Model Predictive Control

Introduction

This document is a tutorial for the code inside *mpc.c* at the project *CControl.*
The document is going to explain the code in three steps. First step is going to explain mathematics. The second step is going to explain the mathematics in MATLAB-code. The third step is going to explain the MATLAB-code in C-code from the *mpc.c* file.

The mathematics in this document has its origin from the B.Sc thesis “Model Predictive Control for an artificial pancreas” by Matias Sørensen and Simon Kristiansen from Technical University of Denmark, Kongens Lyngby 2007.

The reason why I created this *mpc.c* file is because I have seen many MPC-projects that are applied inside a laboratory environment such as numerical software simulations or expensive physical hardware. Regardless of which, they are not suitable for low cost devices.

The goal of this document is to distribute the knowledge of practical MPC for low cost devices. This document will explain the pros and cons with MPC compared to PID. The document will also explain the rule of thumbs how to set the parameters.

The MPC algorithm inside this *mpc.c contains the following:*

- *Iterative quadratic programming*
- *Integral action*
- *Disturbance input*
- *Constraints on input*
- *Soft constraints on output*
- *Constraints on rate of movement*
- *Regularization*
- *Kalman-Bucy filter*

Daniel Mårtensson
Sweden
2025-05-03

Table of contents

Prerequisites

In order to understand this document. One must know the following:

- Arrays, variables, functions and memory allocation with C programming
- Matrix multiplication
- Vector multiplication
- Discrete state space modeling
- Numerical integrals by using code

Main

Header

There are two files for the MPC. It is the source file *mpc.c* and the header file *mpc.h*. Begin to read the header file *mpc.h.* One should observe the following functions inside code

```
bool mpc_init(MPC* mpc, const float A[], const float B[], const float C[], const float E[], const float sampleTime_mpc, const float sampleTime_kf, const float qw, const float rv, const float qz, const float s, const float Spsi_spsi, const size_t row_a, const size_t column_b, const size_t row_c, const size_t column_e, const size_t N, const size_t iterations);

void mpc_set_constraints(MPC* mpc, const float umin[], const float umax[], const float zmin[], const float zmax[], const float deltaumin[], const float deltaumax[], const float alpha, const float antiwindup);

STATUS_CODES mpc_optimize(MPC* mpc, float u[], const float r[], const float y[], const float d[], const bool integral_active);

void mpc_estimate(MPC* mpc, const float u[], const float y[], const float d[]);

bool mpc_free(MPC* mpc);
```

The function *mpc_init* is the first function you should start with. That function creates all the matrices and computes the kalman-bucy filter gain matrix K etc.

After that function has been called, then it's time for calling the function *mpc_set_constraints* for setting the limitations of the system. For example, the input cannot be more than 12-bit (0 to 4095) and the output cannot be warmer than 100 degrees.

The function *mpc_optimize* shall be placed inside the control loop such as a while loop. It will optimize the input signal that is going to be fed into the control output e.g PWM.

For every control input that has been optimized, then *mpc_estimate* must be called to estimate the next state *x* for the MPC.

The function *mpc_free* is only for deleting all allocated memory of the MPC.

Source

mpc_init

Discretization of MPC and Kalman-Bucy filter

We begin with the *mpc_init* function. The first thing that happens is that it needs to turn the time continuous state space model, as described in Equation 1, into a time discrete state space model, as described in Equation 2.

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

Equation 1: Time continuous state space model

$$x(k+1) = \bar{A}x(k) + \bar{B}u(k) + \bar{E}d(k)$$
$$y(k) = Cx(k)$$

Equation 2: Time discrete state space model

MPC works with prediction horizons and therefore it must have a discrete model. The benefit by using MPC is that it can “look” into the future and predict the future control signals. If the sample time of the discrete state space model is $Ts = 1$ second and the MPC has its prediction horizon e.g $N = 5$. That means the MPC is “looking” $Ts \times N = 5$ seconds further into the “future”. The MPC “simulates” the model behavior during control time and computes the suitable input signals. This cannot be achieved using a time continuous state space model because the sample time of time continuous model is $Ts = 0$ seconds.

The Figure 1 illustrates this MPC algorithm in action where the input is predicted with a control horizon of $N = 20$ and a discrete model of sample time $Ts = 10$.

