Object-Oriented Interface to OCP QPs in `tmpc`

Mikhail Katliar

Max Planck Institute for Biological Cybernetics
& Systems Control and Optimization Laboratory,
University of Freiburg

Syscop Group Retreat
September 18, 2017

Table of Contents

1 Introduction to `tmpc`

2 Obect-Oriented Interface to OCP QPs

Brief Introduction of tmpc

tmpc is a C++ library for MPC that:

- provides you with ingredients to build your MPC controller;

Brief Introduction of `tmpc`

`tmpc` is a C++ library for MPC that:

- provides you with ingredients to build your MPC controller;
- allows you to make your own recipe (does not enforce the way the ingredients are combined);

Brief Introduction of tmpc

tmpc is a C++ library for MPC that:

- provides you with ingredients to build your MPC controller;
- allows you to make your own recipe (does not enforce the way the ingredients are combined);
- tries to be convenient;

Brief Introduction of tmpc

tmpc is a C++ library for MPC that:

- provides you with ingredients to build your MPC controller;
- allows you to make your own recipe (does not enforce the way the ingredients are combined);
- tries to be convenient;
- tries to be efficient.

The Ingredients Provided by `tmpc`

- Defining system dynamics and sensitivities

The Ingredients Provided by `tmpc`

- Defining system dynamics and sensitivities
- Integrators

The Ingredients Provided by `tmpc`

- Defining system dynamics and sensitivities
- Integrators
- Quadratic Programming

The Ingredients Provided by tmpc

- Defining system dynamics and sensitivities
- Integrators
- Quadratic Programming
- Sequential Quadratic Programming (SQP)

The Ingredients Provided by `tmpc`

- Defining system dynamics and sensitivities
- Integrators
- Quadratic Programming
- Sequential Quadratic Programming (SQP)
- Other Nonlinear Programming (NLP)

The Ingredients Provided by `tmpc`

- Defining system dynamics and sensitivities
- Integrators
- Quadratic Programming
- Sequential Quadratic Programming (SQP)
- Other Nonlinear Programming (NLP)
- Realtime Iteration Scheme (RTI)

The Ingredients Provided by `tmpc`

- Defining system dynamics and sensitivities
- Integrators
- Quadratic Programming
- Sequential Quadratic Programming (SQP)
- Other Nonlinear Programming (NLP)
- Realtime Iteration Scheme (RTI)
- ...

Design Objectives

- Modularity

Design Objectives

- Modularity
  - Easy to replace a component with a different one without affecting the rest of the system

Design Objectives

- Modularity
  - Easy to replace a component with a different one without affecting the rest of the system
- Extensibility

Design Objectives

- Modularity
  - Easy to replace a component with a different one without affecting the rest of the system
- Extensibility
  - Easy to implement a new component which can interact with other components

Design Objectives

- Modularity
  - Easy to replace a component with a different one without affecting the rest of the system
- Extensibility
  - Easy to implement a new component which can interact with other components
- Maintainability

Design Objectives

- Modularity
  - Easy to replace a component with a different one without affecting the rest of the system
- Extensibility
  - Easy to implement a new component which can interact with other components
- Maintainability
  - The code is human-readable and allows testing, debugging and modification

Design Objectives

- Modularity
  - Easy to replace a component with a different one without affecting the rest of the system
- Extensibility
  - Easy to implement a new component which can interact with other components
- Maintainability
  - The code is human-readable and allows testing, debugging and modification
- Performance

Design Objectives

- Modularity
  - Easy to replace a component with a different one without affecting the rest of the system
- Extensibility
  - Easy to implement a new component which can interact with other components
- Maintainability
  - The code is human-readable and allows testing, debugging and modification
- Performance
  - Efficient matrix arithmetics

Design Objectives

- Modularity
  - Easy to replace a component with a different one without affecting the rest of the system
- Extensibility
  - Easy to implement a new component which can interact with other components
- Maintainability
  - The code is human-readable and allows testing, debugging and modification
- Performance
  - Efficient matrix arithmetics
  - No unnecessary memory copying

Design Objectives

- Modularity
  - Easy to replace a component with a different one without affecting the rest of the system
- Extensibility
  - Easy to implement a new component which can interact with other components
- Maintainability
  - The code is human-readable and allows testing, debugging and modification
- Performance
  - Efficient matrix arithmetics
  - No unnecessary memory copying
  - No unnecessary dynamic memory allocation

Design Objectives

- Modularity
  - Easy to replace a component with a different one without affecting the rest of the system
- Extensibility
  - Easy to implement a new component which can interact with other components
- Maintainability
  - The code is human-readable and allows testing, debugging and modification
- Performance
  - Efficient matrix arithmetics
  - No unnecessary memory copying
  - No unnecessary dynamic memory allocation
  - No unnecessary dynamic polymorphism

Design Objectives

- Modularity
  - Easy to replace a component with a different one without affecting the rest of the system
- Extensibility
  - Easy to implement a new component which can interact with other components
- Maintainability
  - The code is human-readable and allows testing, debugging and modification
- Performance
  - Efficient matrix arithmetics
  - No unnecessary memory copying
  - No unnecessary dynamic memory allocation
  - No unnecessary dynamic polymorphism
- Robustness

Design Objectives

- Modularity
  - Easy to replace a component with a different one without affecting the rest of the system
- Extensibility
  - Easy to implement a new component which can interact with other components
- Maintainability
  - The code is human-readable and allows testing, debugging and modification
- Performance
  - Efficient matrix arithmetics
  - No unnecessary memory copying
  - No unnecessary dynamic memory allocation
  - No unnecessary dynamic polymorphism
- Robustness
  - Minimize possibility of resource leaks

