Appendix A
FE Modeling of the Active Structure

This appendix complements the discussion of FEM simulation results in Sect. 5.4 with a detailed account of the code used to generate those results. Thus, it is recommended to readers who are aiming to create a FEM model of a smart system equipped with piezoelectric actuators in ANSYS. ANSYS is one of the well-known commercial finite element analysis software packages available today that is capable of performing simulations including the electro-mechanical interaction from piezoelectric sensors and actuators. ANSYS has been used in numerous works aimed at investigating the vibration and damping behavior of smart structures equipped with piezoelectric actuators [3, 4, 5, 15].

The basis for the code featured here are the demonstration examples found in the ANSYS manual [1] and using the material data for the PZT actuator and the supporting aluminum beam from various sources [6, 9, 10, 12, 13, 14]. For more information about the simulation of vibrating systems in ANSYS the reader may be interested in the book by Hatch [5] or other similar publications on the topic of finite element analysis [2, 7, 8, 11, 16]. This appendix is divided into several sections discussing analysis setup, parameter declaration, creating a solid model, meshing the model, creating boundary conditions and setting up various simulation types.

The cantilever beam of the active vibration control (AVC) demonstrator—featured throughout this work as a comparison benchmark—has been modeled using the 3D structural solid element SOLID5. The higher order tetrahedral version of this element is SOLID98, although the usage of this is not justified in this application. The settings of the element properties allow for a simple mechanical use, where only the displacement of the nodes is considered.

SOLID5 is defined by eight nodes, each having three degrees of freedom. These DOF are translations in the nodal $x$, $y$ and $z$ directions [1]. Material properties required for this element in this case is Young’s modulus, Poisson’s ratio and density. The material properties for aluminum were obtained from engineering tables and by direct experimentation. All units used were SI default, since ANSYS utilizes the constant value for the permittivity of free space in F/m.
The piezoelectric material was modeled using a 3D coupled field solid element SOLID 5. This element was chosen because of its piezo capability. A similar element SOLID45 has also been featured in studies aimed at vibration control, for example the work by Dong et al. in [4]. This element has eight nodes with up to six degrees of freedom. This application utilizes three axial displacements and one voltage DOF. Polarization direction is assumed to be along the $z$ axis of the element coordinate system. Required material properties to sufficiently describe the material properties of the crystal are: density, permittivity at constant stress, elastic compliance matrix and piezoelectric strain matrix. Material properties are anisotropic, meaning that it has different stress/strain behavior in $x$, $y$ and $z$ directions. The axes of anisotropy coincide with the axes of the element.

The assembly of the material property matrices proved to be difficult because of the misleading indexing conventions used in practice. Material properties for the piezoelectric material PZT5A were obtained from the manufacturer and diverse other sources [6, 12, 13]. Some elements of the matrices were numerically calculated from other known parameters, as often not all constants are readily available.

Physical dimensions and the placement of PZT patches respect the actual configuration of the experimental setup introduced in Sect. 5.1. Because the geometric design of the beam and actuator placement was initially subject to a frequent redesigning procedure due to the iterative workflow, the geometric model is working with a millimeter precision. Piezoelectric patches and the aluminum cantilever beam were meshed separately. Meshing had a uniform 2 mm density. This ensured sufficient resolution and precision without sacrificing too much computing time.

The overlapping surfaces provide coincident nodes. These nodes were merged with the relevant command, to ensure mesh connectivity. Epoxy resin used on the physical model was assumed to have an effect on the overall dynamic response and it has been explicitly modeled at first, however this assumption was later removed. Instead an ideally thin and rigid adhesive layer was assumed in more recent simulations.

### A.1 Analysis Setup

Although ANSYS presents a possibility to use a graphical user interface to create FEM models and perform simulations; writing command macros has several advantages. These include the possibility to correct mistakes without tediously repeating all preceding tasks and easy reconfiguration. This work utilized the built-in ANSYS macro language ANSYS Parametric Design Language (APDL) to create models and initiate simulations.

The following sections describe the process of creating a finite element model of the considered active structure. These steps include simulation environment initiation, parameter definitions, geometric solid modeling, meshing, establishing boundary and initial conditions and of course setting up simulations.
A.1.1 Initializing

Initializing of the modeling and simulation process starts by exiting from the processor and clearing ANSYS database. The title of the job is stated, while successive commands suppress extended interpreted data input and place comments in the output. To begin the modeling phase, the pre-processor is engaged. To prevent ANSYS from displaying irrelevant warning messages, shape checking is carried out in silent mode. Shape checking limit is also increased from the default value:

```plaintext
FINISH
/CLEAR
/title
/nopr
/com
/PREP7
SHPP,SILENT,ON
SHPP,MODIFY,1,40
```

A.2 Defining Problem Variables and Parameters

Problem parameters and other variables are defined in the following code segments. These variables mainly describe the geometrical properties of transducers, support beam, adhesive layer and transducer placement. Material properties are also stated:

A.2.1 Dimensions of the Piezoelectric Transducer and its Material

Dimensions of the piezoelectric wafer are stated, along with the electric potential if needed in the actual simulation. Material properties of the piezoelectric patch are also declared. This is including density of PZT5A, piezoelectric strain coefficients\(^1\) and relative permittivity at constant stress:

\[
\begin{align*}
L &= 44e^{-3} \\
H &= 0.25e^{-3} \\
W &= 20e^{-3} \\
V &= 120 \\
\end{align*}
\]

---

\(^1\) According to manufacturer data and available PZT5A material properties.
\begin{verbatim}
s11=16.4e-12
s22=s11
s12=-5.74e-12
s13=-7.22e-12
s23=s13
s33=18.8e-12
s44=47.5e-12
s55=47.5e-12
s66=44.3e-12
pidns=7700
d31=-179e-12
d32=-179e-12
d33=350e-12
d24=584e-12
d16=584e-12
ept11=1730
ept22=1730
ept33=1700
\end{verbatim}

\subsection*{A.2.2 Support Beam Dimensions and Material}

Length, thickness and width of the support beam are declared with millimeter precision. The material properties for aluminum are stated, including Young’s modulus,\footnote{For generic aluminum this is 70 GPa, Dural has a 73 GPa modulus, while the material in question has been approximated at a softer 66 GPa modulus.} Poisson’s ratio\footnote{From various sources: Poisson’s ratio is 0.35 for aluminum.} and density:

\begin{verbatim}
Lb=550e-3
Hb=3e-3
Wb=40e-3
EXY=66.70e + 09
POIS=0.35
aldns=2834
\end{verbatim}

\subsection*{A.2.3 Adhesive Geometry and Material Properties}

Geometrical properties of the adhesive layer are stated, including length, width and bonding thickness. The offset from the piezoelectric patch sides is also declared, this is the dimension the adhesive exceeds wafer geometry.


Mechanical properties of the adhesive layer are stated. Its Young’s modulus, Poisson’s ratio and density are conforming to generic epoxy resin properties:

\[ \begin{align*} 
  H_a &= 0.05 \times 10^{-3} \\
  L_a &= L \\
  W_a &= W \\
  \text{tmp} &= 0 \times 10^{-3} \\
  L_{da1} &= \text{tmp} \\
  W_{da1} &= \text{tmp} \\
  L_{da2} &= \text{tmp} \\
  W_{da2} &= \text{tmp} \\
  L_{da3} &= \text{tmp} \\
  W_{da3} &= \text{tmp} \\
  L_{da4} &= \text{tmp} \\
  W_{da4} &= \text{tmp} \\
  E_{XYa} &= 4 \times 10^9 \\
  \text{POISa} &= 0.38 \\
  \text{dnsa} &= 1160 
\end{align*} \]

### A.2.4 Placement of the Piezoelectric Transducers on the Beam

The following code segment states parameters defining the geometric placement of piezoelectric transducers on the surface of the aluminum support beam. All units are in meters and referenced from the default coordinate system origin:

\[ \begin{align*} 
  L_1 &= 14 \times 10^{-3} \\
  W_1 &= 10 \times 10^{-3} \\
  L_2 &= 90 \times 10^{-3} \\
  W_2 &= 10 \times 10^{-3} \\
  L_3 &= 322 \times 10^{-3} \\
  W_3 &= 10 \times 10^{-3} \\
  L_4 &= 14 \times 10^{-3} \\
  W_4 &= 10 \times 10^{-3} 
\end{align*} \]

### A.3 Solid Model

The following code segments create a solid volumetric model of the active structure. Due to the simple shape of the device, practically only block shapes are needed with the relevant dimensions. Base beam has to be divided into sections for

---

4 See [14] for details on epoxy resin tensile strength and other mechanical properties.

5 This is the clamped end of the blade.
practical meshing considerations, while the modeling of transducers and adhesives is straightforward.

A.3.1 Beam Model

The first command, \texttt{BLC4} creates a block with the dimensions of the support blade. Later the beam is divided into sections according to the distribution of the piezoelectric patches. This is needed for the meshing phase of the work.

Division of the original block is carried out by rotating the work plane with the \texttt{WROTA} command and offsetting it to the desired place with the \texttt{WPOFFS} command. Finally, \texttt{VSBB} subtracts intersection of the working plane from volumes, effectively dividing volumes.

The very last \texttt{WPCSYS} command in this code segment resets the working plane into its original location by defining its position based on the working coordinate system:

\begin{verbatim}
BLC4, 0, 0, Lb, Wb, Hb
WPROTA,, -90,
WPOFFS,, W1
VSBB, ALL
WPOFFS,, W
VSBB, ALL
WAVE, 0, 0, 0,
WPROTA,, 90
WPOFFS,, L1
VSBB, ALL
WPOFFS,, L
VSBB, ALL
WPOFFS,, L2-L1-L
VSBB, ALL
WPOFFS,, L
VSBB, ALL
WPOFFS,, L3-L2-L
VSBB, ALL
WPOFFS,, L
VSBB, ALL
WPCSYS,, 0
\end{verbatim}

A.3.2 Solid Model of the Transducers

Piezoelectric transducer geometry is modeled by defining blocks at each respective location. The three upper and finally the bottom transducers are created using the
previously defined dimension parameters. All dimensions relate to the particular command instruction and coordinate reference system:

\[
\text{BLOCK, } L_1, L_1 + L, w_1, w_1 + W, H_b + H_a, H_b + H + H_a
\]

\[
\text{BLOCK, } L_2, L_2 + L, w_2, w_2 + W, H_b + H_a, H_b + H + H_a
\]

\[
\text{BLOCK, } L_3, L_3 + L, w_3, w_3 + W, H_b + H_a, H_b + H + H_a
\]

\[
\text{BLOCK, } L_4, L_4 + L, w_4, w_4 + W, 0 - (H_a), - (H + H_a)
\]

**A.3.3 Solid Model of the Adhesive Layer**

The geometry of the adhesive epoxy resin layer is modeled similar to the piezoelectric transducers. Adhesive layer is modeled by defining blocks at each respective location. All dimensions relate to the particular command instruction and coordinate reference system:

\[
\text{BLOCK, } L_1 - L_{da1}, L_1 + L_{da1} + L_a, W_1 - W_{da1}, W_1 + W_{da1} + W_a, H_b, H_b + H_a
\]

\[
\text{BLOCK, } L_2 - L_{da2}, L_2 + L_{da2} + L_a, W_2 - W_{da2}, W_2 + W_{da2} + W_a, H_b, H_b + H_a
\]

\[
\text{BLOCK, } L_3 - L_{da3}, L_3 + L_{da3} + L_a, W_3 - W_{da3}, W_3 + W_{da3} + W_a, H_b, H_b + H_a
\]

\[
\text{BLOCK, } L_4 - L_{da4}, L_4 + L_{da4} + L_a, W_4 - W_{da4}, W_4 + W_{da4} + W_a, 0, - (H_a)
\]

**A.4 Assigning Material and Element Types**

Materials for the piezoelectric transducer, aluminum beam and epoxy resin adhesive layer are created by defining a finite element type for each and assigning material properties. All three materials use the same element *SOLID5* with piezoelectric capability, although only the transducers have this feature enabled. The *ET* command inputs element type with the proper switch controlling the presence of an additional electric potential degree of freedom.