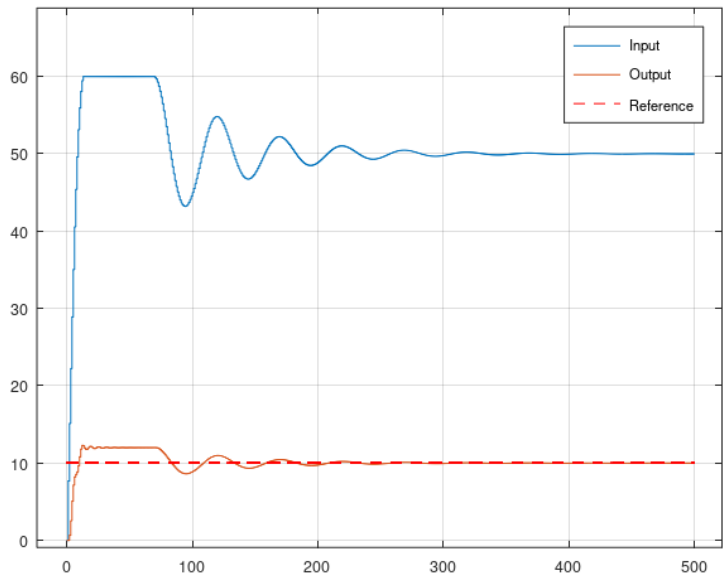

Figure 1: MPC in action from MataveControl in GNU Octave

Mathematics:

To make time continuous to time discrete, use this formula in Equation 3.

$$\begin{bmatrix} A & B & E \\ 0 & I & I \end{bmatrix} = e^{M \times Ts} \rightarrow M = \begin{bmatrix} \bar{A} & \bar{B} & \bar{E} \\ 0 & 0 & 0 \end{bmatrix}$$

Equation 3: From time continuous to time discrete

Where:

- System matrix: $A \in \Re^{nx \times nx}$. Input matrix: $B \in \Re^{nx \times nu}$. Output matrix: $C \in \Re^{nz \times nx}$
- Disturbance matrix: $E \in \Re^{nx \times nd}$. Number of states: nx. Number of inputs: nu
- Number of outputs: nz. Number of disturbances: nd. Sample time: $Ts \in \Re$

MATLAB code:

The ending *d* is the label for *discrete* matrix.

```
[Ad, Bd, Cd, Ed] = DiscreteMatrices(A, B, C, E, Ts_mpc);
[Adkf, Bdkf, Cdkf, Edkf] = DiscreteMatrices(A, B, C, E, Ts_kf);
```

C code:

```
mpc->Ad = (float*)malloc(row_a * row_a * sizeof(float));
mpc->Bd = (float*)malloc(row_a * column_b * sizeof(float));
mpc->Cd = (float*)malloc(row_c * row_a * sizeof(float));
mpc->Ed = (float*)malloc(row_a * column_e * sizeof(float));
mpc_discrete_matrices(sampleTime_mpc, A, B, C, E, mpc->Ad, mpc->Bd, mpc->Cd, mpc->Ed,
row_a, column_b, row_c, column_e);
```

The discretization is exactly the same for the Kalman-bucy filter. The only difference is that the sample time for Kalman-Bucy must be equivalent as the real time dynamics of the system. Meanwhile the sample time for MPC should be consider as a tuning parameter. The ending *kf* is the label for *kalman filter.*

```
mpc->Adkf = (float*)malloc(row_a * row_a * sizeof(float));
mpc->Bdkf = (float*)malloc(row_a * column_b * sizeof(float));
mpc->Cdkf = (float*)malloc(row_c * row_a * sizeof(float));
mpc->Edkf = (float*)malloc(row_a * column_e * sizeof(float));
mpc_discrete_matrices(sampleTime_kf, A, B, C, E, mpc->Adkf, mpc->Bdkf, mpc->Cdkf, mpc-
>Edkf, row_a, column_b, row_c, column_e);
```

Kalman-Bucy gain matrix K

Kalman-Bucy filter is used instead of regular Kalman filter. The reason is that Kalman-Bucy filter computes the kalman gain matrix K from the Algebratic Riccati Equations, meanwhile the regular Kalman filter converges the kalman gain matrix K during runtime, which takes more computational resources. *Fun fact: The Kalman-Bucy filter is actully the Kalman filter that was used inside the Apollo program.* The Kalman-Bucy filter is used because it's successor to the original Kalman filter.

