

DCA

Generated by Doxygen 1.9.1



<b>1 DCA - Differentiable Collision Avoidance</b>	<b>1</b>
1.1 Primitives	1
1.2 Example	1
1.3 API	1
1.4 Documentation	1
<b>2 Module Index</b>	<b>3</b>
2.1 Modules	3
<b>3 Namespace Index</b>	<b>5</b>
3.1 Namespace List	5
<b>4 Hierarchical Index</b>	<b>7</b>
4.1 Class Hierarchy	7
<b>5 Class Index</b>	<b>9</b>
5.1 Class List	9
<b>6 File Index</b>	<b>11</b>
6.1 File List	11
<b>7 Module Documentation</b>	<b>13</b>
7.1 Partial Derivatives of Rotation Matrices	13
7.1.1 Detailed Description	13
7.1.2 Function Documentation	14
7.1.2.1 ddR_0_0()	14
7.1.2.2 ddR_0_1()	14
7.1.2.3 ddR_0_2()	14
7.1.2.4 ddR_1_0()	15
7.1.2.5 ddR_1_1()	15
7.1.2.6 ddR_1_2()	15
7.1.2.7 ddR_2_0()	17
7.1.2.8 ddR_2_1()	17
7.1.2.9 ddR_2_2()	17
<b>8 Namespace Documentation</b>	<b>21</b>
8.1 DCA Namespace Reference	21
8.1.1 Typedef Documentation	22
8.1.1.1 primitive_t	22
8.1.1.2 uint	23
8.1.1.3 Vector0d	23
8.1.1.4 Vector1d	23
8.1.1.5 Vector6d	23
8.1.1.6 Vector12d	23
8.1.1.7 Matrix12d	23

8.1.1.8 pair_t	23
<b>9 Class Documentation</b>	<b>25</b>
9.1 DCA::API Class Reference	25
9.1.1 Detailed Description	25
9.1.2 Member Function Documentation	26
9.1.2.1 compute_D() [1/2]	26
9.1.2.2 compute_dDdS() [1/2]	26
9.1.2.3 compute_d2DdS2() [1/2]	26
9.1.2.4 compute_closest_points() [1/2]	27
9.1.2.5 compute_D() [2/2]	27
9.1.2.6 compute_dDdS() [2/2]	28
9.1.2.7 compute_d2DdS2() [2/2]	28
9.1.2.8 compute_closest_points() [2/2]	29
9.1.2.9 compute_t()	29
9.2 DCA::Box Class Reference	30
9.2.1 Detailed Description	31
9.2.2 Constructor & Destructor Documentation	31
9.2.2.1 Box()	31
9.2.2.2 ~Box()	31
9.2.3 Member Function Documentation	32
9.2.3.1 compute_P()	32
9.2.3.2 compute_dPdS()	32
9.2.3.3 compute_dPdT()	33
9.2.3.4 compute_d2PdS2()	33
9.2.3.5 compute_d2PdSdT()	34
9.2.3.6 get_s()	34
9.2.3.7 SIZE_T()	34
9.2.3.8 get_largest_dimension_from_center()	35
9.2.3.9 getLocalVectors()	35
9.2.4 Member Data Documentation	35
9.2.4.1 center	35
9.2.4.2 orientation	35
9.2.4.3 dimensions	36
9.3 DCA::Capsule Class Reference	36
9.3.1 Detailed Description	37
9.3.2 Constructor & Destructor Documentation	37
9.3.2.1 Capsule()	37
9.3.2.2 ~Capsule()	37
9.3.3 Member Function Documentation	37
9.3.3.1 compute_P()	38
9.3.3.2 compute_dPdS()	38