Design Objectives

- Modularity
  - Easy to replace a component with a different one without affecting the rest of the system
- Extensibility
  - Easy to implement a new component which can interact with other components
- Maintainability
  - The code is human-readable and allows testing, debugging and modification
- Performance
  - Efficient matrix arithmetics
  - No unnecessary memory copying
  - No unnecessary dynamic memory allocation
  - No unnecessary dynamic polymorphism
- Robustness
  - Minimize possibility of resource leaks
  - Minimize possibility of memory access violations

Design Objectives

- Modularity
  - Easy to replace a component with a different one without affecting the rest of the system
- Extensibility
  - Easy to implement a new component which can interact with other components
- Maintainability
  - The code is human-readable and allows testing, debugging and modification
- Performance
  - Efficient matrix arithmetics
  - No unnecessary memory copying
  - No unnecessary dynamic memory allocation
  - No unnecessary dynamic polymorphism
- Robustness
  - Minimize possibility of resource leaks
  - Minimize possibility of memory access violations
  - Minimize possibility of ignoring an error

Cool Features of tmpc

- `tmpc` widely uses static polymorphism. Therefore, it can be seen as a code-generation tool, with the code-generation done by a C++ compiler.

[1] Currently only some of them.

Cool Features of tmpc

- `tmpc` widely uses static polymorphism. Therefore, it can be seen as a code-generation tool, with the code-generation done by a C++ compiler.
- `tmpc` does not rely on a specific matrix arithmetics implementation. Algorithms[1] are parameterized by a class that defines an implementation of matrix arithmetics.

[1] Currently only some of them.

Current Status

- Defining system dynamics and sensitivities

Current Status

- Defining system dynamics and sensitivities
  - Implemented a convenient interface to `CasADi`-generated functions

Current Status

- Defining system dynamics and sensitivities
  - Implemented a convenient interface to `CasADi`-generated functions
- Integrators

Current Status

- Defining system dynamics and sensitivities
  - Implemented a convenient interface to `CasADi`-generated functions
- Integrators
  - Several variations of the RK4 integrator implemented

Current Status

- Defining system dynamics and sensitivities
  - Implemented a convenient interface to `CasADi`-generated functions
- Integrators
  - Several variations of the RK4 integrator implemented
  - Petr Listov is trying to implement collocation (pseudospectral) methods

Current Status

- Defining system dynamics and sensitivities
  - Implemented a convenient interface to `CasADi`-generated functions
- Integrators
  - Several variations of the RK4 integrator implemented
  - Petr Listov is trying to implement collocation (pseudospectral) methods
  - It is not clear how a good integrators interface should look like

Current Status

- Defining system dynamics and sensitivities
  - Implemented a convenient interface to `CasADi`-generated functions
- Integrators
  - Several variations of the RK4 integrator implemented
  - Petr Listov is trying to implement collocation (pseudospectral) methods
  - It is not clear how a good integrators interface should look like
- Quadratic Programming

Current Status

- Defining system dynamics and sensitivities
  - Implemented a convenient interface to `CasADi`-generated functions
- Integrators
  - Several variations of the RK4 integrator implemented
  - Petr Listov is trying to implement collocation (pseudospectral) methods
  - It is not clear how a good integrators interface should look like
- Quadratic Programming
  - `qpOASES` interface is ready

Current Status

- Defining system dynamics and sensitivities
  - Implemented a convenient interface to `CasADi`-generated functions
- Integrators
  - Several variations of the RK4 integrator implemented
  - Petr Listov is trying to implement collocation (pseudospectral) methods
  - It is not clear how a good integrators interface should look like
- Quadratic Programming
  - `qpOASES` interface is ready
  - `HPMPC` interface is ready

Current Status

- Defining system dynamics and sensitivities
  - Implemented a convenient interface to `CasADi`-generated functions
- Integrators
  - Several variations of the RK4 integrator implemented
  - Petr Listov is trying to implement collocation (pseudospectral) methods
  - It is not clear how a good integrators interface should look like
- Quadratic Programming
  - `qpOASES` interface is ready
  - `HPMPC` interface is ready
  - `HPIPM` interface is ready

Current Status

- Defining system dynamics and sensitivities
  - Implemented a convenient interface to `CasADi`-generated functions
- Integrators
  - Several variations of the RK4 integrator implemented
  - Petr Listov is trying to implement collocation (pseudospectral) methods
  - It is not clear how a good integrators interface should look like
- Quadratic Programming
  - `qpOASES` interface is ready
  - `HPMPC` interface is ready
  - `HPIPM` interface is ready
  - Variable stage size supported

Current Status

- Defining system dynamics and sensitivities
  - Implemented a convenient interface to `CasADi`-generated functions
- Integrators
  - Several variations of the RK4 integrator implemented
  - Petr Listov is trying to implement collocation (pseudospectral) methods
  - It is not clear how a good integrators interface should look like
- Quadratic Programming
  - `qpOASES` interface is ready
  - `HPMPC` interface is ready
  - `HPIPM` interface is ready
  - Variable stage size supported
  - Soft constraints support not ready but in progress

Current Status

- Defining system dynamics and sensitivities
  - Implemented a convenient interface to `CasADi`-generated functions
- Integrators
  - Several variations of the RK4 integrator implemented
  - Petr Listov is trying to implement collocation (pseudospectral) methods
  - It is not clear how a good integrators interface should look like
- Quadratic Programming
  - `qpOASES` interface is ready
  - `HPMPC` interface is ready
  - `HPIPM` interface is ready
  - Variable stage size supported
  - Soft constraints support not ready but in progress
  - $\mathcal{O}^3$ condensing implemented

Current Status

- Defining system dynamics and sensitivities
  - Implemented a convenient interface to `CasADi`-generated functions
- Integrators
  - Several variations of the RK4 integrator implemented
  - Petr Listov is trying to implement collocation (pseudospectral) methods
  - It is not clear how a good integrators interface should look like
- Quadratic Programming
  - `qpOASES` interface is ready
  - `HPMPC` interface is ready
  - `HPIPM` interface is ready
  - Variable stage size supported
  - Soft constraints support not ready but in progress
  - $\mathcal{O}^3$ condensing implemented
  - QP output to `MATLAB`