The command *MP* inputs material properties like density, Poisson’s ratio and permittivity at constant stress or Young’s modulus if applicable. Command *TB* activates a data table for nonlinear material properties or special element input. Fields in the command control the particular type of data input: *ANEL* assembles an elastic compliance matrix, *PIEZ* a piezoelectric strain matrix:

\[
et, 1, \text{SOLID5}, 3
\]

\[
\text{MP, DENS}, 1, \text{pidns}
\]

\[
\text{MP, PERX}, 1, \text{ept11}
\]

\[
\text{MP, PERY}, 1, \text{ept22}
\]

\[
\text{MP, PERZ}, 1, \text{ept33}
\]

\[
\text{tb, ANEL}, 1, , , 1
\]

\[
\text{tbda}, 1, s_{11}, s_{12}, s_{13}
\]
A.5 Meshing of the Solid Model

The following code segment assigns materials and elements to geometric shapes, and creates a finite element meshed model. First, a uniform finite element size is declared by the command \textit{ESIZE}. Although this is not the only option, successive volumes defining beam, transducers and adhesives are selected based on their geometric location using the \textit{VSEL} command with the \textit{LOC} option. Material properties and elements are assigned to geometry using the \textit{VATT} command:

\begin{verbatim}
  ESIZE, 4e-3
  VSEL, S, LOC, Z, 0, Hb
  VATT, 2, 1, 2, 0
  VSEL, S, LOC, Z, 0, -Ha
  VSEL, A, LOC, Z, Hb, Hb + Ha
  VATT, 3, 1, 3, 0
  VSEL, S, LOC, Z, -Ha, -Ha-H
  VSEL, A, LOC, Z, Hb + Ha, Hb + Ha + H
  VATT, 1, 1, 1, 0
\end{verbatim}

All shapes are reselected with the \textit{ALLSEL} command, and replotted to the screen using \textit{VPLOT}. For the SOLID5 element that supports multiple shapes, \textit{MSHAPE} specifies the 3D quadrilateral volume element shape to be used for meshing. Command \textit{VMESH} initiates the meshing of all volumes:
Finally, all nodes are selected after meshing. This is followed by the very important merging command, using `NUMMRG` with the node option and given precision. Neighboring nodes are merged into one. This actually ensures that the beam, adhesive and transducers are glued together in the simulation and act as one physical entity, thus mimicking a perfectly rigid bond:

```
nsel, all
nummrg, node, 1e-5
```

### A.6 Defining Boundary Conditions

The following code segments define the boundary conditions of the problem. Gravitational field or the clamp at the fixed end of the beam are present and identical to the case of each simulation type. Depending on whether one performs a modal analysis or a harmonic one, boundary conditions on the piezoelectric patches may be different.

The vector of gravitational pull is defined perpendicular to the direction of vibrations, just as in the case of the real system. The gravitational field is defined with the `ACEL` command. The left end of the blade identical with the origin of the coordinate reference system is clamped. First, the nodes at that location are selected by the `NSEL` command, then zero displacement degree of freedom constraints are engaged using the `D` command. Finally, all the nodes are reselected:

```
acel, 0, -9.81, 0
nsel, s, loc, x, 0
d, all, ux, 0, , , , uy, uz
nsel, all
```

Electrodes are defined on the piezoelectric patches by a similar methodology. First, nodes in the uppermost layer of transducer mesh are selected. Areas are picked by the `ASEL` command and then nodes in these areas selected by `NSLA`. The lowest index number node is retrieved and stored in a variable. Command `CP` defines the voltage coupled degree of freedom at these nodes, and the index node is supplied with a zero potential. This process is then repeated for all transducers:

```
asel, s, loc, z, Hb+Ha+H
asel, r, loc, x, L1, L1+L
nsla, s, 1
```
*get, p1, node, 0, num, min
cp, 1, volt, all
d, p1, VOLT, V, 0
allsel
asel, s, loc, z, Hb+Ha+H
asel, r, loc, x, L2, L2+L
nsla, s, 1
*get, p2, node, 0, num, min
!cp, 2, volt, all
d, p2, VOLT, 0, 0
allsel
asel, s, loc, z, Hb+Ha+H
asel, r, loc, x, L3, L3+L
nsla, s, 1
*get, p3, node, 0, num, min
!cp, 3, volt, all
d, p3, VOLT, 0, 0
allsel
asel, s, loc, z, -Ha-H
asel, r, loc, x, L1, L1+L
nsla, s, 1
*get, p4, node, 0, num, min
!cp, 4, volt, all
d, p4, VOLT, 0, 0
allsel

Electrodes at the lower, bonded side are defined in a similar manner. In this case, the voltage potential is set to zero, preparing the model for a closed-circuit modal analysis. A closed-circuit modal analysis presumes that the electrode terminals are shorted, as opposed to the open state:

asel, s, loc, z, Hb+Ha
asel, r, loc, x, L1, L1+L
nsla, s, 1
d, all, volt, 0, 0
nset, all
asel, s, loc, z, Hb+Ha
asel, r, loc, x, L2, L2+L
nsla, s, 1
d, all, volt, 0, 0
nset, all
asel, s, loc, z, Hb+Ha
asel, r, loc, x, L3, L3+L
nsla, s, 1
d, all, volt, 0, 0
nset, all
A.7 Setting Up Simulations

Once a properly parameterized finite element model is created by establishing geometry, meshing and defining boundary conditions, one is presented with a plethora of analysis possibilities. Here the setup and initiation of a modal and harmonic analysis will be presented. In addition to that, the model may be quickly altered to perform transient analyses with various initial conditions or for example a static deflection test.

A.7.1 Modal Analysis Setup

Setup and initiation of a modal analysis is very simple, given a well-configured model. First, the analysis type is determined to be modal, by the \texttt{ANTYP} command. Next, options of the modal analysis are set by the \texttt{MODOPT} command. Here the block Lanczos mode extraction method is used to retrieve the first ten modes, in the 0 to 500 Hz bandwidth. All ten modes are expanded by specifying the \texttt{MXPAND} command. Solution is started with the \texttt{SOLVE} command. Finally, the pre-solution program stage is left, one may proceed to post-processing:

\begin{verbatim}
ANTYP, MODAL
MODOPT, LANB, 10, 0, 500
MXPAND, 10
SOLVE
FINISH
\end{verbatim}

A.7.2 Harmonic Analysis Setup

Setting up a harmonic analysis is not a particularly challenging task, given a properly configured FEM model. One has to consider the desired resolution of the outputs, since an over-meshed model and very exact required resolution may
necessitate lengthy computational times. A harmonic analysis with about a thousand frequency points on this relatively simple example takes several hours to complete on average hardware.

Solution setup begins with expanding the default number of result sets to a higher number. Analysis type is specified using the \texttt{ANTYPE} command, and invoking the harmonic option. Command \texttt{HROUT} determines whether ANSYS produces real and imaginary or phase angles and degrees in the output. Solution data written to the database is controlled by \texttt{OUTRES}, which in this case is set to store everything. Rayleigh damping factors are input using \texttt{ALPHAD} and \texttt{BETAD} commands:

\begin{verbatim}
/ CONFIG, NRES, 2500
ANTYPE, HARMIC
HROUT, OFF
OUTRES, ALL, ALL
ALPHAD, 0
BETAD, 0.001
\end{verbatim}

The bandwidth of interest may be divided into subdomains with differing resolution of detail. In this particular example, only two portions are considered. Frequency range is input by the \texttt{HARFRQ} command, while the number of frequency substeps to be evaluated is determined by the \texttt{NSUBST} command. \texttt{KBC} specifies whether the load should be ramped or stepped, in this case it is set to be stepped. The given partial solution is defined to an indexed solution step by \texttt{LSWRITE}:

\begin{verbatim}
HARFRQ, 0, 70
NSUBST, 500,
KBC, 1
LSWRITE, 1
HARFRQ, 70, 500
NSUBST, 500,
KBC, 1
LSWRITE, 2
\end{verbatim}

The solution is finally started by the \texttt{LSSOLVE} command, listing the indices of starting solution step and finishing solution step. Finally, the pre-solution program stage is left and one may proceed to post-processing:

\begin{verbatim}
LSSOLVE, 1, 2, 1
FINISH
\end{verbatim}
Appendix B

MPC Code Implementation Details

Appendix B of this book is an important extension of Chap. 10 and it is intended for the reader who is interested in the implementation details of MPC algorithms for vibration control or other applications. This appendix mainly contains code segments from the laboratory implementation of various MPC strategies for the experimental AVC demonstrator considered in this work. After generalization, the code featured here can be utilized for the feedback control of other dynamic systems.

This appendix may be divided into three different sections, each devoted to a certain type of MPC strategy. The first section concentrates on the offline computation of prediction matrices and formulation of constraints for the QPMPC controller. In addition to this, it also features a short discussion on simulation. As the online controller is solved using qpOASES, its parsing and internal structure is not discussed here. Each quadratic programming solver has a different input syntax, one may apply the controller to the solver according to the general rules of MPC or use the brief discussion presented in Sect. 10.1. Section B.2 features the code segments necessary both to compute an MPMPC controller offline through the MPT-Toolbox and the real-time application of the MPMPC strategy. Finally, Sect. B.3 gives a detailed account of the off and online NRMPC code with an analysis of the possible SDP solvers.