Mathematics:

In order to find the Kalman-Bucy gain matrix K filter, one most solve the Discrete Algebratic Riccati Equation (DARE) by using an ODE-solver such as Runge-Kutta 4:th order and others. Begin first to replace $\bar{A} \leftarrow \bar{A}^T$ and $\bar{B} \leftarrow \bar{C}^T$. Then find the solution P.

$$P = \bar{A}^T \times P \times \bar{A} - \left(A^T \times P \times \bar{B}\right) \times \left(R + \bar{B}^T \times P \times \bar{B}\right)^{-1} \times \left(\bar{B}^T \times P \times \bar{A}\right) + Q$$

Equation 4: Discrete Algebratic Riccati Equation

Once the solution P is found, then compute the kalman gain matrix K form the Linear Quadratic Regulator (LQR) control law.

$$K = \left(\bar{B}^T \times P \times \bar{B} + R\right)^{-1} \times \left(\bar{B}^T \times P \times \bar{A}\right)$$

Equation 5: LQR control law

MATLAB code:

```
% Create the kalman gain matrix K
syse = mc.ss(delayc, Adkf, Bdkf, Cdkf);
syse.sampleTime = Ts_kf;
Qw = qw * eye(nx);
Rv = rv * eye(nz);
[K] = mc.lqe(syse, Qw, Rv);
```

C code:

```
mpc->K = (float*)malloc(row_a * row_c * sizeof(float));
mpc_kalman_gain(iterations, sampleTime_kf, mpc->Adkf, mpc->Cdkf, qw, rv, mpc->K, row_a, row_c);
```

Controllability matrix

The controllability matrix $\Phi \in \Re^{(N \times nz) \times nx}$ is a large matrix that holds $\bar{C}$ and $\bar{A}$.

Mathematics:

This is the controllability matrix in Equation 6 where N is the horizon of prediction.

$$\Phi = \begin{bmatrix} \bar{C}\,\bar{A} \\ \bar{C}\,\bar{A}^2 \\ \bar{C}\,\bar{A}^3 \\ \vdots \\ \bar{C}\,\bar{A}^N \end{bmatrix}$$

Equation 6: Controllability matrix

MATLAB code:

```
Phi = PhiMat(Ad, Cd, N);
```

C code:

```
mpc->Phi = (float*)malloc((N * row_c) * row_a * sizeof(float));
mpc_phi_matrix(mpc->Phi, mpc->Ad, mpc->Cd, row_a, row_c, N);
```

Lower triangular toeplitz of extended observability matrix

The lower triangular toeplitz of extended observability matrix $\Gamma \in \Re^{(N \times nz) \times (N \times nu)}$ is a large matrix that holds $H = \bar{C}\,\bar{A}^N\,\bar{B}$.

Mathematics:

The lower triangular toeplitz of extended observability matrix can be seen in Equation 7.

$$\Gamma = \begin{bmatrix} H_1 & 0 & 0 & \cdots & 0 \\ H_2 & H_1 & 0 & \cdots & 0 \\ H_3 & H_2 & H_1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ H_N & H_{N-1} & H_{N-2} & \cdots & H_1 \end{bmatrix}$$

Equation 7: Lower triangular toeplitz of extended observability matrix

MATLAB code:

```
Gamma = GammaMat(Ad, Bd, Cd, N);
```

C code:

```
float* Gamma = (float*)malloc((N * row_c) * (N * column_b) * sizeof(float));
mpc_gamma_matrix(Gamma, mpc->Phi, mpc->Bd, mpc->Cd, row_a, row_c, column_b, N);
```

Weight matrix

The weight matrix $Q_Z \in \Re^{(N \times nz) \times (N \times nz)}$ holds the scalar matrix $Q_z \in \Re^{nz \times nz}$. The weight matrix made for determine the weights of the MPC. Small weights will give poor performance. Large weights will give too strong outputs.

Mathematics:

The weight matrix can be seen in Equation 8.

$$Q_Z = \begin{bmatrix} Q_z & 0 & \cdots & 0 \\ 0 & Q_z & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & Q_z \end{bmatrix}$$

Equation 8: Weight matrix

MATLAB code:

```
QZ = QZMat(Qz, N, nz);
```

C code:

```
float* QZ = (float*)malloc((N * row_c) * (N * row_c) * sizeof(float));
mpc_QZ_matrix(QZ, qz, row_c, N);
```

Regularization matrix

The regularization matrix $H_S \in \Re^{(N \times nu) \times (N \times nu)}$ holds the scalar matrix $S \in \Re^{nu \times nu}$. The regularization matrix giving a smoother input signals U from of the MPC. Small values will give coarse and fast performance. Large values will give too smooth and slow performance.