Current Status

- Defining system dynamics and sensitivities
  - Implemented a convenient interface to `CasADi`-generated functions
- Integrators
  - Several variations of the RK4 integrator implemented
  - Petr Listov is trying to implement collocation (pseudospectral) methods
  - It is not clear how a good integrators interface should look like
- Quadratic Programming
  - `qpOASES` interface is ready
  - `HPMPC` interface is ready
  - `HPIPM` interface is ready
  - Variable stage size supported
  - Soft constraints support not ready but in progress
  - $\mathcal{O}^3$ condensing implemented
  - QP output to `MATLAB`
  - Other useful functions

Current Status

- Defining system dynamics and sensitivities
  - Implemented a convenient interface to CasADi-generated functions
- Integrators
  - Several variations of the RK4 integrator implemented
  - Petr Listov is trying to implement collocation (pseudospectral) methods
  - It is not clear how a good integrators interface should look like
- Quadratic Programming
  - qpOASES interface is ready
  - HPMPC interface is ready
  - HPIPM interface is ready
  - Variable stage size supported
  - Soft constraints support not ready but in progress
  - $\mathcal{O}^3$ condensing implemented
  - QP output to MATLAB
  - Other useful functions
- Sequential Quadratic Programming (SQP)

Current Status

- Defining system dynamics and sensitivities
  - Implemented a convenient interface to `CasADi`-generated functions
- Integrators
  - Several variations of the RK4 integrator implemented
  - Petr Listov is trying to implement collocation (pseudospectral) methods
  - It is not clear how a good integrators interface should look like
- Quadratic Programming
  - `qpOASES` interface is ready
  - `HPMPC` interface is ready
  - `HPIPM` interface is ready
  - Variable stage size supported
  - Soft constraints support not ready but in progress
  - $\mathcal{O}^3$ condensing implemented
  - QP output to `MATLAB`
  - Other useful functions
- Sequential Quadratic Programming (SQP)
  - Implemented in the context of RTI.

Current Status

- Defining system dynamics and sensitivities
  - Implemented a convenient interface to `CasADi`-generated functions
- Integrators
  - Several variations of the RK4 integrator implemented
  - Petr Listov is trying to implement collocation (pseudospectral) methods
  - It is not clear how a good integrators interface should look like
- Quadratic Programming
  - `qpOASES` interface is ready
  - `HPMPC` interface is ready
  - `HPIPM` interface is ready
  - Variable stage size supported
  - Soft constraints support not ready but in progress
  - $\mathcal{O}^3$ condensing implemented
  - QP output to `MATLAB`
  - Other useful functions
- Sequential Quadratic Programming (SQP)
  - Implemented in the context of RTI.
- Realtime Iteration Scheme (RTI)

Current Status

- Defining system dynamics and sensitivities
  - Implemented a convenient interface to `CasADi`-generated functions
- Integrators
  - Several variations of the RK4 integrator implemented
  - Petr Listov is trying to implement collocation (pseudospectral) methods
  - It is not clear how a good integrators interface should look like
- Quadratic Programming
  - `qpOASES` interface is ready
  - `HPMPC` interface is ready
  - `HPIPM` interface is ready
  - Variable stage size supported
  - Soft constraints support not ready but in progress
  - $\mathcal{O}^3$ condensing implemented
  - QP output to `MATLAB`
  - Other useful functions
- Sequential Quadratic Programming (SQP)
  - Implemented in the context of RTI.
- Realtime Iteration Scheme (RTI)
  - Implemented, but I don't like how I did it.

Table of Contents

1 Introduction to `tmpc`

2 Obect-Oriented Interface to OCP QPs

Definition of OCP QP

The OCP QP is a QP in the form[2]

$$\begin{aligned} \underset{x,u}{\text{minimize}} \quad & \sum_{n=0}^{N} \frac{1}{2} \begin{bmatrix} x_k \\ u_k \\ 1 \end{bmatrix}^\top \begin{bmatrix} Q_k & S_k & q_k \\ S_k^\top & R_k & r_k \\ q_k^\top & r_k^\top & 0 \end{bmatrix} \begin{bmatrix} x_k \\ u_k \\ 1 \end{bmatrix} \\ \text{subject to} \quad & x_{k+1} = A_k x_k + B_k u_k + b_k, \, n = 0, \dots, N-1, \\ & \begin{bmatrix} \underline{x}_k \\ \underline{u}_k \end{bmatrix} \leq \begin{bmatrix} x_k \\ u_k \end{bmatrix} \leq \begin{bmatrix} \overline{x}_k \\ \overline{u}_k \end{bmatrix}, \qquad n = 0, \dots, N, \\ & \underline{d}_k \leq \begin{bmatrix} C_k & D_k \end{bmatrix} \begin{bmatrix} x_k \\ u_k \end{bmatrix} \leq \overline{d}_k, \quad n = 0, \dots, N \end{aligned}$$

where u_k are the control inputs, x_k are the states.

[2] Slightly changed Gianluca's notation; x comes before u.

Definition of OCP QP Stage

- The OCP QP consists of 16 elements: Q, R, S, q, r, A, B, b, $\underline{x}$, $\underline{u}$, $\overline{x}$, $\overline{u}$, C, D, $\underline{d}$, $\overline{d}$.

Definition of OCP QP Stage

- The OCP QP consists of 16 elements: Q, R, S, q, r, A, B, b, $\underline{x}$, $\underline{u}$, $\overline{x}$, $\overline{u}$, C, D, $\underline{d}$, $\overline{d}$.
- Each of the elements has a time index k which runs from 0 to N or to $N-1$.