B.1 QPMPC

This section is devoted to the practical implementation of a quadratic programming-based MPC algorithm, providing constraint handling and a priori stability guarantees. The material complements the discussion in Sect. 10.1 and is based on the theoretical knowledge introduced by Chaps. 6 and 7. The theoretical basis for the formulation of the dual-mode infinite horizon quadratic programming-based MPC algorithm stabilized by a constraint checking horizon is common
and can be found in many popular works written on MPC, such as [17, 25, 34, 35, 51, 54].

The online portion of the code is parsed to the qpOASES off-the-shelf quadratic programming solver [29, 32] as discussed in Sect. 10.1. The particular way of parsing the problem to the solver always depends on the solver choice, therefore this chapter concentrates only on the problem setup such as the prediction matrices and constraint formulation. In addition to this, pointers are given to those who wish to use the algorithm in an offline simulation.

B.1.1 Setup

Let us begin with the setup of the predictive control problem. For this, the code shall be implemented in the Matlab m-script language [58, 60]. First, it is required to specify certain parameters for the QP MPC controller. Some of these parameters are: the sampling period $T_s = T_s$, if it is an offline simulation a stop time is required as well, and it is also essential to state the prediction horizon $n_c = n_c$. In case symmetric bounds on the input are assumed, the constraints are set using $\bar{u} = \underline{u} = u_{\text{max}}$. In the case of PZT actuation, the system constraint is the polarization voltage of the piezoelectric transducers

$$T_s = 0.01; \quad T = 0.5 \quad n_c = 70; \quad \text{run} = T / T_s; \quad u_{\text{max}} = 120;$$

where run is the runtime in case an offline simulation is needed.

In the next step, it is required to load and specify a prediction model and possibly an initial state for Kalman filtering or simulation purposes. In this example, the model is loaded from a saved system identification file:

```matlab
load n4s2A;
A = n4s2.A;
B = n4s2.B;
C = n4s2.C;
nx = length(A);
X1(:,1) = zeros(1,nx);
```

Penalization for the inputs and states needs to be stated as well. The input penalization can be determined by direct experimentation with the algorithm, or simply evaluating different linear-quadratic controllers in simulation and determining a good balance between controller performance and aggressiveness. In this case, the input penalty has been found using an LQ controller with the

---

6 Like those featured in Chap. 11.
settings $\mathbf{R} = \mathbf{r} = \text{R}=1\text{E}-4$ and $\mathbf{Q} = \mathbf{Q} = \mathbf{C}' \ast \mathbf{C}$, while balancing the input somewhat above the constraints.

\[ \mathbf{R} = 1\text{E}-4; \]
\[ \mathbf{Q} = \mathbf{C}' \ast \mathbf{C}; \]

### B.1.2 Prediction Matrices

To simplify the program structure it is possible to pass the information to a stand-alone function. This custom function uses the system model, penalty matrices, the constraints, a horizon and possibly system order information. Its outputs are the properly formulated prediction matrices and possibly the re-formulated constraints, which have been defined in Sects. 6.2, 6.3 and 6.7. The function creating the MPC structure including can be formulated as follows:

\[ [\mathbf{H}, \mathbf{F}, \mathbf{G}, \mathbf{A}_c, \mathbf{b}_0, \mathbf{B}_x, \mathbf{K}_i, \mathbf{N}_c] = \text{predmodelqp}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{R}, \mathbf{Q}, \mathbf{u}_\text{max}, \mathbf{n}_c, \mathbf{n}_x); \]

Now let us begin with examining what such a function may do, in order to generate the prediction matrices and cost function prediction matrices for the online run. As in most vibration damping applications, this implementation assumes a symmetric constraint on the input:

\[ \mathbf{u}_l = -\mathbf{u}_h; \]

This is followed by calculating the unconstrained linear-quadratic (LQ) optimal gain, along with the terminal weighting matrix:

\[ [\mathbf{K}, \mathbf{S}, \mathbf{e}] = \text{dlqr}(\mathbf{A}, \mathbf{B}, \mathbf{Q}, \mathbf{R}); \]
\[ \mathbf{K} = -\mathbf{K}; \]
\[ \mathbf{Q}_e = \text{dlyap}((\mathbf{A} + \mathbf{B} \ast \mathbf{K} )', (\mathbf{Q} + \mathbf{K}' \ast \mathbf{R} \ast \mathbf{K} )); \]

The forced and free state prediction matrices from 6.2 are calculated through a set of nested loops according to the following script:

\[ \mathbf{N} = \text{zeros}(\mathbf{n}_c \ast \mathbf{n}_x); \]
\[ \text{for } \mathbf{n} = 1 : \mathbf{n}_c; \]
\[ \text{M}(\mathbf{n} \ast \mathbf{n}_x - \mathbf{n}_x + 1 : \mathbf{n} \ast \mathbf{n}_x, :) = [(\mathbf{A} \ast \mathbf{n})]; \]
\[ \text{for } \mathbf{na} = 0 : \mathbf{n}_x : (\mathbf{n}_c \ast \mathbf{n}_x); \]
\[ \mathbf{N}(\mathbf{n}_x \ast (\mathbf{n} - 1) + (\mathbf{na} + 1) : \mathbf{n}_x \ast (\mathbf{n} - 1) + (\mathbf{na} + \mathbf{n}_x), \mathbf{n}) = [(\mathbf{A(\mathbf{na}/\mathbf{nx}) \ast \mathbf{B})]; \]
\[ \text{end}; \]
\[ \text{end}; \]
\[ \mathbf{N} = \mathbf{N}(1 : \mathbf{n}_x \ast \mathbf{n}_c, :) ; \]

where several other possible solutions may exist. These solutions can be equivalently good, and while their runtime may differ this should not be an issue in
an offline problem setup process. The last $n_x$ block row of the matrices $M = M$ and $N = N$ is also selected:

$$
M_1 = M(n_x*n_c-(n_x-1):n_x*n_c,:);
N_1 = N(n_x*n_c-(n_x-1):n_x*n_c,:);
$$

The next step is to create the cost prediction matrices as defined in Sect. 6.3. One has to begin with an initialization procedure:

$$
H_1=0;
F_1=0;
G_1=A^0*Q;
$$

This is then followed by creating the cost prediction matrices $H, F$ and $G$ first by running the following loop to get partial results:

```matlab
for i=0:nc-2
    H1t=N(1+i*nx:i*nx+nx,:)'*Q*N(1+i*nx:i*nx+nx,:);
    H1=H1+H1t;
    F1t=N(1+i*nx:i*nx+nx,:)'*Q*M(1+i*nx:i*nx+nx,:);
    F1=F1+F1t;
    G1t=M(1+i*nx:i*nx+nx,:)'*Q*M(1+i*nx:i*nx+nx,:);
    G1=G1+G1t;
end
```

and then finally assembling cost prediction matrices $H = G, F = F$ and $G = G$:

$$
H=H1+N(1+(n_c-1)*nx:(n_c-1)*nx+nx,:)'*Qe*N(1+(n_c-1)*nx:
(n_c-1)*nx+nx,:)+R*eye(n_c);
F=F1+N(1+(n_c-1)*nx:(n_c-1)*nx+nx,:)'*Qe*M(1+(n_c-1)
*nx:(n_c-1)*nx+nx,:);
G=G1+M(1+(n_c-1)*nx:(n_c-1)*nx+nx,:)'*Qe*M(1+(n_c-1)
*nx:(n_c-1)*nx+nx,:);
$$

To ensure feasibility and stability beyond the prediction horizon, the constraint checking horizon is calculated as well. This process has been described in Sect. 7.4 and in practice is started by an initialization procedure:

$$
K_i=K;
K_i(2,:)=K*(A+B*K);
i=1;
N_c=0;
u=uh+1;
$$

The length of the constraint checking horizon is then computed in the following loop:

7 See 7.4 for more details on the calculation of the constraint checking horizon.
while (u > uh);
    Ki(i+2,:) = K*(A+B*K)^(i+1);
    f=Ki(i+2,:);
    Am=[Ki(1:(i+1),:);-Ki(1:(i+1),:)]
    b=[uh*ones((i+1),1); -ul*ones((i+1),1)];
    x0=linprog(-f,Am,b);
    u=Ki(i+2,:)*x0;
    Nc=Nc+1;
    i=i+1;
end

B.1.3 Constraint Formulation

The routine which generates the prediction matrices also contains a simple algorithm, which is designed to define the constraints and reformulate them to be useful for direct quadratic programming solution. This formulation assumes symmetric input constraints:

\[ \text{Ac1} = \begin{bmatrix} \text{eye}(nc) \end{bmatrix} \; \text{for} \; i = 0 : Nc \]
\[ \text{b0} = \begin{bmatrix} \text{uh} \times \text{ones}(nc+Nc,1) \; -\text{ul} \times \text{ones}(nc+Nc,1) \end{bmatrix} \]
\[ \text{Bx1} = \text{zeros}((nc-1),nx) \]
\[ \text{Ac1}(nc+i,:) = [\text{Ki}(i+1,:) \times M2l] \]
\[ \text{Bx1}(nc+i,:) = [-\text{Ki}(i+1,:) \times M1l] \]
\[ \text{Ac} = [\text{Ac1}; -\text{Ac1}] \]
\[ \text{Bx} = [\text{Bx1}; -\text{Bx1}] \]

The prediction matrices need to be reformulated in a way that they can be used directly with the Simulink interface of qpOASES [31, 30]. First, the cost prediction matrix \( H \) is reformulated, so it is suitable to pass on directly to the qpOASES problem:

\[ H_{qp} = [\;] \]
\[ \text{for} \; i = 1 : nc \]
\[ H_{qpt} = H(i,:) \]
\[ H_{qp} = [H_{qp} \; H_{qpt}] \]
\[ \text{end} \]

---

8 See 6.7 for more details on constant formulation.
Passing on $F$ is possible with the original formulation. The cost matrix $A_c = A_c$ needs to be transformed likewise:

$$
A_{c Q P h} = A_c (1:nc+Nc,:) ;
A_{c Q P} = [] ;
$$

for $i=1:(nc+Nc)$
$$
A_{c Q P t} = A_{c Q P h} (i,:) ;
A_{c Q P} = [A_{c Q P} A_{c Q P t}] ;
$$
end

where the matrices need to be divided in the usual C programming style, along with constraint matrices $B_x = B_x$ and $b_0 = b_0$:

$$
B_{x Q P} = B_x (1:nc+Nc,:) ;
b_{0 Q P} = b_0 (1:nc+Nc,:) ;
$$

### B.1.4 Offline Simulation

An offline simulation is often necessary the verification of results or design purposes. In this case we may use the Matlab default quadratic programming solver, named *quadprog* [59]. To do this, one needs to launch a cycle with one iteration for each sampling instant and supply the optimization problem to the solver as follows:

for $k=1:run$;
$$
[U1(:,k), f, status1(1,k), output] = quadprog(H, F*(X1(:,k)) , A_c, b_0+B_x*X1(:,k), [], [], [], [], [], options) ;
X1(:,k+1) = A*X1(:,k) + B*(U1(1,k))' ;
Y1(k) = C*X1(:,k) ;
$$
end

where $X1$ is the matrix containing the states response results and $Y1$ is the vector containing the deflection response data. The last two lines assume that there is no need for state observation; this is to make the simulation simpler. In case the cost is needed as well, one may also include:

$$
J(k,:) = U1(:,k)'*H*U1(:,k) + 2*(X1(:,k))'*F'*U1(:,k) + \quad X1(:,k)'*G*X1(:,k) ;
$$

where the cost can be calculated using the original prediction matrices $H, F$ and $G$ or possibly by using the definition of the cost through the penalty matrices $Q, R$ and $P_f$. 


