff: buffer, ``StringIO``
:param numpy: numpy python module
```

Definition at line 154 of file _MPCState.py.

```
154
155  def serialize_numpy(self, buff, numpy):
156    """
157    serialize message with numpy array types into buffer
158    :param buff: buffer, ``StringIO``
159    :param numpy: numpy python module
160    """
161    try:
162      _x = self
163      buff.write(_struct_3I.pack(_x.header.seq, _x.header.stamp.secs,
     _x.header.stamp.nsecs))
164      _x = self.header.frame_id
```

```
      length = len(_x)
      if python3 or type(_x) == unicode:
        _x = _x.encode('utf-8')
        length = len(_x)
      buff.write(struct.pack('<I%ss'%length, length, _x))
      length = len(self.states)
      buff.write(_struct_I.pack(length))
      pattern = '<%sd'%length
      buff.write(self.states.tostring())
      length = len(self.reference_states)
      buff.write(_struct_I.pack(length))
      pattern = '<%sd'%length
      buff.write(self.reference_states.tostring())
      length = len(self.inputs)
      buff.write(_struct_I.pack(length))
      pattern = '<%sd'%length
      buff.write(self.inputs.tostring())
    except struct.error as se: self._check_types(se)
    except TypeError as te: self._check_types(te)
```

The documentation for this class was generated from the following file:

- /home/rene/ros_workspace/model-predictive-control/mpc/src/mpc/msg/_MPCState.py

8.8 mpc::optimizer::Optimizer Class Reference

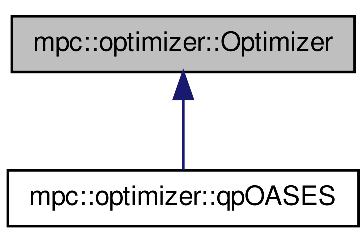
Abstract class to define the optimization algorithm for Model Predictive Control. This class acts as an interface to use a defined optimization solver software as a part of this library in order to provide different solver options for the end user to solve the basic optimization problem that rises in MPC. As more solvers are adapted to this library with this class, more options to try different optimization methods are available to select the most suitable one depending on each case. The basic

$$\begin{aligned} \text{Minimize } F(x) & \\ \text{subject to } G(x) &= 0 \\ H(x) &\geq 0 \end{aligned}$$

As more solvers are adapted to this library with this class, more options to try different optimization methods are available to select the most suitable one depending on each case.

`#include <optimizer.h>`

Inheritance diagram for mpc::optimizer::Optimizer:



Public Member Functions

- Optimizer ()

  Constructor function.
- ∼Optimizer ()

  Destructor function.
- virtual bool init ()=0

  Function to perform the initialization of optimizer, if this applies.
- virtual bool computeOpt (double ∗H, double ∗g, double ∗G, double ∗lb, double ∗ub, double ∗lbG, double ∗ubG, double cputime)=0

  Function to compute the optimization algorithm associated to the MPC problem.
- virtual double ∗ getOptimalSolution ()=0

  Get the vector of optimal or sub-optimal solutions calculated by the mpc::optimizer::Optimizer::computeOpt() function (optimality of the function is defined by the solver that is adapted).
- virtual int getConstraintNumber () const

  Get the number of constraints.
- virtual int getVariableNumber () const

  Get the number of variables, i.e inputs ∗ horizon.
- virtual void setHorizon (int horizon)

  Set the horizon of the MPC.
- virtual void setVariableNumber (int variables)

  Set the number of variables, i.e inputs ∗ horizon.

Public Attributes

- int variables_

  Number of variables, i.e inputs ∗ horizon.
- int constraints_

  Number of constraints.
- int horizon_

  Horizon of MPC.

8.8.1 Detailed Description

Abstract class to define the optimization algorithm for Model Predictive Control. This class acts as an interface to use a defined optimization solver software as a part of this library in order to provide different solver options for the end user to solve the basic optimization problem that rises in MPC. As more solvers are adapted to this library with this class, more options to try different optimization methods are available to select the most suitable one depending on each case. The basic

$$\begin{aligned} \text{Minimize } F(x) & \\ \text{subject to } G(x) &= 0 \\ H(x) &\geq 0 \end{aligned}$$

As more solvers are adapted to this library with this class, more options to try different optimization methods are available to select the most suitable one depending on each case.