Definition of OCP QP Stage

- The OCP QP consists of 16 elements: Q, R, S, q, r, A, B, b, $\underline{x}$, $\underline{u}$, $\overline{x}$, $\overline{u}$, C, D, $\underline{d}$, $\overline{d}$.
- Each of the elements has a time index k which runs from 0 to N or to $N-1$.

Definition (QP stage)

A OCP QP stage is a combination of elements corresponding to the same time index k:

$$\mathcal{S}_k = (Q_k, R_k, S_k, q_k, r_k, A_k, B_k, b_k, \underline{x}_k, \underline{u}_k, \overline{x}_k, \overline{u}_k, C_k, D_k, \underline{d}_k, \overline{d}_k) \ .$$

Definition of OCP QP Stage Size

Within one stage, matrices and vectors have consistent dimensions:

$$Q_k \in \mathbb{R}^{n_{x,k} \times n_{x,k}},\ R \in \mathbb{R}^{n_{u,k} \times n_{u,k}},\ S \in \mathbb{R}^{n_{x,k} \times n_{u,k}},\ q \in \mathbb{R}^{n_{x,k}},\ r \in \mathbb{R}^{n_{u,k}},$$
$$A \in \mathbb{R}^{n_{x,k+1} \times n_{x,k}},\ B \in \mathbb{R}^{n_{x,k+1} \times n_{u,k}},\ b \in \mathbb{R}^{n_{x,k+1}},\ \underline{x}, \overline{x} \in \mathbb{R}^{x_k},\ \underline{u}, \overline{u} \in \mathbb{R}^{u_k},$$
$$C \in \mathbb{R}^{n_{c,k} \times n_{x,k}},\ D \in \mathbb{R}^{n_{c,k} \times n_{u,k}},\ \underline{d}, \overline{d} \in \mathbb{R}^{n_{c,k}} \quad (1)$$

Definition (stage size)

The k-th stage size is

$$\mathcal{N}_k = (n_{x,k}, n_{u,k}, n_{c,k}, n_{x,k+1}) \ .$$

QP Stage Sequence Operations

- An OCP QP can be seen as a collection of stages:

$$\mathcal{QP} = (\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_N)\ .$$

- Any stage subsequence $(\mathcal{S}_m, \mathcal{S}_{m+1}, \ldots, \mathcal{S}_n)$, $0 \leq m < n \leq N$ of a QP is also a QP.
- Note that A_N,B_N,b_N do not enter the minimization problem formulation. . .
- . . . but they are useful if you concatenate two QPs:

$$(\mathcal{QP}^{(1)}, \mathcal{QP}^{(2)}) = (\mathcal{S}_0^{(1)}, \mathcal{S}_1^{(1)}, \ldots, \mathcal{S}_N^{(1)}, \mathcal{S}_0^{(2)}, \mathcal{S}_1^{(2)}, \ldots, \mathcal{S}_N^{(2)})$$

  provided that the matrix sizes are consistent, i.e. the number of rows in $A_N^{(1)}$, $B_N^{(1)}$, $b_N^{(1)}$ is equal to $n_{x,0}^{(2)}$.
- By eliminating the equality constraints and intermediate state variables (condensing), a new QP can be obtained, which consists of a single stage of size

$$\left(n_{x,0}, \sum_{k=0}^{N} n_{u,k}, \sum_{k=0}^{N} n_{d,k}, n_{+,N} \right)\ .$$

Constructing and Initializing a QP Stage

```
// Declare matrix math kernel type
using Kernel = BlazeKernel<double>;

// Construct a QpStage object with specified dimensions
QpStage<Kernel> stage {QpSize {3, 2, 0}, 0};

// Fill the values
stage
.Q({
    {1., 0., 0.},
    {0., 2., 0.},
    {0., 0., 3.}
})
.R({
    {5., 0.},
    {0., 6.}
})
.S({
    {7., 8.},
    {9., 10.},
    {11., 12.}
})
.q({13., 14., 15.})
.r({16., 17.});        // ... A, B, b, lx, lu, ux, uu and so on
```

Constructing a Multiple Stage OCP QP

```cpp
// An alias for QpStage<Kernel>
using Stage = QpStage<Kernel>;

// Init stages
Stage stage0 = createStage0();
Stage stage1 = createStage1();
Stage stage2 = createStage2();

// An OCP QP is just a collection of QpStage
std::vector<Stage> qp;
qp.push_back(stage0);
qp.push_back(stage1);
qp.push_back(stage2);
```

Using C++ Standard Algorithms on QP Stage Sequences

- In C++, OCP QPs can be treated as stage iterator ranges.

Using C++ Standard Algorithms on QP Stage Sequences

- In C++, OCP QPs can be treated as stage iterator ranges.
- This allows applying standard algorithms (e.g. `std::copy`, `std::find_if`, `std::transform`) to OCP QPs.

Using C++ Standard Algorithms on QP Stage Sequences

- In C++, OCP QPs can be treated as stage iterator ranges.
- This allows applying standard algorithms (e.g. `std::copy`, `std::find_if`, `std::transform`) to OCP QPs.

Example 1: print all stages of a QP

```
std::copy(qp.begin(), qp.end(), std::ostream_iterator<Stage>(std::cout, "\n"));
```

Using C++ Standard Algorithms on QP Stage Sequences

- In C++, OCP QPs can be treated as stage iterator ranges.
- This allows applying standard algorithms (e.g. `std::copy`, `std::find_if`, `std::transform`) to OCP QPs.