The optimization procedure can be fine-tuned by the `optimset` command:

```matlab
options = optimset('LargeScale','off','Display','off','TolFun',1e-12);
```

It is also possible to substitute the Matlab built-in `quadprog` function with the `qpOASES` Matlab interface [29, 32]. After compilation, the sequential `qpOASES` solver can be simply called by using the following code and the `qpOASES` command:

```matlab
for k=1:run; [objOA,U2(:,k),yd,status2(1,k),nWSRout2(1,k)] = qpOASES('i',H,F*X2(:,k),Ac,'',',',',',',',10);
X2(:,k+1)=A*X2(:,k)+B*U2(1,k);
Y2(k)=C*X2(:,k);
end
```

where `X2` and `Y2` is a new or alternate set of state and deflection matrices. In case the formulation is correct, the two simulation data sets shall be equal to each other up to small and negligible numerical errors.

In case one desires to solve a sequential quadratic programming problem, where the MPC problem is formulated for an adaptive system, or it is necessary to reconfigure the constraints during the online computation process, it is advised to use the sequential module `qpOASES_sequence` of `qpOASES` for maximum computational efficiency [31].

### B.2 MPMPC

This section of Appendix B lists the steps necessary to create an MPMPC controller via the MPT-Toolbox [40, 42, 43, 44]. The theoretical basis for multi-parametric programming has been briefly introduced in Sect. 8.2.1 while Sect. 10.2 showed the properties of the controllers as applied to the AVC demonstrator. One may find additional details on the formulation of multi-parametric controllers and the exact meaning of commands in the help of the MPT Toolbox or in the book by Kvasnica [40]. This section lists code segments that apply to both the offline and real-time code formulation. The discussion is augmented with the option to create a suboptimal but computationally efficient controller.
B.2.1 Offline Controller Computation

We begin with loading the system model to the workspace [41]:

```matlab
load n4s2A.mat
sysStruct.A = n4s2.A;
sysStruct.B = n4s2.B;
sysStruct.C = n4s2.C;
sysStruct.D = 0;
```

This initialization process is followed by naming the state variables and setting system constraints on inputs. Output constraints $y = -\dot{y}$ are set to infinity, therefore practically they are neglected:

```matlab
sysStruct.StateName{1} = 'x1';
sysStruct.StateName{2} = 'x2';
sysStruct.umin = -120;
sysStruct.umax = 120;
sysStruct.ymin = -inf;
sysStruct.ymax = inf;
```

Suboptimality level is zero; this means that the software generates an optimal controller with a quadratic cost function—the norm is set to 2. Input and state penalties are defined as well, along with the controller horizon. This in the case of the controller used in experiments is $n_c = 70$ steps.

```matlab
probStruct.norm=2;
probStruct.subopt_lev=0;
probStruct.Q=sysStruct.C'*sysStruct.C;
probStruct.R=1e-4;
probStruct.N=70;
```

The next step is the calculation of the controller regions. The main calling function takes the system and the problem structure as an argument and outputs the multi-parametric MPC problem [40, 41]:

```matlab
ctrlex=mpt_control(sysStruct,probStruct);
```

After the computation, the controller can be saved in its original structure, so it can be later loaded into the workspace or used in offline simulations via the standard Matlab interface:

```matlab
save ctrlex.mat ctrlex;
```

One may need data such as the volume of the region of attraction. This for example may be used to compare different controller designs or models. The total volume of the region of attraction is the sum of the individual volumes, and can be simply calculated by:
result.areareach = sum(volume(ctrlex.Pn))

Similarly, the number of regions of the controller can be determined by:

regions = length(ctrlex);

The maximal absolute deflection of the beam can be calculated by creating a convex hull around the region of attraction, transforming this into a vertex representation and by multiplying the individual edges of the region of attraction with the output matrix \( C = C \) we may get the direct output equivalents. The maximum of this is the maximal possible deflection at the beam tip:

\[
[P, Pn] = \text{hull}(\text{ctrlex}.Pn);
result.V = \text{extreme}(P);
result.maxdef = \text{max}(\text{abs(sysStruct}.C*result.V')');
\]

Non-essential tasks aimed at reducing the memory requirements and online search times of the MPMPC controller [46] such as optimal region merging [34] and binary search tree generation [62] are not considered here.

**B.2.2 Real-Time Deployment**

Exporting the controller into a C code via the MPT Toolbox is a very simple and straightforward process and it can be carried out using the following command [45, 47]:

mpt_exportc(ctrlex);

The function code built through the S-Function Builder takes the state vector \( x_k \) as its input and has a single output, the direct controller voltage. The core code is very simple and only involves calling the routine supplied by the MPT Toolbox in the form [41]:

```c
double region;
region = mpt_getInput(x0, u);
```

where \( x_0 \) is the state vector obtained through an observer block, \( u \) is the controller output. An additional variable \( region \) returns the index number of the polyhedral region corresponding to the acquired current state measurement. For the correct functionality of the S-Function block the sampling rate is given, along with the MPMPC external function declaration:

```c
extern double mpt_getInput(double *, double *);
```

and a declaration of the C source file as well:

```c
mpt_getInput.c
```
B.2.3 Minimum-Time MPMPC

The minimum-time option is set in the MPT Toolbox by declaring:

\[
[P,Pn]=\text{hull}(\text{ctrlex}.Pn);
\]
\[
\text{result.V=extreme}(P);
\]
\[
\text{result.maxdef}=\max(\abs(\text{sysStruct}.C*\text{result.V}'))';
\]

An additional output constraint is needed so the offline minimum-time MPC algorithm can terminate. This is defined through

\[
\text{sysStruct}.ymin=-ym;
\]
\[
\text{sysStruct}.ymin=ym;
\]

where \(\max|ym|\) is the maximal output deflection which is implied through the region of attraction of the controller.

B.3 NRMPC

The NRMPC code implementation will be introduced in the following sections, concentrating on the essential functions and steps themselves. Individual code portions taken from an actual version of the NRMPC application will be explained in detail. However, some non-essential functions will be omitted. These are mainly for diagnostic purposes, testing closed-loop stability or plotting the region of attraction and other graphical aids. This is a working version of the code, therefore some aspects certainly could be made better or more elegant, nevertheless of the several development versions this one too provides the desired parameters for the online run.

This section augments the theoretical discussion presented earlier in Sect. 8.1 and some implementation aspects in Sect. 10.3 by listing actual code segments for both the off and the online NRMPC strategy. This code has been developed for the AVC demonstrator and considers a second order prediction model. This code has been based on the publications by Kouvaritakis and Cannon et al. in [26, 37, 38].

One may further develop this code by introducing the theoretical findings suggested by the more recent works of Kouvaritakis et al. and Lee et al. in [39, 48] to further enhance the optimality of the NRMPC approach.

\[9\] The active help of Prof. Basil Kouvaritakis with the theoretical foundations and Dr. Mark Cannon with the code implementation is acknowledged and very much appreciated.
**B.3.1 Offline Code Implementation**

**B.3.1.1 Initialization: Parameters and Penalties**

The beginning portion of the offline code initializes the algorithm. Amongst others, simulation stop time is defined along with the desired deflection, which in the case of the vibration attenuation example is always zero. A linear time-invariant state-space model is loaded from a file\(^{10}\) and its individual matrices are assigned. Sampling time is also defined as \(T_s = T_s\):

\[
T_{\text{stop}} = 60; \quad y_d = 0; \quad \text{load m2ss.mat;}
\]

\[
A = \text{m2ss.A}; \quad B = \text{m2ss.B}; \quad C = \text{m2ss.C}; \quad D = \text{m2ss.D};
\]

\[
T_s = 0.01;
\]

Other types of settings and tuning parameters are also declared at the beginning of the script file. Symmetric input constraints are stated as \(u_h\),\(^{11}\) along with the state constraints if there are any.\(^{12}\) Simulations and experiments performed on the active vibration attenuation system did not make use of this option. State penalties are set as \(Q = Q_{\text{le}}\) leaving \(Q = C^T C\) which includes the output deflection in the computed cost. Input weighting is declared as the variable \(R = R = r = 1E - 4\):

\[
u_h = 120;
\]

\[
VX = [ ];
\]

\[
Q = C^T C;
\]

\[
R = 1e-4;
\]

Prediction cost performance bound \(\gamma\) is stated, which is necessary to be limited in order to preserve numerical stability of the process (See 11.3). A tolerance limit is also set, which is used to modify the behavior of YALMIP regarding the handling of strict inequalities [49, 50]. For constraints defined with strict constraints, a small perturbation controlled by the tolerance value is added:

\[
\text{gamma} = 1e6;
\]

\[
\text{tolerance} = 1e-8;
\]

Dimensionality of the problem is determined to allow the use of different state-space models for generating predictions. Number of states, inputs and other properties of the problem are acquired and stored for later use:

---

\(^{10}\) See 5.2 for a discussion on the mathematical models. Most simulations and experiments in this work utilize second order models, identified from diverse data sets and identification settings. Practical experiments show no quality difference between these.

\(^{11}\) \(u_h\) meaning \(\bar{u}\), as the upper and symmetric bound on the inputs.