9.3.3.3 compute_dPdT()	38
9.3.3.4 compute_d2PdS2()	39
9.3.3.5 compute_d2PdSdT()	39
9.3.3.6 get_s()	40
9.3.3.7 SIZE_T()	40
9.3.3.8 get_largest_dimension_from_center()	41
9.3.4 Member Data Documentation	41
9.3.4.1 startPosition	41
9.3.4.2 endPosition	41
9.4 DCA::ExpCoords Class Reference	41
9.4.1 Detailed Description	42
9.4.2 Member Function Documentation	42
9.4.2.1 get_theta()	43
9.4.2.2 get_dThetadRi()	43
9.4.2.3 get_R()	43
9.4.2.4 get_dRi()	44
9.4.2.5 get_ddR_i_j()	44
9.4.2.6 get_w()	45
9.4.2.7 get_dwdr()	45
9.4.2.8 get_ddwdr_dr1()	46
9.4.2.9 get_ddwdr_dr2()	46
9.4.2.10 get_ddwdr_dr3()	47
9.4.2.11 getSkewSymmetricMatrix()	47
9.4.2.12 safeACOS()	47
9.4.2.13 safeDACOS()	48
9.4.3 Member Data Documentation	48
9.4.3.1 tol	48
9.4.3.2 _PI	48
9.5 DCA::FiniteDifference Class Reference	48
9.5.1 Detailed Description	49
9.6 DCA::Logger Class Reference	49
9.6.1 Detailed Description	49
9.6.2 Member Enumeration Documentation	49
9.6.2.1 PRINT_COLOR	49
9.6.3 Member Function Documentation	50
9.6.3.1 print()	50
9.7 DCA::NeighborsPairGenerator Class Reference	50
9.7.1 Detailed Description	51
9.7.2 Constructor & Destructor Documentation	51
9.7.2.1 NeighborsPairGenerator()	51
9.7.3 Member Function Documentation	51
9.7.3.1 generate() [1/2]	51

9.7.3.2 generate() [2/2]	52
9.7.3.3 estimate_distance()	52
9.7.4 Member Data Documentation	53
9.7.4.1 m_radius	53
9.8 DCA::NewtonOptimizer Class Reference	53
9.8.1 Detailed Description	54
9.8.2 Constructor & Destructor Documentation	54
9.8.2.1 NewtonOptimizer()	54
9.8.2.2 ~NewtonOptimizer()	54
9.8.3 Member Function Documentation	54
9.8.3.1 optimize()	55
9.8.4 Member Data Documentation	55
9.8.4.1 name	55
9.8.4.2 solverResidual	55
9.8.4.3 maxLineSearchIterations	55
9.8.4.4 lineSearchStartValue	56
9.8.4.5 useDynamicRegularization	56
9.8.4.6 printInfo	56
9.8.4.7 checkLinearSystemSolve	56
9.8.4.8 checkHessianRank	56
9.8.4.9 objectiveValue	56
9.8.4.10 t_tmp	57
9.8.4.11 searchDir	57
9.8.4.12 gradient	57
9.8.4.13 hessian	57
9.9 DCA::Objective Class Reference	57
9.9.1 Detailed Description	59
9.9.2 Member Enumeration Documentation	59
9.9.2.1 PRIMITIVE	59
9.9.3 Constructor & Destructor Documentation	60
9.9.3.1 Objective()	60
9.9.3.2 ~Objective()	60
9.9.4 Member Function Documentation	60
9.9.4.1 compute_Ps()	60
9.9.4.2 compute_D()	61
9.9.4.3 compute_pDpT()	61
9.9.4.4 compute_pDpS()	61
9.9.4.5 compute_p2DpT2()	62
9.9.4.6 compute_p2DpS2()	62
9.9.4.7 compute_p2DpTpS()	62
9.9.4.8 compute_O()	63
9.9.4.9 compute_pOpT()	63

9.9.4.10 compute_pOpS()	63
9.9.4.11 compute_p2OpT2()	64
9.9.4.12 compute_p2OpS2()	64
9.9.4.13 compute_p2OpTpS()	64
9.9.4.14 get_sizes_t()	65
9.9.4.15 get_total_size_t()	65
9.9.4.16 get_primitives_description()	65
9.9.4.17 get_s()	66
9.9.4.18 get_t()	66
9.9.4.19 check_inputs()	66
9.9.5 Member Data Documentation	67
9.9.5.1 regularizerWeight	67
9.9.5.2 constraintWeight	67
9.9.5.3 unilateralConstraint	67
9.9.5.4 primitive_A	67
9.9.5.5 primitive_B	68
9.10 DCA::PermutationPairGenerator Class Reference	68
9.10.1 Detailed Description	68
9.10.2 Member Function Documentation	68
9.10.2.1 generate()	68
9.11 DCA::Primitive Class Reference	69
9.11.1 Detailed Description	70
9.11.2 Constructor & Destructor Documentation	70
9.11.2.1 Primitive()	70
9.11.2.2 ~Primitive()	70
9.11.3 Member Function Documentation	71
9.11.3.1 compute_P()	71
9.11.3.2 compute_dPdS()	71
9.11.3.3 compute_dPdT()	72
9.11.3.4 compute_d2PdS2()	72
9.11.3.5 compute_d2PdSdT()	72
9.11.3.6 get_s()	73
9.11.3.7 SIZE_T()	73
9.11.3.8 get_center_point()	74
9.11.3.9 get_largest_dimension_from_center()	74
9.11.3.10 check_t()	74
9.11.4 Member Data Documentation	74
9.11.4.1 safetyMargin	75
9.12 DCA::Rectangle Class Reference	75
9.12.1 Detailed Description	76
9.12.2 Constructor & Destructor Documentation	76
9.12.2.1 Rectangle()	76