Example 1: print all stages of a QP

```
std::copy(qp.begin(), qp.end(), std::ostream_iterator<Stage>(std::cout, "\n"));
```

Example 2: find a stage with a Hessian which is not positive-definite

```
auto bad_stage = std::find_if(qp.begin(), qp.end(),
  [] (Stage const& s) { return !s.isPositiveDefinite(); });
```

More Fancy Operations: Gauss-Newton Approximation

- Consider the Gauss-Newton cost Hessian approximation of a quadratic cost function

$$H = \begin{bmatrix} J_{yx} & J_{yu} \end{bmatrix}^\top \begin{bmatrix} J_{yx} & J_{yu} \end{bmatrix} = \begin{bmatrix} J_{yx}^\top J_{yx} & J_{yx}^\top J_{yu} \\ J_{yu}^\top J_{yx} & J_{yu}^\top J_{yu} \end{bmatrix} = \begin{bmatrix} Q & S \\ S^\top & R \end{bmatrix}$$

and the cost gradient

$$g = \begin{bmatrix} J_{yx} & J_{yu} \end{bmatrix}^\top y = \begin{bmatrix} J_{yx}^\top y \\ J_{yu}^\top y \end{bmatrix} = \begin{bmatrix} q \\ r \end{bmatrix} ,$$

where $J_{yx} = \frac{\mathrm{d}y}{\mathrm{d}x}$, $J_{yu} = \frac{\mathrm{d}y}{\mathrm{d}u}$ and y is the residual.

More Fancy Operations: Gauss-Newton Approximation

- Consider the Gauss-Newton cost Hessian approximation of a quadratic cost function

$$H = \begin{bmatrix} J_{yx} & J_{yu} \end{bmatrix}^\top \begin{bmatrix} J_{yx} & J_{yu} \end{bmatrix} = \begin{bmatrix} J_{yx}^\top J_{yx} & J_{yx}^\top J_{yu} \\ J_{yu}^\top J_{yx} & J_{yu}^\top J_{yu} \end{bmatrix} = \begin{bmatrix} Q & S \\ S^\top & R \end{bmatrix}$$

  and the cost gradient

$$g = \begin{bmatrix} J_{yx} & J_{yu} \end{bmatrix}^\top y = \begin{bmatrix} J_{yx}^\top y \\ J_{yu}^\top y \end{bmatrix} = \begin{bmatrix} q \\ r \end{bmatrix} ,$$

  where $J_{yx} = \frac{\mathrm{d}y}{\mathrm{d}x}$, $J_{yu} = \frac{\mathrm{d}y}{\mathrm{d}u}$ and y is the residual.
- This corresponds to setting elements of a QP stage like following:

procedure $\textsc{GaussNewtonCostApproximation}(y, J_{yx}, J_{yu})$

$Q \leftarrow J_{yx}^\top J_{yx}$

$R \leftarrow J_{yu}^\top J_{yu}$

$S \leftarrow J_{yx}^\top J_{yu}$

$q \leftarrow J_{yx}^\top y$

$r \leftarrow J_{yu}^\top y$

end procedure

More Fancy Operations: Linearized Shooting Equality

- Consider the shooting constraint of the form

$$x_{k+1} = f(x_k, u_k)$$

and its linearized version

$$\Delta x_{k+1} = \underbrace{\frac{\mathrm{d}f}{\mathrm{d}x}(x_k, u_k)}_{A_k} \Delta x_k + \underbrace{\frac{\mathrm{d}f}{\mathrm{d}u}(x_k, u_k)}_{B_k} \Delta u_k + \underbrace{f(x_k, u_k) - x_{k+1}}_{b_k}$$

More Fancy Operations: Linearized Shooting Equality

- Consider the shooting constraint of the form

$$x_{k+1} = f(x_k, u_k)$$

and its linearized version

$$\Delta x_{k+1} = \underbrace{\frac{\mathrm{d}f}{\mathrm{d}x}(x_k, u_k)}_{A_k} \Delta x_k + \underbrace{\frac{\mathrm{d}f}{\mathrm{d}u}(x_k, u_k)}_{B_k} \Delta u_k + \underbrace{f(x_k, u_k) - x_{k+1}}_{b_k}$$

- **procedure** $\textsc{LinearizedShootingEquality}(f, J_{fx}, J_{fu}, x^+)$

  $A \leftarrow J_{fx}$

  $B \leftarrow J_{fu}$

  $b \leftarrow f - x^+$

  end procedure

More Fancy Operations: Linearized General Constraints

- Consider general constraints of the form

$$\underline{g} \leq g(x_k, u_k) \leq \overline{g}$$

and its linearized version

$$\underbrace{\underline{g} - g(x_k, u_k)}_{\underline{d}_k} \leq \underbrace{\frac{\mathrm{d}g}{\mathrm{d}x}(x_k, u_k)}_{C_k} \Delta x_k + \underbrace{\frac{\mathrm{d}g}{\mathrm{d}u}(x_k, u_k)}_{D_k} \Delta u_k \leq \underbrace{\overline{g} - g(x_k, u_k)}_{\overline{d}_k}$$

More Fancy Operations: Linearized General Constraints

- Consider general constraints of the form

$$\underline{g} \leq g(x_k, u_k) \leq \overline{g}$$

and its linearized version

$$\underbrace{\underline{g} - g(x_k, u_k)}_{\underline{d}_k} \leq \underbrace{\frac{\mathrm{d}g}{\mathrm{d}x}(x_k, u_k)}_{C_k} \Delta x_k + \underbrace{\frac{\mathrm{d}g}{\mathrm{d}u}(x_k, u_k)}_{D_k} \Delta u_k \leq \underbrace{\overline{g} - g(x_k, u_k)}_{\overline{d}_k}$$