Mathematics:

The regularization value and its values can be seen in Equation 9.

$$H_S = \begin{bmatrix} 2S & -S & 0 & 0 & 0 \\ -S & 2S & -S & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & -S & 2S & -S \\ 0 & 0 & 0 & -S & S \end{bmatrix}$$

Equation 9: Regularization matrix

MATLAB code:

```
S = SMat(s, nu);
HS = HSMat(S, N, nu);
```

C code:

```
float* S = (float*)malloc(column_b * column_b * sizeof(float));
mpc_S_matrix(S, s, column_b);
float* HS = (float*)malloc((N * column_b) * (N * column_b) * sizeof(float));
mpc_HS_matrix(HS, S, column_b, N);
```

QP solver matrix

The QP solver matrix $H \in \Re^{(N \times nu) \times (N \times nu)}$ is a matrix for the Quadratic Programming solver.

Mathematics:

The QP solver matrix can be seen in Equation 10.

$$H = \Gamma^T \times Q_Z \Gamma + H_S$$

Equation 10: The QP solver matrix

MATLAB code:

```
H = HMat(Gamma, QZ, HS);
```

C code:

```
float* H = (float*)malloc((N * column_b) * (N * column_b) * sizeof(float));
mpc_H_matrix(H, Gamma, QZ, HS, row_c, column_b, N);
```

Lower triangular toeplitz of extended observability matrix of disturbance

The lower triangular toeplitz of extended observability matrix of disturbance $\Gamma_d \in \Re^{(N \times nz) \times (N \times nu)}$ is a large matrix that holds $H_d = \bar{C}\,\bar{A}^N\,\bar{E}$.

Mathematics:

The lower triangular toeplitz of extended observability matrix can be seen in Equation 11.

$$\Gamma_d = \begin{bmatrix} H_{1,d} & 0 & 0 & \cdots & 0 \\ H_{2,d} & H_{1,d} & 0 & \cdots & 0 \\ H_{3,d} & H_{2,d} & H_{1,d} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ H_{N,d} & H_{N-1,d} & H_{N-2,d} & \cdots & H_{1,d} \end{bmatrix}$$

Equation 11: Lower triangular toeplitz of extended observability matrix of disturbances

MATLAB code:

```
Gammad = GammaMat(Ad, Ed, Cd, N);
```

C code:

```
float* Gammad = (float*)malloc((N * row_c) * (N * column_e) * sizeof(float));
mpc_gamma_matrix(Gammad, mpc->Phi, mpc->Ed, mpc->Cd, row_a, row_c, column_e, N);
```

QP solver gradient matrices

In order to create the gradient of the QP solver. One must first create its matrices $M_{x_0} \in \Re^{(N \times nu) \times nx}$, $M_{u_{-1}} \in \Re^{(N \times nu) \times nu}$, $M_R \in \Re^{(N \times nu) \times nz}$ and $M_D \in \Re^{(N \times nu) \times nd}$.

Mathematics:

$$M_{x_0} = \Gamma \times Q_Z \times \Phi$$
$$M_{u_{-1}} = S$$
$$M_R = \Gamma \times Q_Z$$
$$M_D = \Gamma \times Q_Z \Gamma_d$$

Equation 12: QP solver gradient matrices

MATLAB code:

```
Mx0 = Mx0Mat(Gamma, QZ, Phi);
Mum1 = Mum1Mat(N, nu, S);
MR = MRMat(Gamma, QZ);
MD = MDMat(Gamma, QZ, Gammad);
```

C code:

```
mpc->Mx0 = (float*)malloc((N * column_b) * row_a * sizeof(float));
mpc_Mx0_matrix(mpc->Mx0, Gamma, QZ, mpc->Phi, row_a, row_c, column_b, N);
mpc->Mum1 = (float*)malloc((N * column_b) * column_b * sizeof(float));
mpc_Mum1_matrix(mpc->Mum1, S, column_b, N);
mpc->MR = (float*)malloc((N * column_b) * (N * row_c) * sizeof(float));
mpc_MR_matrix(mpc->MR, Gamma, QZ, row_c, column_b, N);
mpc->MD = (float*)malloc((N * column_b) * (N * column_e) * sizeof(float));
mpc_MD_matrix(mpc->MD, Gamma, mpc->Gammad, QZ, row_c, column_b, column_e, N);
```

Rate of movement matrix

The rate of movement matrix $\Lambda \in \Re^{((N-1)\times nu)\times(N\times nu)}$ holds the identity matrix $I \in \Re^{nu\times nu}$. The rate of movement matrix controls how fast the input U can change from the MPC.