Definition at line 31 of file optimizer.h.

The documentation for this class was generated from the following file:

- /home/rene/ros_workspace/model-predictive-control/mpc/include/mpc/optimizer/optimizer.h

8.9 mpc::optimizer::qpOASES Class Reference

Class to interface the qpOASES library This class gives an interface with qpOASES library in order to implement a quadratic program using online active set strategy for MPC controller. qpOASES solve a convex optimization class of the following form

$$\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x} + \mathbf{x}^T\mathbf{g}(\mathbf{x_0})$$

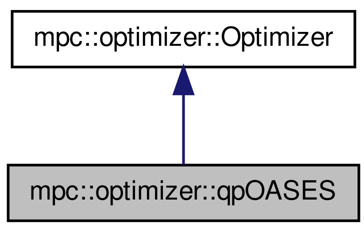
suject to

$$\begin{array}{rcl} lbG(\mathbf{x_0}) \leq & \mathbf{G}\mathbf{x} & \leq ubG(\mathbf{x_0}) \\ lb(\mathbf{x_0}) \leq & \mathbf{x} & \leq ub(\mathbf{x_0}) \end{array}$$

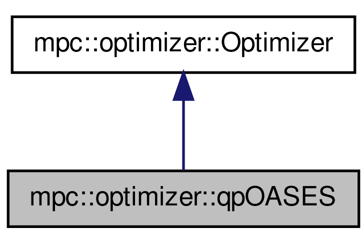
.

```
#include <qpOASES.h>
```

Inheritance diagram for mpc::optimizer::qpOASES:



Collaboration diagram for mpc::optimizer::qpOASES:



Public Member Functions

- qpOASES (ros::NodeHandle node_handle)

Constructor function.

- ∼qpOASES ()

    Destructor function.

- virtual bool init ()

    Function to define the initialization of qpOASES optimizer.

- virtual bool computeOpt (double ∗H, double ∗g, double ∗G, double ∗lb, double ∗ub, double ∗lbG, double ∗ubG, double cputime)

    Function to solve the optimization problem formulated in the MPC.

- double ∗ getOptimalSolution ()

    Get the vector of optimal solutions calculated by qpOASES.

Protected Attributes

- double ∗ optimal_solution_

    Optimal solution obtained with the implementation of qpOASES.

Additional Inherited Members

8.9.1 Detailed Description

Class to interface the qpOASES library This class gives an interface with qpOASES library in order to implement a quadratic program using online active set strategy for MPC controller. qpOASES solve a convex optimization class of the following form

$$\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x} + \mathbf{x}^T\mathbf{g}(\mathbf{x_0})$$

suject to

$$\begin{array}{rcl} lbG(\mathbf{x_0}) \leq & \mathbf{G}\mathbf{x} & \leq ubG(\mathbf{x_0}) \\ lb(\mathbf{x_0}) \leq & \mathbf{x} & \leq ub(\mathbf{x_0}) \end{array}$$

.

Definition at line 36 of file qpOASES.h.

The documentation for this class was generated from the following files:

- /home/rene/ros_workspace/model-predictive-control/mpc/include/mpc/optimizer/qpOASES.h
- /home/rene/ros_workspace/model-predictive-control/mpc/src/optimizer/qpOASES.cpp

8.10 mpc::model::Simulator Class Reference

This class provides methods to simulate a given model of a process defined by a class mpc::model::Model object This class provides methods to simulate a given model

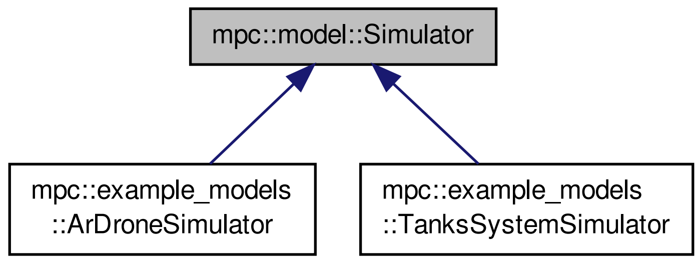
$$\dot{x}(t) = Ax(t) + Bu(t)$$

$$y(t) = Cx(t)$$

on a fixed prediction horizon interval $[t_0, t_N]$ with initial value $x(t_0, x_0) = x_0$ and given control $u(\cdot, x_0)$. That is, for a given class mpc::model::Model object and a given control u the simulator can solve the differential or difference equation forward in time.