- **procedure** $\textsc{LinearizedGeneralConstraints}(g, J_{gx}, J_{gu}, \underline{g}, \overline{g})$
  - $C \leftarrow J_{gx}$
  - $D \leftarrow J_{gu}$
  - $\underline{d} \leftarrow \underline{g} - g$
  - $\overline{d} \leftarrow \overline{g} - g$

  end procedure

More Fancy Operations: Initial Value Embedding

- Consider the initial value constraint

$$x_0 = \tilde{x}_0 \Leftrightarrow \Delta x_0 = \tilde{x}_0 - x_0 \ .$$

More Fancy Operations: Initial Value Embedding

- Consider the initial value constraint

$$x_0 = \tilde{x}_0 \Leftrightarrow \Delta x_0 = \tilde{x}_0 - x_0 \ .$$

- Substituting it into the linearized shooting equality and the linearized general equalities gives

$$\Delta x_1 = \underbrace{\frac{\mathrm{d}f}{\mathrm{d}u}(x_0, u_0)}_{B_0} \Delta u_0 + \underbrace{f(x_0, u_0) - x_1 + \frac{\mathrm{d}f}{\mathrm{d}x}(x_0, u_0)(\tilde{x}_0 - x_0)}_{b_0}$$

$$\underbrace{\underline{g} - g(x_0, u_0) - \frac{\mathrm{d}g}{\mathrm{d}x}(x_0, u_0)(\tilde{x}_0 - x_0)}_{\underline{d}_0} \leq \underbrace{\frac{\mathrm{d}g}{\mathrm{d}u}(x_0, u_0)}_{D_0} \Delta u_0$$

$$\leq \underbrace{\overline{g} - g(x_0, u_0) - \frac{\mathrm{d}g}{\mathrm{d}x}(x_0, u_0)(\tilde{x}_0 - x_0)}_{\overline{d}_0}$$

More Fancy Operations: Initial Value Embedding

procedure $\textsc{InitialValueEmbedding}(\tilde{x}_0, x_0, x_1, f, J_{fx}, J_{fu}, g, J_{gx}, J_{gu}, g_l, g_u)$

Require: $n_{x,0} = 0$

$B_0 \leftarrow J_{fu}$

$b_0 \leftarrow f - x_1 + J_{fx}(\tilde{x}_0 - x_0)$

$D_0 \leftarrow J_{gu}$

$\underline{d}_0 \leftarrow g_l - g - J_{gx}(\tilde{x}_0 - x_0)$

$\overline{d}_0 \leftarrow g_u - g - J_{gx}(\tilde{x}_0 - x_0)$

end procedure

More Fancy Operations: Relative Bounds

- Consider the bound constraints

$$\begin{bmatrix} \underline{\underline{x}}_k \\ \underline{\underline{u}}_k \end{bmatrix} \leq \begin{bmatrix} x_k \\ u_k \end{bmatrix} \leq \begin{bmatrix} \overline{\overline{x}}_k \\ \overline{\overline{u}}_k \end{bmatrix}$$

which in the case of SQP transforms to

$$\underbrace{\begin{bmatrix} \underline{\underline{x}}_k - x_k \\ \underline{\underline{u}}_k - u_k \end{bmatrix}}_{\begin{bmatrix} \underline{x}_k \\ \underline{u}_k \end{bmatrix}} \leq \begin{bmatrix} \Delta x_k \\ \Delta u_k \end{bmatrix} \leq \underbrace{\begin{bmatrix} \overline{\overline{x}}_k - x_k \\ \overline{\overline{u}}_k - u_k \end{bmatrix}}_{\begin{bmatrix} \overline{x}_k \\ \overline{u}_k \end{bmatrix}}$$

More Fancy Operations: Relative Bounds

- Consider the bound constraints

$$\begin{bmatrix} \underline{\underline{x}}_k \\ \underline{\underline{u}}_k \end{bmatrix} \leq \begin{bmatrix} x_k \\ u_k \end{bmatrix} \leq \begin{bmatrix} \overline{\overline{x}}_k \\ \overline{\overline{u}}_k \end{bmatrix}$$

  which in the case of SQP transforms to

$$\underbrace{\begin{bmatrix} \underline{\underline{x}}_k - x_k \\ \underline{\underline{u}}_k - u_k \end{bmatrix}}_{\begin{bmatrix} \underline{x}_k \\ \underline{u}_k \end{bmatrix}} \leq \begin{bmatrix} \Delta x_k \\ \Delta u_k \end{bmatrix} \leq \underbrace{\begin{bmatrix} \overline{\overline{x}}_k - x_k \\ \overline{\overline{u}}_k - u_k \end{bmatrix}}_{\begin{bmatrix} \overline{x}_k \\ \overline{u}_k \end{bmatrix}}$$

- **procedure** $\textsc{RelativeBounds}(x, u, x_l, u_l, x_u, u_u)$

  $\underline{x} \leftarrow x_l - x$

  $\underline{u} \leftarrow u_l - u$

  $\overline{x} \leftarrow x_u - x$

  $\overline{u} \leftarrow u_u - u$

  end procedure

Code Example: Setting Up an OCP QP Stage

```
// Alias for the matrix math kernel
using K = BlazeKernel<double>;

// Construct the QpStage
QpStage<K> stage { QpSize {NX, NU, 0}, NX };

// Variables
extern K::StaticVector lx, ux;      // absolute state bounds
extern K::StaticVector lu, uu;      // absolute control bounds
K::StaticVector<NX> x, x_plus, f;  // current state, next state, next calculated state
K::StaticVector<NU> u;              // current input
K::StaticVector<NY> y;              // residual (the cost function is y^T * y)
K::StaticMatrix<NX, NX> df_dx;      // sensitivities
K::StaticMatrix<NX, NU> df_du;
K::StaticMatrix<NY, NX> dy_dx;
K::StaticMatrix<NY, NU> dy_du;

// Set x and u, calculate x_plus, y and the corresponding sensitivities:
//
// ...

// Set up the QP stage. Isn't it expressive?
stage.gaussNewtonCostApproximation(y, dy_dx, dy_du);
stage.linearizedShootingEquality(f, df_dx, df_du, x_plus);
stage.relativeBounds(x, u, lx, lu, ux, uu);
```

Solving an OCP QP: the Holy Trinity

- This is all nice, but what about solving a QP?

Solving an OCP QP: the Holy Trinity

- This is all nice, but what about solving a QP?
- Well, you need two more entities: a solution and a solver.