\(^{12}\) The option \([\ ]\) is used for no state constraints.
Matrix square-roots of the penalization variables are taken, which are required in the construction of the invariance condition. Closed-loop linear quadratic gain is calculated, utilizing the prediction model and penalties defined previously. The closed-loop matrix of the system is also calculated:

\[
\sqrt{R} = \text{sqrtm}(R); \quad \sqrt{Q} = \text{sqrtm}(Q);
\]

\[
K = -\text{dlqr}(A_n, B, Q, \sqrt{R} \ast \sqrt{R});
\]

\[
\Phi_0 = (A_n - B \ast K);
\]

### B.3.1.2 Variables and Constraints

Four optimization variables are declared, respecting the dimensionality of the problem. Matrix \(N\)\(^{13}\) is fully parameterized and square, requiring an additional setup option. The rest of the optimization variables are real valued and symmetric:

\[
X_q = \text{sdpvar}(w.\text{dim}(1,1),w.\text{dim}(1,1));
\]

\[
Y_q = \text{sdpvar}(w.\text{dim}(1,1),w.\text{dim}(1,1));
\]

\[
N = \text{sdpvar}(w.\text{dim}(1,1),w.\text{dim}(1,1),'\text{full}');
\]

\[
M = \text{sdpvar}(w.\text{dim}(1,2),w.\text{dim}(1,1));
\]

The invariance condition described by Eqs. (8.20) and (8.49) for the optimized dynamics formulation is translated into linear matrix inequalities. These LMI constrain the semidefinite programming (SDP) problem [53]. The if–else conditional statement constructs the problem according to whether there is a performance bound defined. Optimized prediction dynamics is required to enlarge the region of attraction to ensure sufficient deflection at the beam tip. In practice, the vibration attenuation example requires the performance bound to be always engaged.

The only command worth mentioning in this code portion is set, which instructs the parser YALMIP to construct a constraint in an LMI form. This high-level prototyping language formulation makes implementing changes in constraints fast and easy, letting the programmer concentrate on the high-level problem.

\[
\text{Inv1} = [Y_q,X_q;X_q,X_q];
\]

\[
\text{Inv3} = [\Phi_0 \ast X_q + B \ast M \Phi_0 \ast X_q; N + \Phi_0 \ast Y_q + B \ast M \Phi_0 \ast X_q];
\]

\[
\text{if (gamma} < 1/\text{tolerance)}
\]

\[
\text{gInv} = \text{gamma} \ast \text{eye}(w.\text{dim}(1,5) + w.\text{dim}(1,2));
\]

\[
\text{zInv} = \text{zeros}(w.\text{dim}(1,5) + w.\text{dim}(1,2),2 \ast w.\text{dim}(1,1));
\]

\(^{13}\) Refer to 8.1.3 for details and exact notation of these matrices.
\[ \text{Inv2} = \text{blkdiag} \left( \sqrt{Q}, \sqrt{R} \right) \left[ Yq, Xq; K^* Yq + M, K^* Xq \right]; \]
\[ F = \text{set}(\left[ g Inv, z Inv, \text{Inv2}; z Inv', \right. \]
\[ \text{Inv1, Inv3; Inv2', Inv3', Inv1 \right]; \]
\[ K^* Yq \right. \]
\[ M K^* Xq; \]
\[ \left. K^* Yq \right. \]
\[ M K^* Xq \right. \]
\[ \text{_inv1} \right]; \]
\[ \text{else} \]
\[ F = \text{set}(\left[ \text{Inv1, Inv3; Inv3', Inv1} \right] > 0); \]
\[ \text{end} \]

The if construct checks whether there is an input constraint defined or not. If yes, the feasibility condition defined by (8.50) is translated into the proper LMI and added to the set of constraints defining the SDP problem. Input constraints were engaged in every simulation and experiment considered in this work:

\[ \text{if} \sim \text{isempty} (uh) \]
\[ F = F + \text{set}(\left[ uh^2 \left[ K^* Yq + M K^* Xq; \right. \]
\[ \left. \left[ K^* Yq + M K^* Xq \right]' \text{Inv1} \right] > 0); \]
\[ \text{end} \]

The simple if construct starting this code portion, determines whether there is a state constraint in the problem. None of the experiments or simulations using the vibration cancelling example utilized state constraints, although its inclusion is straightforward according to:

\[ \text{if} \sim \text{isempty} (VX) \]
\[ \text{for } i = 1 : \text{size}(VX,1) \]
\[ F = F + \text{set}(1-\text{VX}(i,:), Yq + VX(i,:)' \text{);} \]
\[ \text{end} \]
\[ \text{end} \]

**B.3.1.3 Solver Setup and Solution Initiation**

Options are passed to the LMI parser and also to the solver, in this case SeDuMi [56]. Verbose diagnostic and progress output is suppressed during the optimization phase. Strict inequality constraints are relaxed and perturbed by the shift setting. To increase numerical precision and prevent violation of the invariance constraint during the online run, solver precision is set to the maximal level. The particular zero setting instructs the solver to search for the solution until progress is made (See 11.3.2 for the reasons of this setting.).

---

14 See B.3.3.6 for possible solvers and SeDuMi.
Solution of the above-defined SDP problem is initiated by the `solvesdp` YALMIP command. The LMI defining constraints are passed onto the solver as the variable $F$, options are contained in the `ops` parameter. The aim of this optimization problem is to maximize the volume of the ellipsoids defining the region of attraction and target set. This can be carried out by utilizing the fact that the volume of an ellipsoid is proportional to its determinant as stated in Eq. (7.77). Instead of directly minimizing (or in this case by the help of minus sign maximizing) the determinant, YALMIP utilizes the command `geomean` for such problems to minimize the geometric mean of eigenvalues:

$$\max(\text{vol}_{E_z}) = -(\det w)^{1/m}$$

where $w$ is the optimization parameter in general and $m$ is the dimension of $w$. In this case, optimization objectives and parameters are $Yq$ and $Xq$, defining the projection and intersection of the augmented ellipsoid into $x$ space. Not only is the largest possible region of attraction desired, but also a large target set enables the system to switch to LQ as soon as possible. It is desirable to maximize the volumes of ellipsoids defined by $Yq$ and $Xq$ at the same time, by including them in a block diagonal construct.

In practice, the optimization task can be performed in the order of seconds. After it is completed, YALMIP passes problem diagnostics into the parameter `info`. Optimization parameters $Yq, Xq, N$ and $M$ are converted into the standard double precision matrix format, from the YALMIP optimization variable notation:

```matlab
info = solvesdp(F, -geomean(blkdiag(Yq, Xq)), ops);
Yq = double(Yq); Xq = double(Xq);
N = double(N); M = double(M);
```

### B.3.1.4 Factoring, Storing and Preparing Parameters for Online NRMPC

After the optimization variables are available, the parameters used in the online NRMPC run have to be factored out and stored. These tasks are largely to be attributed to the inclusion of prediction dynamics in the optimization problem, and the resulting mathematical operations to preserve convexity.
Since the augmented ellipsoid is defined by matrix $\Gamma_z$ and it is also obvious that $\Gamma_z^{-1}\Gamma_z = I$, a resulting identity expressed in (8.45) helps to obtain variables $V$ and $X^{-1}U$ by the use of LU factorization. After this, (8.41) is used to reconstruct the original matrices defining the augmented hyperellipsoid.

This is actually a code segment directly resulting from Eqs. (8.45) and (8.41), yielding matrix $\Gamma_z$ from the optimization variables in several steps:

$$ [V, X_i U] = \text{lu} \left( \text{eye} \left( w \cdot \text{dim}(1, 1) \right) - Yq / Xq \right); $$

$$ X_i U = X_i U'; $$

$$ Qzi = [\text{inv} (Xq), X_i U; X_i U', -V \backslash Yq * X_i U]; $$

$$ Qz = [Yq, V; V', -(Xq * X_i U) \backslash V]; $$

Code segments $-V \backslash (Yq * X_i U)$ and $-(Xq * X_i U) \backslash V$ actually implement mathematical operations $-V^{-1}YX^{-1}U$ and $-(XX^{-1}U)^{-1}V$.

The full, optimized shift matrices $T(A_0)$ and $E(C_0)$ are calculated according to the relations defined in (8.43). Here the code segment $(Xq * X_i U) \backslash (N/V')$ is equivalent to the operation $(XX^{-1}U)^{-1}KVT^{-1}$. A matrix right division is used in the segment $M/V'$ to implement the operation $\Theta/(V^T)^{-1}$. After this, respective partitions of $\Gamma_z$ are stored in variables for the needs of the online NRMPC code. It is true that partitions $\Gamma_{zf}, \Gamma_{fx}$ are related in symmetry as $\Gamma_{zf} = \Gamma_{fx}^T$, which is true up to a numerical precision of $1E - 12$ in this particular optimization task:

$$ A_0 = (Xq * X_i U) \backslash (N/V'); $$

$$ C_0 = M/V'; $$

$$ Q_{11} = \text{inv}(Xq) $$

$$ Q_{12} = X_i U; $$

$$ Q_{21} = X_i U'; $$

$$ Q_{22} = -V \backslash Yq * X_i U); $$

$$ Kt = [K C_0]; $$

### B.3.1.5 Cost Transformation

The following code segment is related to cost transformation, and the resulting conversion of augmented states. The Lyapunov equations in (10.4) and (10.5) are solved successively in the offline code to obtain the cost transformation matrix

---

15 This code implementation uses $Q$ to denote the ellipsoid definition matrices $\Gamma$

16 $X^{-1}U$ is noted as X_i U in the code.

17 Implemented by the command lu in Matlab, expressing a matrix as the product of two essentially triangular matrices, one of them a permutation of a lower triangular matrix and the other an upper triangular matrix.

18 $\Theta$ in the text is denoted as $M$ in the implemented code.
partitions. The Matlab command `dlyap` is used to evaluate the equations as follows:\footnote{The text used notation $L$ for the cost transformation matrix and its partitions, here $M$ is used to denote the same concept.}
\begin{align*}
Mx &= \texttt{dlyap}(\Phi0',Q+K'*R*K); \\
Mc &= \texttt{dlyap}(A0',C0'*(R+B'*Mx*B)*C0);
\end{align*}

The transformation matrix $\text{inv}T$ is constructed using the decomposition and scaler, finally assembling the actual transformation matrix $\text{invTT}$:
\begin{align*}
[V,D]=\texttt{eig}(Mc); \\
d=\texttt{sqrt}(\text{max}(\texttt{diag}(D),\text{tolerance})); \\
\text{invT} &= V*\texttt{diag}(1./d)/V; \\
\text{invTT} &= \texttt{blkdiag(eye}(w.\dim(1,1)),\text{invT});
\end{align*}

Select parameters are passed onto the online formulation, while some minor practical operations are performed in the final code segment. Computation of $\Delta$ and $S$\footnote{$S$ is denoted in Sect. 10.3.3 as $Sv$ in the code segment while $\Delta$ is denoted as $R$. In this context $R$ is not the input penalization matrix rather the result of the eigenvalue decomposition.} as introduced in Sect. 10.3.3 is carried out, along with preparing a transformed augmented ellipsoid matrix ($Pt$) and its partition ($Q21$):
\begin{align*}
Pt &= \text{invTT}'*Qzi*\text{invTT}; \\
[R,S]=\texttt{eig}(Pt(\dim+1:2*\dim,\dim+1:2*\dim)); \\
Sm &= \texttt{diag}(S); \\
Q21 &= Pt(\dim+1:2*\dim,1:}\dim);
\end{align*}