```
#include <simulator.h>
```

Inheritance diagram for mpc::model::Simulator:



Public Member Functions

- Simulator ()

  Constructor function.
- ∼Simulator ()

  Destructor function.
- virtual double ∗ simulatePlant (double ∗state_vect, double ∗input_vect, double sampling_time)=0

  Function used to simulate the specified plant.

Protected Attributes

- double ∗ new_state_

  New state vector.

8.10.1 Detailed Description

This class provides methods to simulate a given model of a process defined by a class mpc::model::Model object This class provides methods to simulate a given model

$$\dot{x}(t) = Ax(t) + Bu(t)$$

$$y(t) = Cx(t)$$

on a fixed prediction horizon interval $[t_0, t_N]$ with initial value $x(t_0, x_0) = x_0$ and given control $u(\cdot, x_0)$. That is, for a given class mpc::model::Model object and a given control u the simulator can solve the differential or difference equation forward in time.

Definition at line 18 of file simulator.h.

8.10.2 Member Function Documentation

8.10.2.1 virtual double∗ mpc::model::Simulator::simulatePlant (double ∗ *state_vect*, double ∗ *input_vect*, double *sampling_time*) [pure virtual]

Function used to simulate the specified plant.

Parameters

double∗	state_vect State vector
double∗	input_vect Input vector
double	sampling_time Sampling time

Returns

double∗ New state vector

Implemented in mpc::example_models::TanksSystemSimulator, and mpc::example_models::ArDroneSimulator.

The documentation for this class was generated from the following file:

- /home/rene/ros_workspace/model-predictive-control/mpc/include/mpc/model/simulator.h

8.11 mpc::STDMPC Class Reference

Class for solving the explicit model predictive control problem

The aim of this class is to solve the explicit model predictive control of the following form:

$$\begin{aligned}
\min_{\mathbf{u}_{k_0},\cdots,\mathbf{u}_{k_0+N-1}} J_N(x,u) &= \frac{1}{2}(\mathbf{x}_{k_0+N}-\mathbf{x}_{ref})^T\mathbf{P}(\mathbf{x}_{k_0+N}-\mathbf{x}_{ref})+\frac{1}{2}\sum_{k=k_0}^{k_0+N-1}(\mathbf{x}_k-\mathbf{x}_{ref})^T\mathbf{Q}(\mathbf{x}_k-\mathbf{x}_{ref})+(\mathbf{u}_k-\mathbf{u}_{ref})^T\mathbf{R}(\mathbf{u}_k-\mathbf{u}_{ref}) \\
\mathbf{x}_{k_0} &= \omega_0(k_0) \\
\mathbf{x}_{k+1} &= \mathbf{A}\mathbf{x}_k+\mathbf{B}\mathbf{u}_k \qquad \forall k\in[k_0,N] \\
\bar{x} &\leq \mathbf{M}\mathbf{x}_k \qquad \forall k\in[k_0,N] \\
\bar{u} &\leq \mathbf{N}\mathbf{u}_k \qquad \forall k\in[k_0,N]
\end{aligned}$$

To solve each of these optimal control problems the function mpc::STDMPC::initMPC initialized the control problem. The resulting optimization problem is then solved by a (predefined) minimization routine.

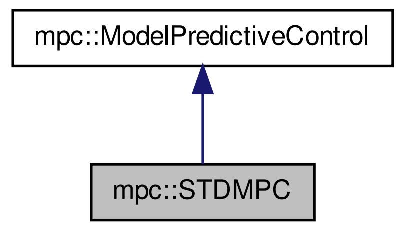
Then the first value of the computed control is implemented and the optimization horizon is shifted forward in time. This allows the procedure to be applied iteratively and computes a (suboptimal) infinite horizon control.

Note that the function mpc::STDMPC::updateMPC() can be used to compute a control signal for the next time-step.