Solving an OCP QP: the Holy Trinity

- This is all nice, but what about solving a QP?
- Well, you need two more entities: a solution and a solver.
- External solvers (e.g. `qpOASES`, `HPMPC`, `HPIPM`) have different requirements on how the problem and solution data should be organized.

Solving an OCP QP: the Holy Trinity

- This is all nice, but what about solving a QP?
- Well, you need two more entities: a solution and a solver.
- External solvers (e.g. `qpOASES`, `HPMPC`, `HPIPM`) have different requirements on how the problem and solution data should be organized.
- The dimensions of a problem, a solution and a solver must match.

Solving an OCP QP: the Holy Trinity

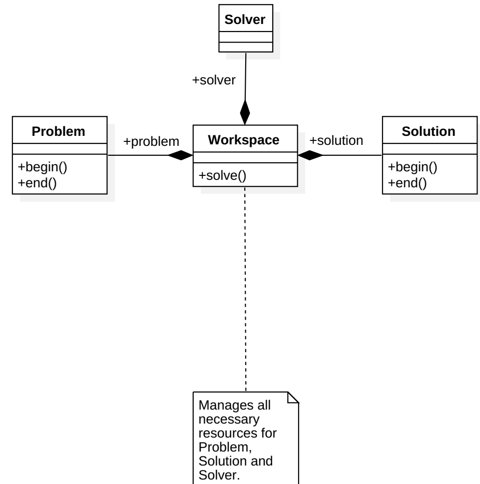
- This is all nice, but what about solving a QP?
- Well, you need two more entities: a solution and a solver.
- External solvers (e.g. `qpOASES`, `HPMPC`, `HPIPM`) have different requirements on how the problem and solution data should be organized.
- The dimensions of a problem, a solution and a solver must match.
- This creates tight coupling between a problem, a solution and a solver, making a new entity called QP Workspace.

Solving an OCP QP: the Holy Trinity

- This is all nice, but what about solving a QP?
- Well, you need two more entities: a solution and a solver.
- External solvers (e.g. qpOASES, HPMPC, HPIPM) have different requirements on how the problem and solution data should be organized.
- The dimensions of a problem, a solution and a solver must match.
- This creates tight coupling between a problem, a solution and a solver, making a new entity called QP Workspace.

Solving an OCP QP: the Holy Trinity

- This is all nice, but what about solving a QP?
- Well, you need two more entities: a solution and a solver.
- External solvers (e.g. `qpOASES`, `HPMPC`, `HPIPM`) have different requirements on how the problem and solution data should be organized.
- The dimensions of a problem, a solution and a solver must match.
- This creates tight coupling between a problem, a solution and a solver, making a new entity called QP Workspace.



Code Example: Solving a QP

```
// A type to use for real numbers
using Real = double;
// Alias for Workspace. Want qpOASES? Just change to QpOasesWorkspace!
using Workspace = HpmpcWorkspace<Real>;

// Create a QP Workspace for 2 stages with specified dimensions.
// All the data structures for the solver will be created here.
Workspace workspace {QpSize {3, 0, 0}, QpSize {0, 0, 0}};
// Reference to stage 0. It is not a QpStage object, although it has the same interface.
// Modifiers will write directly to solver data structures - no memory overhead!
auto& stage0 = workspace.problem()[0];

// Set cost
stage0.gaussNewtonCostApproximation(
    DynamicVector<Real> {1., 2., 42.},
    IdentityMatrix<Real> {3u},
    DynamicMatrix<Real> {3u, 0u}
);
// Set bounds
stage0.bounds(-infinity<Real>(), -infinity<Real>(), infinity<Real>(), infinity<Real>());

// Solve the problem
workspace.solve();
// Output the solution
std::cout << workspace.solution()[0].x() << std::endl;
```

Code Example: Solving a QP

What will it print out?

```
// A type to use for real numbers
using Real = double;
// Alias for Workspace. Want qpOASES? Just change to QpOasesWorkspace!
using Workspace = HpmpcWorkspace<Real>;

// Create a QP Workspace for 2 stages with specified dimensions.
// All the data structures for the solver will be created here.
Workspace workspace {QpSize {3, 0, 0}, QpSize {0, 0, 0}};
// Reference to stage 0. It is not a QpStage object, although it has the same interface.
// Modifiers will write directly to solver data structures - no memory overhead!
auto& stage0 = workspace.problem()[0];

// Set cost
stage0.gaussNewtonCostApproximation(
    DynamicVector<Real> {1., 2., 42.},
    IdentityMatrix<Real> {3u},
    DynamicMatrix<Real> {3u, 0u}
);
// Set bounds
stage0.bounds(-infinity<Real>(), -infinity<Real>(), infinity<Real>(), infinity<Real>());

// Solve the problem
workspace.solve();
// Output the solution
std::cout << workspace.solution()[0].x() << std::endl;
```

Code Example: Solving a QP

```
// A type to use for real numbers
using Real = double;
// Alias for Workspace. Want qpOASES? Just change to QpOasesWorkspace!
using Workspace = HpmpcWorkspace<Real>;

// Create a QP Workspace for 2 stages with specified dimensions.
// All the data structures for the solver will be created here.
Workspace workspace {QpSize {3, 0, 0}, QpSize {0, 0, 0}};
// Reference to stage 0. It is not a QpStage object, although it has the same interface.
// Modifiers will write directly to solver data structures - no memory overhead!
auto& stage0 = workspace.problem()[0];

// Set cost
stage0.gaussNewtonCostApproximation(
    DynamicVector<Real> {1., 2., 42.},
    IdentityMatrix<Real> {3u},
    DynamicMatrix<Real> {3u, 0u}
);
// Set bounds
stage0.bounds(-infinity<Real>(), -infinity<Real>(), infinity<Real>(), infinity<Real>());

// Solve the problem
workspace.solve();
// Output the solution
std::cout << workspace.solution()[0].x() << std::endl;
```

What will it print out?

```
 -1
 -2
-42
```

Summary

- OCP QP is represented as an ordered collection of objects called stages.