### B.3.2 Online Code Implementation

#### B.3.2.1 BLAS Under the RTW Toolbox

The NRMPC code has been developed in Matlab, running under the Microsoft Windows operating system. Therefore, this version of the Matlab software contains an operating system and architecture optimized version of BLAS, compiled into a dynamically linked library with the file name “libmwblas.dll”. It is possible to utilize operations from the BLAS library if one creates a custom S-Function in Simulink using the C programming language. It is essential to refer to the BLAS documentation and call the routine within the code according to the correct syntax [18, 27]. In addition to that, the external function has to be declared: for example, a function declaration for a double precision general matrix-matrix multiplication is in the form:
extern void dgemm_ (char *, char *, int *, int *, int *,
double *, double *, int *, double *,
int *, double *, double *, int *);

The following BLAS functions have to be declared at compilation times:

- dgemm—double precision multiplication of two general structure matrices
- dgemv—double precision multiplication of a general structure matrix and vector
- dsymv—double precision multiplication of a symmetric matrix with a vector
- ddot—dot product of two vectors

In addition to the function declaration, one has to include the library files used by the custom C S-Functions—in this case the precompiled Matlab optimized BLAS library in the form:

libmwblas.dll

Conforming to these prerequisites, it is possible to use the BLAS routines in Simulink, if one runs the simulation on the computer on which the Matlab optimized BLAS libraries are originally included. However, if an S-Function containing BLAS routines is to be used in the Real-Time Workshop running under the xPC-Target kernel, the precompiled BLAS libraries are no longer suitable.

In case a custom Simulink S-Function making use of BLAS has been compiled for a Matlab distribution for the Microsoft Windows operating system, it will not be able to run under other platforms—unless the BLAS library files are compiled for that specific environment. This is also true for xPC target. An S-Function is perfectly suitable to be used in simulations on the computer running Microsoft Windows, but it will fail to function correctly under the xPC kernel. This is because the xPC kernel is a DOS-like environment, requiring a different BLAS library.

The problem is that neither a Windows nor a DOS optimized precompiled BLAS library will be suitable for this purpose. At the S-Function compilation stage, a Windows compatible library is required, but as soon as the controller is loaded onto the xPC machine, a DOS compiled library is needed with no Windows-specific memory calls.

To solve this dilemma, the NRMPC online implementation uses a custom compiled BLAS library [57]. A viable way to do this is to use cygwin a Linux style API, running under the Microsoft Windows operating system [28]. To prepare a suitable environment for the custom built BLAS packages, in addition to the core cygwin installation some packages have to be included:

- Download and install cygwin.
- Install gcc compiler for C and F77. (To be found in “dev”.)
- Install LAPACK native. (To be found in “math”.)

The software environment is now ready to make a custom BLAS library. It is needed to unzip the package and copy the appropriate makefile [55] by typing the following lines into the terminal:
The next step is to modify a function called \texttt{xerbla}, which is only used for error reporting. This may cause problems throughout the build process, therefore it is advised to delete all functionality and leave only the core function. In case it is called, it would return \texttt{void} to the parent function. Now it is needed to modify the file \texttt{make.inc} by changing the following lines:

\begin{verbatim}
FORTRAN = g77  
LOADER = g77  
\end{verbatim}

This is to specifically instruct the compiler to make a library, which is capable of running outside \texttt{cygwin}. Therefore, the previously introduced lines have to be found in \texttt{make.inc} and changed to:

\begin{verbatim}
FORTRAN = g77 -mno -cygwin  
LOADER = g77 -mno -cygwin  
\end{verbatim}

The make file has to be setup to compile the default BLAS library. To do this, one needs to change the line starting with \texttt{lib} in the \texttt{makefile} as follows:

\begin{verbatim}
lib: blaslib lapacklib tmglib  
\end{verbatim}

The package is compiled, tested and timed by typing \texttt{make} into the \texttt{cygwin} terminal. In addition to the xPC compatible BLAS library, a LAPACK library is generated. LAPACK is used for more complex operations, currently not required for the NRMPC implementation\textsuperscript{21}. It is worth noting that by using a BLAS library prepared in the previous method, they are capable of running under the xPC Target kernel, even though the S-Functions are compiled using Microsoft Visual C v. 6 \textsuperscript{52}.

\textbf{B.3.2.2 Custom C Functions}

In addition to the main C code and BLAS functions for matrix algebra operations, there are two additional custom functions. One of them performs element-wise division of two vectors, similar to the \texttt{/} operator in Matlab. The other one is an algorithm-specific operation, scaling the elements of a vector by a scalar value and subtracting it from one.

Both these functions have to be declared similar to the BLAS routines as externals. No library declarations are needed, since instead of using a dynamically linked library, the C source codes are necessary at compilation. Function declarations are as follows:

\begin{verbatim}
496 Appendix B: MPC Code Implementation Details  
\end{verbatim}

\textsuperscript{21} However, the one step-ahead extension not currently used would require solving sets of linear equations, which indicates the need for LAPACK functions.
The following function performs element-wise division of two vectors. Vectors \( x \) and \( y \) have a common dimension of \( n \). Their elements are indexed with the locally declared variable \( i \). The function takes the dimension \( n \) and vectors \( x \) and \( y \) as its input, and places the result in vector \( z \). Strictly speaking, this C function has no output—the results are placed in the proper memory location by pointers. The algorithm simply loops through the elements, dividing the \( i \)-th element of \( x \) with \( y \) and places it into \( z \). The trailing underscore in the function name mimics the calling sequence of BLAS functions.

```c
void dediv_(int *n, double *x, double *y, double *z)
{
    int i = 0;
    for (i; i < *n; i++)
    {
        z[i]=x[i]/y[i];
    }
}
```

The next custom is specific for the NRMPC code, its usage is hardly universal. After the unknown \( \lambda \) is calculated by the algorithm, each element of vector \( x \) is scaled by it. This simple scaling operation is extended by an additional step for computational efficiency. Each scaled element of \( x \) is subtracted from 1, and the result is placed in the vector \( y \).

The function takes scalar dimension \( n \), vector \( x \) and scalar \( \lambda \) as its input. The output is placed in \( y \), where the dimensions of both vectors are \( n \). An inside loop performs the formerly described simple operation, where the elements are indexed with the locally defined variables \( i \). Strictly speaking, this C function produces a void output; the results are placed into the output vector by using pointers. The trailing underscore in the function name mimics the calling sequence of BLAS functions.

```c
void descal_(int *n, double *lamN, double *x, double *y)
{
    int i = 0;
    for (i; i < *n; i++)
    {
        y[i]=1-*lamN*x[i];
    }
}
```

B.3.2.3 The NRMPC Code in C

This section introduces the online NRMPC code implementation in C and explains the general operations performed within. The algorithm will be divided into
A prerequisite for the correct functionality of this code is the correct calling of external BLAS functions and the xPC optimized library as described in Sect. B.3.2.1. The two external custom functions described in Sect. B.3.2.2 are also needed to be present and properly declared at compilation time.

At the online control process, the following real-time NRMPC algorithm is called on and evaluated at each sampling interval:

Local variables are declared at the beginning of the code. The BLAS functions require character variables. The transposition of matrices is controlled by N and T-as in not to transpose and transpose. Some of these functions also require to mark, whether the upper or lower triangular portion of a symmetric matrix is read in.

The order of the system is declared, just as some common values as zero, one or minus one. The value of $\lambda$ is set to zero at starting time, tolerance and error thresholds are stated. Finally, local matrix and vector variables are declared:

```c
char *chn = "N", *cht = "T", *chu = "U", *chl = "L";
int onei = 1, order = 2;
double one = 1.0, mone = -1.0, zero = 0.0, lamN = 0, .......tol = 1e-5, err = 2e-5;
double W0, W2, fval, fderval;
double tempv[2], tempm[4], vec[2], tempv2[2], ...
  W1[2], W1d[2], W1dd[2], m[2], f[2];
```

After the local variable declarations are expressed, the following mathematical operation is performed in two steps:

$$W_0 = x_0^T \Gamma_x x_0 - 1$$

where $x_0$ marks the current observed state, and $\Gamma_x$ is a partition of the matrix defining the invariant ellipsoid, as calculated in the offline process. $W_0$ is a by-product, resulting the logical simplification of matrix operations. The first code line creates a temporary vector, a result of the matrix-vector operation $C_{x0}$ while the second finishes the task by evaluating the rest:

```c
dsymv_(chu,&order,&one,Q11,&order,...
  ...x0,&onei,&zero,tempv2,&onei);
W0 = ddot_(&order,x0,&onei,tempv2,&onei)-1;
```

In case the resulting vector will be $W_0 \leq 0$, the following code portion calculates the next by-product, a vector reused in later code portions:

$$v = -(A^T \Gamma_{xf} x_0) ./ S$$

where $v$ denotes the vector result of this operation, and $/.$ is an element-wise division. This operation is carried out in two steps. First, a general matrix-matrix multiplication saves the result of $A^T \Gamma_{xf}$ into a temporary matrix. Then the expression $v$ is calculated by multiplying the result with the negative of the current state measurement, and its elements divided by vector $S$: 498 Appendix B: MPC Code Implementation Details
if \((W_0 \geq 0)\) {
    \texttt{dgemm\_}(cht, chn, \&order, \&order, \&order, \&one, ...
    \dots R, \&order, Q21, \&order, \&zero, tempm, \&order);
    \texttt{dgemv\_}(chn, \&order, \&order, \&mone, tempm, \&order,
    \dots x0, \&one, \&zero, W1, \&onei);
    \texttt{dediv\_}(\&order, W1, Sm, vec);
}

Another partial result is calculated, by evaluating a dot product of two vectors
and adding \(W_0\) to the result:

\[
W_2 = -\texttt{ddot\_}(\&order, vec, \&onei, W1, \&onei) + W_0;
\]

where in case \(W_2 \geq 0\), the perturbation vector can be directly calculated by
evaluating:

\[
f = -Av
\]  

(B.4)

where \(f\) is the perturbation vector, is an input from the offline optimization, and \(v\)
is a vector product recalculated at each sampling interval.

if \((W_2 \geq tol)\) {
    \{\texttt{dgemv\_}(chn, \&order, \&order, \&mone, R, \&order, ...
    \dots vec, \&one, \&zero, f, \&onei);\}
}

The other option in the else construct is to evaluate for the unknown \(\lambda\) using the
Newton–Raphson procedure. This conditional statement launches a \texttt{while} loop,
which cycles through the NR procedure, until the floating point absolute value of
error is larger than the preset tolerance. In practice, loop termination occurs in no
more than twenty iterations.