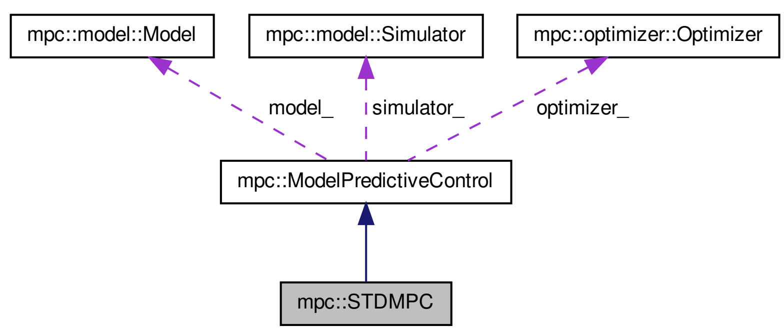
.

```
#include <stdmpc.h>
```

Inheritance diagram for mpc::STDMPC:



Collaboration diagram for mpc::STDMPC:



Public Member Functions

- STDMPC (ros::NodeHandle node_handle)

  Constructor function.
- ∼STDMPC ()

  Destructor function.
- virtual bool resetMPC (mpc::model::Model ∗model, mpc::optimizer::Optimizer ∗optimizer, mpc::model::Simulator ∗simulator)

  Function to specify and set the settings of all the components within the MPC problem. The mpc::ModelPredictiveControl class can change individual parts of the MPC problem; such as the model (mpc::model::Model and derived classes), the optimizer (mpc::optimizer::Optimizer and derived classes) and, if used, the plant simulator (mpc::model::Simulator and derived classes) in order to allow different combinations of these parts when solving.
- virtual bool initMPC ()

  Function to initialize the calculation of the MPC algorithm. The function reads all required parameters from ROS' parameter server that has been previously loaded from a configuration YAML file, and performs all the initial calculations of variables to be used in the optimization problem.

- virtual void updateMPC (double *x_measured, double *x_reference)

  Function to solve the optimization problem formulated in the MPC.

Additional Inherited Members

8.11.1 Detailed Description

Class for solving the explicit model predictive control problem

The aim of this class is to solve the explicit model predictive control of the following form:

$$\begin{aligned}
\min_{\mathbf{u}_{k_0},\cdots,\mathbf{u}_{k_0+N-1}} J_N(x,u) &= \frac{1}{2}(\mathbf{x}_{k_0+N}-\mathbf{x}_{ref})^T\mathbf{P}(\mathbf{x}_{k_0+N}-\mathbf{x}_{ref})+\frac{1}{2}\sum_{k=k_0}^{k_0+N-1}(\mathbf{x}_k-\mathbf{x}_{ref})^T\mathbf{Q}(\mathbf{x}_k-\mathbf{x}_{ref})+(\mathbf{u}_k-\mathbf{u}_{ref})^T\mathbf{R}(\mathbf{u}_k-\mathbf{u}_{ref}) \\
\mathbf{x}_{k_0} &= \omega_0(k_0) \\
\mathbf{x}_{k+1} &= \mathbf{A}\mathbf{x}_k+\mathbf{B}\mathbf{u}_k \qquad \forall k\in[k_0,N] \\
\bar{x} &\leq \mathbf{M}\mathbf{x}_k \qquad \forall k\in[k_0,N] \\
\bar{u} &\leq \mathbf{N}\mathbf{u}_k \qquad \forall k\in[k_0,N]
\end{aligned}$$

To solve each of these optimal control problems the function mpc::STDMPC::initMPC initialized the control problem. The resulting optimization problem is then solved by a (predefined) minimization routine.

Then the first value of the computed control is implemented and the optimization horizon is shifted forward in time. This allows the procedure to be applied iteratively and computes a (suboptimal) infinite horizon control.

Note that the function mpc::STDMPC::updateMPC() can be used to compute a control signal for the next time-step.

.

Definition at line 35 of file stdmpc.h.