Summary

- OCP QP is represented as an ordered collection of objects called stages.
  - OCP QPs can be represented as C++ iterator ranges.

Summary

- OCP QP is represented as an ordered collection of objects called stages.
  - OCP QPs can be represented as C++ iterator ranges.
  - Algorithms from the C++ standard library (e.g. `std::copy`, `std::find_if`, `std::transform`) can be applied to OCP QPs.

Summary

- OCP QP is represented as an ordered collection of objects called stages.
  - OCP QPs can be represented as C++ iterator ranges.
  - Algorithms from the C++ standard library (e.g. `std::copy`, `std::find_if`, `std::transform`) can be applied to OCP QPs.
  - Can this approach be extended to scenario trees?

Summary

- OCP QP is represented as an ordered collection of objects called stages.
  - OCP QPs can be represented as C++ iterator ranges.
  - Algorithms from the C++ standard library (e.g. `std::copy`, `std::find_if`, `std::transform`) can be applied to OCP QPs.
  - Can this approach be extended to scenario trees?

Summary

- OCP QP is represented as an ordered collection of objects called stages.
  - OCP QPs can be represented as C++ iterator ranges.
  - Algorithms from the C++ standard library (e.g. `std::copy`, `std::find_if`, `std::transform`) can be applied to OCP QPs.
  - Can this approach be extended to scenario trees?

  The same applies to solution.
- Problem formulation is separated from its solving. A problem can be

Summary

- OCP QP is represented as an ordered collection of objects called stages.
  - OCP QPs can be represented as C++ iterator ranges.
  - Algorithms from the C++ standard library (e.g. `std::copy`, `std::find_if`, `std::transform`) can be applied to OCP QPs.
  - Can this approach be extended to scenario trees?

  The same applies to solution.
- Problem formulation is separated from its solving. A problem can be
  - ...set up in one place and solved in other place;

Summary

- OCP QP is represented as an ordered collection of objects called stages.
  - OCP QPs can be represented as C++ iterator ranges.
  - Algorithms from the C++ standard library (e.g. `std::copy`, `std::find_if`, `std::transform`) can be applied to OCP QPs.
  - Can this approach be extended to scenario trees?

  The same applies to solution.
- Problem formulation is separated from its solving. A problem can be
  - ...set up in one place and solved in other place;
  - ...saved, loaded or copied.

Summary

- OCP QP is represented as an ordered collection of objects called stages.
  - OCP QPs can be represented as C++ iterator ranges.
  - Algorithms from the C++ standard library (e.g. `std::copy`, `std::find_if`, `std::transform`) can be applied to OCP QPs.
  - Can this approach be extended to scenario trees?

  The same applies to solution.
- Problem formulation is separated from its solving. A problem can be
  - ...set up in one place and solved in other place;
  - ...saved, loaded or copied.
- The interfaces of problem, solver and solution are separated from their implementation.

Summary

- OCP QP is represented as an ordered collection of objects called stages.
  - OCP QPs can be represented as C++ iterator ranges.
  - Algorithms from the C++ standard library (e.g. `std::copy`, `std::find_if`, `std::transform`) can be applied to OCP QPs.
  - Can this approach be extended to scenario trees?

  The same applies to solution.
- Problem formulation is separated from its solving. A problem can be
  - ...set up in one place and solved in other place;
  - ...saved, loaded or copied.
- The interfaces of problem, solver and solution are separated from their implementation.
  - Allows different solvers to be used interchangeably.

Summary

- OCP QP is represented as an ordered collection of objects called stages.
  - OCP QPs can be represented as C++ iterator ranges.
  - Algorithms from the C++ standard library (e.g. `std::copy`, `std::find_if`, `std::transform`) can be applied to OCP QPs.
  - Can this approach be extended to scenario trees?

  The same applies to solution.
- Problem formulation is separated from its solving. A problem can be
  - ...set up in one place and solved in other place;
  - ...saved, loaded or copied.
- The interfaces of problem, solver and solution are separated from their implementation.
  - Allows different solvers to be used interchangeably.
- The matrix math interface is separated from its implementation.

Summary

- OCP QP is represented as an ordered collection of objects called stages.
  - OCP QPs can be represented as C++ iterator ranges.
  - Algorithms from the C++ standard library (e.g. `std::copy`, `std::find_if`, `std::transform`) can be applied to OCP QPs.
  - Can this approach be extended to scenario trees?

  The same applies to solution.
- Problem formulation is separated from its solving. A problem can be
  - ...set up in one place and solved in other place;
  - ...saved, loaded or copied.
- The interfaces of problem, solver and solution are separated from their implementation.
  - Allows different solvers to be used interchangeably.
- The matrix math interface is separated from its implementation.
  - One can switch between different implementations of matrix math (single precision, double precision, `Eigen3`, `Blaze`, any custom).

Summary

- OCP QP is represented as an ordered collection of objects called stages.
  - OCP QPs can be represented as C++ iterator ranges.
  - Algorithms from the C++ standard library (e.g. `std::copy`, `std::find_if`, `std::transform`) can be applied to OCP QPs.
  - Can this approach be extended to scenario trees?

  The same applies to solution.
- Problem formulation is separated from its solving. A problem can be
  - ...set up in one place and solved in other place;
  - ...saved, loaded or copied.
- The interfaces of problem, solver and solution are separated from their implementation.
  - Allows different solvers to be used interchangeably.
- The matrix math interface is separated from its implementation.
  - One can switch between different implementations of matrix math (single precision, double precision, `Eigen3`, `Blaze`, any custom).
- Tight coupliing between problem, solver and solution is resolved by introducing workspaces.

Final Slide

Questions? Comments?