Mathematics:

The rate of movement matrix can be seen in Equation 13.

$$\Lambda = \begin{bmatrix} -I & I & 0 & 0 & 0 \\ 0 & -I & I & 0 & 0 \\ 0 & 0 & \ddots & \ddots & 0 \\ 0 & 0 & 0 & -I & I \end{bmatrix}$$

Equation 13: Rate of movement matrix

MATLAB code:

```
Lambda = LambdaMat(N, nu);
```

C code:

```
float* Lambda = (float*)malloc(((N - 1) * column_b) * (N * column_b) * sizeof(float));
mpc_Lambda_matrix(Lambda, column_b, N);
```

Slack variables

The slack variable matrix $\bar{S_\psi} \in \Re^{(N \times nu) \times (N \times nu)}$ holds the scalar matrix $S_\psi \in \Re^{nu \times nu}$ and the slack variable vector $\bar{s_\psi} \in \Re^{N \times nu}$ holds the vector $s_\psi \in \Re^{nu}$. The slack variables increase the guarantee for the QP solver to find a feasible solution of U. The vector $s_\psi \in \Re$ shares the same value as the scalar matrix $S_\psi \in \Re^{nu \times nu}$.

Mathematics:

The slack variable matrix and vector can be seen in Equation 14 and Equation 15.

$$\bar{S_\psi} = \begin{bmatrix} S_\psi & 0 & \cdots & 0 \\ 0 & S_\psi & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & S_\psi \end{bmatrix}$$

Equation 14: Slack variable matrix

$$\bar{s_\psi} = \begin{bmatrix} s_\psi \\ s_\psi \\ \vdots \\ s_\psi \end{bmatrix}$$

Equation 15: Slack variable vector

MATLAB code:

```
barSpsi = barSpsiMat(Spsi, N, nu);
barspsi = barspsiVec(spsi, N, nu);
```

C code:

```
float* barSpsi = (float*)malloc((N * column_b) * (N * column_b) * sizeof(float));
mpc_barSpsi_matrix(barSpsi, Spsi_spsi, column_b, N);
mpc->barspsi = (float*)malloc(N * column_b * sizeof(float));
mpc_barspsi_vector(mpc->barspsi, Spsi_spsi, column_b, N);
```

QP solver matrix with slack variables

Here we create another QP solver matrix $\bar{H}_S \in \Re^{(2\times N\times nu)\times(2\times N\times nu)}$ that contains slack variables.

Mathematics:

The QP solver matrix with containing slack variables can be seen in Equation 16.

$$\bar{H}_S = \begin{bmatrix} H_S & 0 \\ 0 & \bar{S}_\psi \end{bmatrix}$$

Equation 16: QP solver matrix with slack variables

MATLAB code:

```
barH = barHMat(H, barSpsi, N, nu);
```

C code:

```
mpc->barH = (float*)malloc((2 * N * column_b) * (2 * N * column_b) * sizeof(float));
mpc_barH_matrix(mpc->barH, H, barSpsi, column_b, N);
```

Inequality constraints

The reason why MPC is used, it's because its constraints. Enabling the limitation to the control input signal U. This could also be done with signal saturation, with loss of stability guarantees. Therefore inequality constraints $AA \in \Re^{((N-1)\times nu+2\times N\times nz)\times(2\times N\times nu)}$ can be used to keep the stability guarantees.

Mathematics:

The inequality constraints matrix can be seen in Equation 17.

$$AA = \begin{bmatrix} \Lambda & 0 \\ \Gamma & -I \\ \Gamma & I \end{bmatrix}$$

Equation 17: Inequality constraints

MATLAB code:

```
AA = AAMat(Lambda, Gamma, N, nu, nz);
```

C code:

```
mpc->AA = (float*)malloc(((N - 1) * column_b + 2 * N * row_c) * (2 * N * column_b) *
sizeof(float));
mpc_AA_matrix(mpc->AA, Lambda, Gamma, row_c, column_b, N);
```

Holder arrays

It’s important in C programming language to declare initial arrays for the optimization function. They are only used for store data such as the state vector, integral value and user settings such as constraints.