The first part of this code segment serves only to evaluate the matrices used in the
NR loop. These simplifications increase computational speed and are based on the
assumptions about function \(\chi(\lambda)\) and its \(i\)-th derivatives presented in Sect. 10.3.3.

The second part of the following code segment is the Newton–Raphson
algorithm itself, which searches for the root of (8.26). Here the first step is to
evaluate the value of \(\chi(\lambda)\) and its derivative according to (10.11). The ratio of the
function value and its derivative is the error, which is subtracted from the result for
\(\lambda\) from the previous step:

else{
    \texttt{while}(\texttt{fabs}(err) \geq tol) {
        \texttt{descal\_}(\&order, \&lamN, Sm, m);
        \texttt{dediv\_}(\&order, W1, m, W1d);
        \texttt{dediv\_}(\&order, W1d, m, W1dd);
        fval=\texttt{ddot\_}(\&order, vec, \&onei, W1dd, \&onei) + W2;
        fderval=2*\texttt{ddot\_}(\&order, W1d, \&onei, W1dd, \&onei);
        err=fval/fderval;
        lamN=lamN-err;
    }
}
Since the value of $\lambda$ has been acquired in the previous step, the only task left is to evaluate for the perturbation vector $f$ according to (8.25), which in this case can be also stated as:

$$f = -\lambda T \Delta W_{1d}$$

(B.5)

This single mathematical operation is divided into three parts for the C code. First, the value of $\lambda$ is negated, then a temporary vector is created from the product of $\nu_{\text{temp}} - \lambda \Delta W_{1d}$. The final step is to calculate $f$ by multiplying this temporary vector by $T$ from the left, $f = T \nu_{\text{temp}}$:

```c
lamN = -lamN;
dgemv_(chn, &order, &order, &lamN, R, ...
... &order, W1d, &onei, &zero, &tempv, &onei); 
dsymv_(chu, &order, &one, T, &order, ...
... &tempv, &onei, &zero, f, &onei);
```

With the perturbation value calculated in the previous step, the final task is only to evaluate the current control move according to $u = Kx_0 + C_0 f$. This is performed in the C code by summing up the results of two vector dot operations:

```c
u[0] = ddot_(&order, K, &onei, x0, &onei) +
+ ddot_(&order, C0, &onei, f, &onei);
```

The other option implies that the loop is already optimal, thus the perturbation $f = 0$. There is no need for optimization, this is part of an “if-than-else” decision. In this case, the fixed feedback matrix is used to calculate the control move from the observed state by evaluating $u = Kx_0$. This is again a simple vector dot product:

```c
else{
  u[0] = ddot_(&order, K, &onei, x0, &onei);
}
```

### B.3.3 SDP Solvers for NRMPC

The following passages give a brief description of SDP solver choices.

#### B.3.3.1 DSDP

DSDP is a freely available and open source semidefinite programming solver. It is based on an implementation of an interior point method with relatively low memory requirements, also exploiting data sparsity in the problem structure [19, 21]. This software may be used as a set of subroutines in Matlab. A parallel implementation of DSDP is also available under the name PDSDP [20].
DSDP solver completed the NRMPC offline optimization task, but with warnings informing about the unbounded nature of the primal optimization problem and infeasibility of the dual. The resulting parameters could be used in simulation. Evolution of controller outputs showed no signs of numerical irregularities. However, the outputs remained highly suboptimal and not approaching constraint levels, especially in the case of higher than second order examples.

B.3.3.2 SDPLR

SDPLR is a freely available and distributed optimization software for SDP problems [24], which is suited for solving large-scale problems. Essentially, it is a C package, however a Matlab interface is also provided. Source code and architecture optimized versions are available. It is linked with either un-optimized BLAS or the Automatically Tuned Linear Algebra Software (ATLAS)—an optimized improvement of the generic BLAS.

During the evaluation of the offline NRMPC optimization problem, the solver repeatedly crashed. Simulations with the online controller could therefore not be completed. Although the performance and precision of this solver in relation to the NRMPC problem are unknown, its use has not been considered due to the concerns with reliability.

B.3.3.3 CSDP

CSDP is a freely available C library for semidefinite programming [22]. CSDP is written as a callable C subroutine, capable of running in parallel on shared memory and multi-processor systems [23]. The routine makes use of sparsity in constraint matrices, and works on systems with an ANSI C compiler and BLAS/LAPACK libraries.

The CSDP executable has to be included in the Matlab and Windows path as well. The code communicates with Matlab via text files, which is an inefficient method causing overheads for certain problems. The CSDPA solver was not usable with the NRMPC code, therefore its precision, speed and other properties could not be evaluated in relation to the given optimization problem.

B.3.3.4 SDPT3

SDPT3 is a freely distributed solver for semidefinite programming problems [61, 63]. It is a relatively well tested, although still not bug-free software. The last version implements an infeasible path-following algorithm for solving conic optimization problems involving semidefinite, second order and cone constraints. It can handle determinant maximization problems and SDP with complex data.
The simulation run produced by using online controller parameters acquired via the SDPT3 solver produced numerically unstable results. The controller output exceeded constraints by orders of magnitude higher values. The evolution of outputs did not even remotely resemble the expected oscillating behavior. Adjusting solver gap and step tolerances, increasing the maximal allowable iteration counts did not improve the situation. After the negative experience with this solver, the use of SDPT3 has not been considered for the NRMPC algorithm.

B.3.3.5 SDPA-M

SemiDefinite Programming Algorithm or SDPA is a collection of software tools for solving SDP problems [65]. Its Matlab compatible interface is called SDPA-M. The algorithm is implemented in C++ language, and it is making use of BLAS or some variant of optimized BLAS and LAPACK packages for matrix computations. The algorithm implements an infeasible primal-dual interior point method [65].

An important feature of SDPA is that it is callable as a C function, therefore readily implementable in future possible versions of adaptive NRMPC algorithms, or implementations where the offline portion is also present on the hardware at setup time. In addition to handling block diagonal and sparse data structures, it is able to exploit the sparsity of problem data matrices by using efficient methods for finding search directions. SDPA is also available for parallel computing applications. The Matlab callable version provides no callable library and at the time of completing this work and does not support 64-bit platforms [33].

Running SDPA-M with the problem and settings used to evaluate the rest of solver candidates produced no error messages. However, the resulting optimization parameters were singular matrices. Clearly, this is an unacceptable solution; simulations cannot be performed with the results obtained via SDPA-M. The use of SDPA-M for the NRMPC offline algorithm was not considered viable.

B.3.3.6 SeDuMi

SeDuMi is a second order cone and semidefinite programming problem solver software, freely distributed and available with sources and binaries [56]. SeDuMi can be compiled on any platform running Matlab. The latest version uses BLAS for improved performance and is able to run on 64-bit systems. SeDuMi is the most commonly used solver among YALMIP users. Installation procedure of SeDuMi is as simple as copying the directory containing solver files into the Matlab path. Simulations performed with solutions acquired through SeDuMi showed no signs of serious offline solution suboptimality and therefore has been selected as the first choice of SDP solver for the offline NRMPC routine.
B.3.3.7 LMILAB and PENSDP

LMILAB and PENSDP are commercial solvers. PENSDP is a linear semidefinite programming solver, aimed at large-scale dense and sparse problems. It is available as a standalone program, a MATLAB function callable through YALMIP and as a C routine as well. PENSDP is offered for all major computer architecture systems [36]. At the time of preparing the work in this book, a student or trial license was not available, therefore the use of PENSDP as a basis of the offline NRMPC procedure has been dismissed. The efficiency and precision of this SDP solver algorithm in relation to NRMPC has not been tested and evaluated.

LMILAB is a part of the MathWorks Robust Control Toolbox. The use of LMILAB in combination with YALMIP is generally not recommended [50]. YALMIP cannot exploit a built-in feature in LMILAB, which would formulate a control specific structure increasing computational efficiency. In addition to the often slow computational times, it has a low default tolerance setting and does not return infeasibility flags to YALMIP. YALMIP reports a successful optimization even when the problem is infeasible, hence making algorithm testing and development somewhat difficult. Due to the above stated issue, the use of LMILAB as a SDP solver for the offline NRMPC problem has not been considered.

References

10. MIDÉ Technology Corporation (2008) Attaching the Quickpack/PowerAct Transducer to a Structure with Epoxy. MIDÉ Technology Corporation, quick Pack Technical Notes. MIDÉ Technology Corporation, Medford
Appendix B: MPC Code Implementation Details

Legal Information

Disclaimer

This book has been written by members of the faculty of the Slovak University of Technology in Bratislava as a part of ongoing active research projects. It has been altered and corrected within the means of the usual scientific peer-review process, however, may still contain errors or mistakes.

The book is “as-is” without any guarantees to accuracy, reliability or validity of the expressed views and published data. The book is provided without warranty of any kind, and no other warranties, either expressed or implied are made with respect to this book. The authors of this book and the publisher bear no responsibility and cannot be held liable, including but not limited to: health damage, potential loss of life, loss of property or finances or any other inconvenience as a result of using information published in this monograph.

Endorsement

This work has not been endorsed nor sponsored by any company, corporation, domestic or foreign government entity except the Slovak Research and Development Agency. Company names and government entities are only used in this book for the purpose of identification and reference.

Trademarks

This work makes no claims to trademarks, company or product names. Protected and registered trademarks are only used in the text to refer to products, applications or companies without any intent to infringe. Registered trademarks, company and product names are the properties of their respective owners.

Software Licensing

All work has been carried out on software licensed to the Slovak Technical University in Bratislava. Inquiries on licensing shall be made to the University or its institutes. The authors take no responsibility on the content of the agreements.
and contracts between the Slovak Technical University in Bratislava and its respective institutes and software vendors.

**Graphic Images**

Images, photos, diagrams, illustrations, typesetting, the graphic layout and structure of this document are the work of the authors, unless otherwise indicated. The typesetting of the document makes use of the layout materials provided by the publisher.

This book contains reprinted photographs and illustrators as well. The authors have made every reasonable effort to trace and contact the copyright owners of these materials in order to ask for permission. Despite our numerous efforts we have not received responses in all cases. The authors apologize if permission has not been obtained or copyrighted material is used without due acknowledgment. Please notify us so we can correct these unintentional mistakes in future editions.