The documentation for this class was generated from the following files:

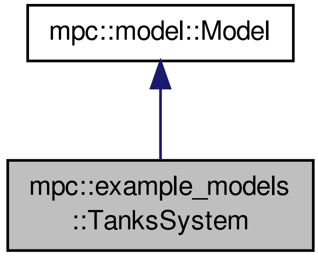
- /home/rene/ros_workspace/model-predictive-control/mpc/include/mpc/mpc/stdmpc.h
- /home/rene/ros_workspace/model-predictive-control/mpc/src/mpc/stdmpc.cpp

8.12 mpc::example_models::TanksSystem Class Reference

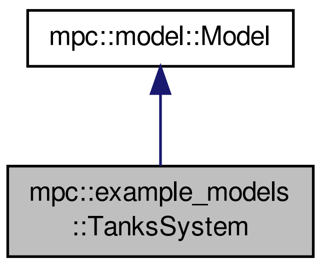
Class to define the process model of the tank system available at Simon Bolivar University's Automatic Control Lab.

```
#include <tanks_system.h>
```

Inheritance diagram for mpc::example_models::TanksSystem:



Collaboration diagram for mpc::example_models::TanksSystem:



Public Member Functions

- TanksSystem ()

  Constructor function.
- ∼TanksSystem ()

  Destructor function.
- virtual void setLinearizationPoints (double ∗op_states)

  After the MPC makes an iteration, this function is used to set the new linearization points for a LTV model into global variables.
- virtual bool computeLinearSystem (Eigen::MatrixXd &A, Eigen::MatrixXd &B)

  Function to compute the dynamic model of the system.

Additional Inherited Members

8.12.1 Detailed Description

Class to define the process model of the tank system available at Simon Bolivar University's Automatic Control Lab.

Definition at line 25 of file tanks_system.h.

The documentation for this class was generated from the following files:

- /home/rene/ros_workspace/model-predictive-control/mpc/include/mpc/example_models/tanks_system.h
- /home/rene/ros_workspace/model-predictive-control/mpc/src/example_models/tanks_system.cpp

8.13 mpc::example_models::TanksSystemSimulator Class Reference

This class provides methods to simulate a example model of tanks system defined by a class mpc::model::Model object

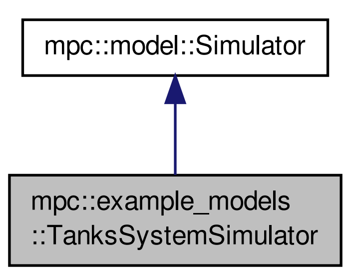
$$\dot{x}(t) = Ax(t) + Bu(t)$$

$$y(t) = Cx(t)$$

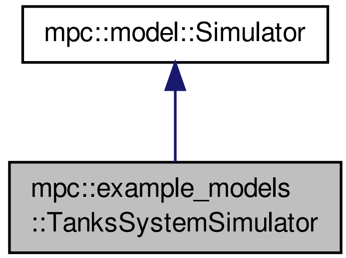
on a fixed prediction horizon interval $[t_0, t_N]$ with initial value $x(t_0, x_0) = x_0$ and given control $u(\cdot, x_0)$. That is, for a given class mpc::model::Model object and a given control u the simulator can solve the differential or difference equation forward in time.

```
#include <tanks_system_simulator.h>
```

Inheritance diagram for mpc::example_models::TanksSystemSimulator:



Collaboration diagram for mpc::example_models::TanksSystemSimulator:



Public Member Functions

- TanksSystemSimulator ()
  Constructor function.
- ∼TanksSystemSimulator ()
  Destructor function.
- double ∗ simulatePlant (double ∗state_vect, double ∗input_vect, double sampling_time)
  Function used to simulate the specified plant.

Additional Inherited Members

8.13.1 Detailed Description

This class provides methods to simulate a example model of tanks system defined by a class mpc::model::Model object

$$\dot{x}(t) = Ax(t) + Bu(t)$$

$$y(t) = Cx(t)$$

on a fixed prediction horizon interval $[t_0, t_N]$ with initial value $x(t_0, x_0) = x_0$ and given control $u(\cdot, x_0)$. That is, for a given class mpc::model::Model object and a given control u the simulator can solve the differential or difference equation forward in time.

This class provides methods to simulate a example model of tanks system

Definition at line 24 of file tanks_system_simulator.h.

The documentation for this class was generated from the following files:

- /home/rene/ros_workspace/model-predictive-control/mpc/include/mpc/example_models/tanks_system_simulator.-h
- /home/rene/ros_workspace/model-predictive-control/mpc/src/example_models/tanks_system_simulator.cpp