Mathematics:

No mathematics provided here.

MATLAB code:

No MATLAB code provided here.

C code:

```
mpc->eta = (float*)malloc(row_c * sizeof(float));
memset(mpc->eta, 0, row_c * sizeof(float));
mpc->x = (float*)malloc(row_a * sizeof(float));
memset(mpc->x, 0, row_a * sizeof(float));
mpc->deltaumin = (float*)malloc(column_b * sizeof(float));
mpc->deltaumax = (float*)malloc(column_b * sizeof(float));
mpc->umin = (float*)malloc(column_b * sizeof(float));
mpc->umax = (float*)malloc(column_b * sizeof(float));
mpc->is_initlized = true;
```

mpc_set_constraints

Input constraints

After initialization of the matrices for the MPC are done. Then the need of setting the values for the constraints are mandatory. The constraints that are going to be set are the minimum input signal limit $u_{min} \in \Re^{nu}$, the maximum input signal limit $u_{max} \in \Re^{nu}$, the minimum output signal limit $Z_{min} \in \Re^{N \times nz}$, the maximum output signal limit $Z_{max} \in \Re^{N \times nz}$, the minimum rate of change input limit $\Delta U_{min} \in \Re^{(N-1) \times nu}$ and the maximum rate of change input limit $\Delta U_{max} \in \Re^{(N-1) \times nu}$.

Mathematics:

$$Z_{min} = \begin{bmatrix} z_{min} \\ z_{min} \\ \vdots \\ z_{min} \end{bmatrix}$$

$$Z_{max} = \begin{bmatrix} z_{max} \\ z_{max} \\ \vdots \\ z_{min} \end{bmatrix}$$

$$\Delta U_{min} = \begin{bmatrix} \Delta u_{min} \\ \Delta u_{min} \\ \vdots \\ \Delta u_{min} \end{bmatrix}$$

$$\Delta u_{max} = \begin{bmatrix} \Delta u_{max} \\ \Delta u_{max} \\ \vdots \\ \Delta u_{max} \end{bmatrix}$$

Equation 18:
Constraints

MATLAB code:

No MATLAB code provided here.

C code:

```
mpc_set_input_constraints(mpc, umin, umax);
mpc_set_output_constraints(mpc, zmin, zmax);
mpc_set_input_change_constraints(mpc, deltaumin, deltaumax);
```

mpc_optimize

Integral action

This MPC algorithm has integral action included. The integral action is an addition to the reference vector. The integral is summation of the error of the trajectory.

Mathematics:

$$\Psi = r - y$$

$$\eta = \eta + \alpha \times \Psi$$

$$\eta = \begin{Bmatrix} antiwindup \times \sigma(\eta), if\ |\eta| > antiwindup \\ \eta \end{Bmatrix}$$

MATLAB code:

```
psi = r - y;
eta = eta + alpha*psi;
if(abs(eta) > antiwindup)
   eta = sign(eta)*antiwindup;
end
```

C code:

```
mpc_eta_vector(mpc->eta, r, y, mpc->alpha, integral_active, row_c);
mpc_antiwindup_vector(mpc->eta, mpc->antiwindup, row_c);
```

Reference vector

The reference vector $R \in \Re^{N \times nz}$ holds the vector $r \in \Re^{nz}$. The reference vector is the trajectory vector where the user want the output of the plant to follow.

Mathematics:

The reference vector can be seen in Equation 19.

$$R = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ \vdots \\ r_N \end{bmatrix}$$

Equation 19: Reference vector

MATLAB code:

```
R = RVec(r, N);
```

C code:

```
float* R = (float*)malloc(N * row_c * sizeof(float));
mpc_vector(R, r, row_c, N);
```

Disturbance vector

The disturbance vector $D \in \Re^{N \times nd}$ holds the vector $d \in \Re^{nd}$. The disturbance vector is the measurement vector which affects the output of the plant. In other words, the disturbance vector is a measurement signal from a sensor attached onto the plant.

Mathematics:

The reference vector can be seen in Equation 20.

$$D = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_N \end{bmatrix}$$

Equation 20: Reference vector

MATLAB code:

```
D = DVec(d, N);
```

C code:

```
float* D = (float*)malloc(N * column_e * sizeof(float));
mpc_vector(D, d, column_e, N);
```

QP gradient vector

The QP algorithm requires the gradient vector $g \in \Re^{N \times nu}$.