Photos released by the United States National Aeronautics and Space Administration are used according to the general terms and conditions of the owner. Images released by NASA to the public domain are not copyrighted and may be used for educational or informational purposes. Written permissions have been obtained to use images from the United State National Aeronautics and Space Administration (NASA), from the European Space Agency (ESA), the Smithsonian Institution, Liebherr-International Deutschland GmbH, the CEDRAT Group, the Japan Society of Maintenology, the Noise and Vibration Control Ltd, Dr. Thomas Huber, Dr. Bishakh Bhattacharya. The image reproduced from Popular Science Monthly is in the public domain because its copyright has expired. Photographs and illustrations reproduced from the Wikimedia Commons database are cleared to use in publications, limited copyright notices are indicated in the references—if applicable. The image obtained from the morgueFile database is free to use for commercial purposes without attributing the original author. The use of photographs and images published under the Creative Commons Attribution 3.0 Unported license permits copy, distribution, adaptation for commercial purposes with attribution. The image released by the author to the public domain is free for use for any purpose without restriction under copyright.

<table>
<thead>
<tr>
<th>Source</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>NASA</td>
<td>4,74, 76,81,87, 91,120,376,379,380,381</td>
</tr>
<tr>
<td>European Space Agency (ESA)</td>
<td>4, 30</td>
</tr>
<tr>
<td>The Boeing Company</td>
<td>9,163</td>
</tr>
<tr>
<td>Smithsonian Institution</td>
<td>3</td>
</tr>
<tr>
<td>Liebherr International</td>
<td>373</td>
</tr>
<tr>
<td>CEDRAT Group</td>
<td>84</td>
</tr>
<tr>
<td>Noise &amp; Vibration Control Ltd</td>
<td>377</td>
</tr>
<tr>
<td>Japan Society of Maintenology</td>
<td>367</td>
</tr>
<tr>
<td>Popular Science Monthly</td>
<td>2</td>
</tr>
<tr>
<td>Martin Helfer / FKFS Stuttgart</td>
<td>365</td>
</tr>
<tr>
<td>Thomas Huber</td>
<td>208</td>
</tr>
</tbody>
</table>
Credit, Citations and Referencing

Every effort has been made to properly reference and give credit to authors whose work has been used here. Published books, articles, papers, theses, manuals and other printed and electronic works are listed in the References sections of individual chapters. In certain select cases, credit may be given in footnotes for personal consultations, ideas and others. Special attention has been devoted to keep the list of references accurate, however if you feel something is missing, its absence is entirely unintentional. Please contact us so we can correct our mistakes in the upcoming editions.
Index

A
Accelerometer, 66, 187, 188, 194, 195
Aircraft, 68, 73, 82, 145, 219, 327, 332, 343–345, 427
Analog inputs, 194
ANC see Noise control
ANSYS, 88, 89, 142, 172–176
AVC see Active Vibration control

B
Bandwidth, 72, 92, 151–153, 160, 167, 174, 175, 185, 192, 194, 195, 287, 308, 327, 328, 330, 331, 334, 368, 409, 436, 437, 439, 441, 446, 447, 450
Bang-bang control, 458
Beam
  clamp, 129
  design, 187
  dimensions, 186
  material, 186
  PZT bonding, 184
  support structure see Beam-clamp, 187
BLAS, 383, 384
Block matrix, 223

C
C language, 402
CCD, 189
Connector block, 190
Constraints
  input, 276, 314, 315, 382
  output, 7
  state, 300
  terminal state, 314
Controller fuzzy, 106, 123, 127–129, 328, 334, 346
  genetic algorithm, 106, 123, 125, 126, 129, 328
  neural network, 106, 123–125, 128, 329, 334
  performance, 207
Convergence, 260, 261, 311


511
C (cont.)
Cost
infinite horizon cost, 118, 223, 257, 261, 262
terminal cost, 222, 257, 258, 259, 364

D
Damping
Rayleigh, 46
structural, 88
Data acquisition, 182, 193
DC gain, 246
Diagonal matrix, 109, 274
Disturbance, 162, 194, 195, 335, 336, 368, 402, 429, 437
chirp signal, 439
frequency domain see Disturbance—chirp signal, 439
initial deflection, 431
pseudo-random, 442
Double precision (number)
DSP, 288
Dynamic model, 143, 209, 344, 394

E
Eigenvalue, 273, 274, 277, 297, 298, 317, 381
decomposition, 298, 381
Electric field strength, 87
Electroactive (polymer), 92, 93
Electrochemical, 67, 145, 362, 394
Electrostriction, 73, 75–77, 83, 90, 362, 394
Ellipsoid, 275, 296
augmented, 382
extension of, 294
hyperellipsoid, 276, 277
projection, 296, 304
shape conditioning, 278, 304
shortest distance from origin, 297
volume by determinant, 296
volume of, 278
Ellipsoidal, 256, 266, 275, 290, 291, 293, 295, 405, 414, 456, 457
Epoxy resin, 184
ES see electrostriction
Experiment, 73, 82, 141, 156, 157, 159, 164–168, 171, 412, 429, 431, 432, 434, 437, 439, 441, 442, 445, 446, 447, 456
Experimental device, 142, 186
Explicit device, 142, 186
Explicit MPC see MPMPC, 306

F
FEA, 463
APDL, 464
coupled field, 464
harmonic, 166, 176
macro, 464
modal, 164
SOLID45, 464
SOLID5, 464
SOLID98, 464
static, 173
transient, 180
FEM, 164, 172, 176, 179, 186
Ferrofluid, 78
Ferromagnetism, 74
FFT, 94
Fixed target tracking, 246
FPGA, 288
Fuzzy controller see Controller

G
Genetic algorithm see Controller

H
Helmholz resonator, 2
Horizon
control, 245, 296, 304, 316
prediction, 245, 279, 294, 298, 317, 392, 398
Hot starting, 317

I
Identification, 93, 141, 146, 150–153, 156, 157, 166, 177, 186, 191, 192, 207, 409, 429
Infinite horizon cost see Cost
Input changes, 247
Input penalty see Penalization
Integration, 247
Invariant set
  augmented, 298, 301
  elliptic, 275, 298, 301
Isopropanol, 184
Isopropyl alcohol see Isopropanol, 184

K
Karush-Kuhn-Tucker Conditions, 315

L
Lagrange multipliers, 297, 314, 315
Laser triangulation, 189
Laser vibrometer see Optical distance measurement, 187
Linear matrix inequality, 277, 278, 296, 302, 304
LQ control, 291, 405, 412, 418, 429
Lyapunov (equation), 223, 255, 257–261, 275, 379

M
Magnetorheological (fluid), 77–82, 90, 106, 123, 124, 129, 333, 334, 336, 337, 339, 344, 345
Magnetostriction, 73–77, 339, 394
Matlab, 146, 166, 177, 60
Minimum-time MPMPC see MT MPMPC, 3011
Modal analysis, 165, 174, 176, 40
Model reduction, 55
MPC, 207, 271
  efficient, 298, 313
  Pontryagin’s principle, 313, 315
ENRMPIC see NRMPC, 301
MPMPC see MPMPC, 291
NRMPC see NRMPC, 291
MPMPC, 192, 306, 368, 398, 431, 439, 454
  computation time, 399
  cost, 371, 372
  executable size, 402
  implementation, 368
  regions, 371, 401
MPT Toolbox, 364, 368–370, 395
MR see magnetorheological, 77

MS see magnetostriction
MT MPMPC, 311

N
Neural network see Controller
Noise control, 332
NRMPC, 291, 298
  adaptive, 459
  extended, 298, 301
  invariance violation, 402
  performance bounds, 405
  solver precision, 407
  multi-model, 459
  optimality, 291, 301, 408, 414, 459
  optimized dynamics, 301

O
Observer, 129
Open-loop, 110, 114, 172, 256
Optical distance measurement, 187
  CPU, 190
  internal memory, 191
  Laser Doppler, 146
  laser head, 189
  placement, 191
  resolution, 191

P
Penalization, 412
  input penalty, 169, 223, 370, 412–414, 416, 430, 431
  penalty matrix, 169, 220, 222, 364, 370
  state penalty, 169, 220, 229, 265, 270, 364, 370, 379, 407, 409, 429
Performance see Controller-performance
P (cont.)
Permittivity, 87
Phase, 67, 69–71, 81, 90, 111, 145, 149
Piezoelectric
amplifier see Voltage amplifier, 185
effect, 82
polarization, 87
simulation, 464
transducer, 182
Pontryagin’s principle see MPC, 313
Positive position feedback, 327, 332, 346
PPF control, 5
Prediction system
augmented, 292
autonomous, 292
Problem definition, 15
Programming
quadratic, 274, 298, 314, 380
active set, 314
second order cone, 278, 279
semidefinite, 277, 278, 295, 298, 300, 304
Proximity sensor, 148, 150, 156, 167, 168, 170, 171, 184, 191, 192, 456
Q
QP see Programming, 274
QpOASES, 244, 290, 365, 367, 368
R
Real-time control, 192, 431, 439, 442, 444
host computer, 192
target computer, 193
Resonance, 115, 152, 155, 160, 164, 176, 179, 181, 194, 313, 335, 340, 342, 344, 439, 441, 448
Rotor, 84, 111, 122, 125, 145, 338, 339, 345, 346, 362
S
Sampling, 189
Schur complements, 277, 295, 303
SDP see Programming, 304
Shaker see Electrodynamic shaker
amplifier, 195
Sliding mode control, 129
Smart material, 65
active wing, 4
electro and magnetorheological, 77
electro and magnetostrictive, 73
helicopter rotor, 4
piezoelectric, 82, 182
PZT see Piezoelectric
PZT5A, 183
SMA, 67
SMA aircraft inlet, 73
SOCP see Programming, 279
Spacecraft, 122, 219, 306, 329, 343, 346, 348
Speed, 444
offline, 398
online, 377, 383, 444
Square matrix, 220, 225
SRF control, 5
Stability see Controller, 406
State see Penalization, 406
State trajectory, 410
State-space representation, 213, 292
Strain, 70–72, 74–76, 89, 82–84, 86–89, 111, 150, 183, 187, 253, 330
Stress, 68–70, 75, 76, 78, 80, 87–89, 185, 253
Suboptimality, 175, 300, 312, 409, 410, 413, 414, 416, 418, 420, 435, 436, 439, 455
System identification, 150
experimental, 150
from FEM model, 67
T
TEFLON, 184
Terminal cost see Cost
V
Velocity feedback, 81, 105, 108, 109, 111, 172, 344
Vibration attenuation see Active vibration control
active, 82
<table>
<thead>
<tr>
<th>Index</th>
<th>515</th>
</tr>
</thead>
<tbody>
<tr>
<td>semi-active, 111</td>
<td>Y</td>
</tr>
<tr>
<td>state switched resistive, 111</td>
<td>YALMIP, 278, 377, 378, 382</td>
</tr>
<tr>
<td>Vibration damping <em>see</em> Active vibration control</td>
<td></td>
</tr>
<tr>
<td>Voltage amplifier, 185</td>
<td></td>
</tr>
</tbody>
</table>