Mathematics:

The gradient vector can be seen in Equation 21.

$$g = M_{x_0} \times x\, M_R \times (R + \eta) + M_D \times D + M_{u_{m-1}} \times u_{-1}$$

Equation 21: Gradient vector

MATLAB code:

```
um1 = u;
Ri = repmat(eta, N, 1);
g = gVec(Mx0, x, MR, R + Ri, MD, D, Mum1, um1);
```

C code:

```
float* um1 = (float*)malloc(column_b * sizeof(float));
memcpy(um1, u, column_b * sizeof(float));
float* g = (float*)malloc(N * column_b * sizeof(float));
mpc_g_vector(g, mpc->Mx0, mpc->x, mpc->MR, R, mpc->eta, mpc->MD, D, mpc->Mum1, um1,
row_a, row_c, column_b, column_e, N);
```

Constraints on inputs

The constraints on input U are $U_{min} \in \Re^{N \times nu}$ and $U_{max} \in \Re^{N \times nu}$.

Mathematics:

The constraints on input U can be seen in Equation 22.

$$U_{min} = \begin{bmatrix} max(u_{min}, \Delta u_{min} + u_{-1}) \\ u_{min} \\ u_{min} \\ \vdots \\ u_{min} \end{bmatrix}, U_{max} = \begin{bmatrix} min(u_{max}, \Delta u_{max} + u_{-1}) \\ u_{max} \\ u_{max} \\ \vdots \\ u_{max} \end{bmatrix}$$

Equation 22: Constraints on input U

MATLAB code:

```
Umin = UminVec(umin, deltaumin, um1, N, nu);
Umax = UmaxVec(umax, deltaumax, um1, N, nu);
```

C code:

```
float* Umin = (float*)malloc(N * column_b * sizeof(float));
float* Umax = (float*)malloc(N * column_b * sizeof(float));
mpc_Umin_vector(Umin, mpc->umin, mpc->deltaumin, um1, N, column_b);
mpc_Umax_vector(Umax, mpc->umax, mpc->deltaumax, um1, N, column_b);
```

Constraints on outputs

The constraints on output Z are $\bar{Z}_{min} \in \Re^{N \times nz}$ and $\bar{Z}_{max} \in \Re^{N \times nz}$.

Mathematics:

The constraints on output Z can be seen in Equation 23.

$$\bar{Z}_{min} = Z_{min} + \Phi \times x_0 + \Gamma_d \times D$$
$$\bar{Z}_{max} = Z_{min} - \Phi \times x_0 - \Gamma_d \times D$$

Equation 23: Constraints on output Z

MATLAB code:

```
barZmin = barZminVec(Zmin, Phi, x, Gammad, D);
barZmax = barZmaxVec(Zmax, Phi, x, Gammad, D);
```

C code:

```
float* barZmin = (float*)malloc(N * row_c * sizeof(float));
float* barZmax = (float*)malloc(N * row_c * sizeof(float));
mpc_barZmin_vector(barZmin, mpc->Zmin, mpc->Phi, mpc->x, mpc->Gammad, D, row_a, row_c,
column_b, column_e, N);
mpc_barZmax_vector(barZmax, mpc->Zmax, mpc->Phi, mpc->x, mpc->Gammad, D, row_a,
row_c, column_b, column_e, N);
```

Gradient vector with soft output constraints

The gradient vector g with soft output constraints is called $\bar{g} \in \Re^{N \times nu + N \times nu}$.

Mathematics:

The gradient vector $\bar{g}$ can be seen in Equation 24.

$$\bar{g} = \begin{bmatrix} g \\ \bar{s}_{\psi} \end{bmatrix}$$

Equation 24: The gradient vector $\bar{g}$

MATLAB code:

```
barg = bargVec(g, barspsi);
```

C code:

```
float* barg = (float*)malloc((N * column_b + column_b * N) * sizeof(float));
mpc_barg_vector(barg, g, mpc->barspsi, column_b, N);
```

Modified constraints on input

The modified constraints on input U is called $\bar{U}_{min} \in \Re^{N \times nu+N}$ and $\bar{U}_{max} \in \Re^{N \times nu+N}$.

Mathematics:

The modified constraints $\bar{U}_{min} \in \Re^{N \times nu+N}$ and $\bar{U}_{max} \in \Re^{N \times nu+N}$ can be seen in Equation 25.

$$\bar{U}_{min} = \begin{bmatrix} U_{min} \\ 0 \end{bmatrix}$$
$$\bar{U}_{max} = \begin{bmatrix} U_{max} \\ \infty \end{bmatrix}$$

Equation 25: The modified constraints on inputs

MATLAB code:

```
barUmin = barUminVec(Umin, N);
barUmax = barUmaxVec(Umax, infinity, N);
UI = eye(N * nu + N, 2 * N * nu);
```

C code:

```
float* barUmin = (float*)malloc((N * column_b + N) * sizeof(float));
float* barUmax = (float*)malloc((N * column_b + N) * sizeof(float));
mpc_barUmin_vector(barUmin, Umin, column_b, N);
mpc_barUmax_vector(barUmax, Umax, column_b, N);
```

Quadratic programming

The quadratic programming is applied with all the inequality constraints: $b_{min} \in \Re^{(N-1)\times nu+N\times nz+N\times nz}$, $b_{max} \in \Re^{(N-1)\times nu+N\times nz+N\times nz}$, $a_{qp} \in \Re^{(2\times((N-1)\times nu+2\times N\times nz)+2\times(N\times nu+N))\times(2\times N\times nu)}$ and $b_{qp} \in \Re^{2\times(N\times nu+N)+2\times((N-1)\times nu+N\times nz+N\times nz)}$.

Mathematics:

The inequality constraints constraints of the quadratic programming can be seen in Equation 26.

$$b_{min} = \begin{bmatrix} \Delta U_{min} \\ -\infty \\ \bar{Z}_{min} \end{bmatrix}, b_{max} = \begin{bmatrix} \Delta U_{max} \\ \bar{Z}_{max} \\ \infty \end{bmatrix}, a_{qp} = \begin{bmatrix} UI \\ AA \\ -UI \\ -AA \end{bmatrix}, b_{qp} = \begin{bmatrix} \bar{U}_{max} \\ \bar{b}_{max} \\ -\bar{U}_{min} \\ b_{min} \end{bmatrix}$$

Equation 26: The inequality constraints for the QP

The objective function can be see in Equation 27.

$$\underbrace{min}_{\bar{U}} : \Phi = \frac{1}{2} \times \bar{U}^T \times \bar{H} \times \bar{U} + \bar{g}^T \times \bar{U}$$
$$s.t : b_{min} \le \bar{U} \le b_{max}$$
$$s.t : a_{qp} \times \bar{U} \le b_{qp}$$

Equation 27: The objective function of QP

MATLAB code:

```
bmin = bminVec(deltaUmin, barZmin, infinity, N, nz);
bmax = bmaxVec(deltaUmax, barZmax, infinity, N, nz);
aqp = [UI; AA; -UI; -AA];
bqp = [barUmax; bmax; -barUmin; -bmin];
```

C code:

```
float* bmin = (float*)malloc(((N - 1) * column_b + N * row_c + N * row_c) * sizeof(float));
float* bmax = (float*)malloc(((N - 1) * column_b + N * row_c + N * row_c) * sizeof(float));
mpc_bmax_vector(bmax, mpc->deltaUmax, barZmax, column_b, row_c, N);
mpc_bmin_vector(bmin, mpc->deltaUmin, barZmin, column_b, row_c, N);
float* aqp = (float*)malloc((2 * ((N - 1) * column_b + 2 * N * row_c) + 2 * (N * column_b + N)) *
(2 * N * column_b) * sizeof(float));
float* bqp = (float*)malloc((2 * (N * column_b + N) + 2 * ((N - 1) * column_b + N * row_c + N *
row_c)) * sizeof(float));
mpc_aqp_matrix(aqp, mpc->AA, column_b, row_c, N);
mpc_bqp_vector(bqp, barUmin, barUmax, bmin, bmax, column_b, row_c, N);
float* U = (float*)malloc(2 * N * column_b * sizeof(float));
const STATUS_CODES status = quadprogslim(mpc->barH, barg, aqp, bqp, NULL, NULL, U, 2 *
((N - 1) * column_b + 2 * N * row_c) + 2 * (N * column_b + N), 0, 2 * N * column_b, false);
memcpy(u, U, column_b * sizeof(float));
```