9.12.2.2 ~Rectangle()	77
9.12.3 Member Function Documentation	77
9.12.3.1 compute_P()	77
9.12.3.2 compute_dPdS()	77
9.12.3.3 compute_dPdT()	78
9.12.3.4 compute_d2PdS2()	78
9.12.3.5 compute_d2PdSdT()	79
9.12.3.6 get_s()	79
9.12.3.7 SIZE_T()	80
9.12.3.8 get_largest_dimension_from_center()	80
9.12.3.9 getLocalVectors()	80
9.12.4 Member Data Documentation	80
9.12.4.1 center	81
9.12.4.2 orientation	81
9.12.4.3 dimensions	81
9.13 DCA::SoftUnilateralConstraint Class Reference	81
9.13.1 Detailed Description	82
9.13.2 Constructor & Destructor Documentation	82
9.13.2.1 SoftUnilateralConstraint()	82
9.13.2.2 ~SoftUnilateralConstraint()	83
9.13.3 Member Function Documentation	83
9.13.3.1 setLimit()	83
9.13.3.2 setEpsilon()	83
9.13.3.3 setStiffness()	83
9.13.3.4 evaluate()	84
9.13.3.5 computeDerivative()	84
9.13.3.6 computeSecondDerivative()	84
9.13.4 Member Data Documentation	85
9.13.4.1 a1	85
9.13.4.2 b1	85
9.13.4.3 c1	85
9.13.4.4 a2	85
9.13.4.5 b2	85
9.13.4.6 c2	85
9.13.4.7 d2	86
9.13.4.8 epsilon	86
9.13.4.9 limit	86
9.14 DCA::Solver Class Reference	86
9.14.1 Detailed Description	87
9.14.2 Constructor & Destructor Documentation	87
9.14.2.1 Solver()	87
9.14.2.2 ~Solver()	87



9.14.3 Member Function Documentation	87
9.14.3.1 compute_t()	88
9.14.3.2 compute_dtds()	88
9.14.3.3 compute_D()	88
9.14.3.4 compute_dDdS()	89
9.14.3.5 compute_d2DdS2()	89
9.14.3.6 compute_closest_points()	89
9.14.3.7 are_t_and_s_in_sync()	90
9.14.3.8 get_s_from_primitives()	90
9.14.3.9 get_total_size_t()	90
9.14.4 Member Data Documentation	91
9.14.4.1 objective	91
9.14.4.2 optimizer	91
9.15 DCA::Sphere Class Reference	91
9.15.1 Detailed Description	92
9.15.2 Constructor & Destructor Documentation	92
9.15.2.1 Sphere()	92
9.15.2.2 ~Sphere()	92
9.15.3 Member Function Documentation	92
9.15.3.1 compute_P()	93
9.15.3.2 compute_dPdS()	93
9.15.3.3 compute_dPdT()	94
9.15.3.4 compute_d2PdS2()	94
9.15.3.5 compute_d2PdSdT()	95
9.15.3.6 get_s()	95
9.15.3.7 SIZE_T()	95
9.15.3.8 get_largest_dimension_from_center()	96
9.15.4 Member Data Documentation	96
9.15.4.1 position	96
<b>10 File Documentation</b>	<b>97</b>
10.1 API.h File Reference	97
10.2 ddR.h File Reference	97
10.3 ExpCoords.h File Reference	98
10.4 FiniteDifference.h File Reference	98
10.5 Logger.h File Reference	99
10.6 NewtonOptimizer.h File Reference	99
10.7 Objective.h File Reference	100
10.8 Pair.h File Reference	100
10.9 Primitives.h File Reference	100
10.10 README.md File Reference	101
10.11 SoftUnilateralConstraint.h File Reference	101

10.12 Solver.h File Reference . . . . .	101
10.13 Utils.h File Reference . . . . .	102
<b>Index</b>	<b>103</b>

# Chapter 1

## DCA - Differentiable Collision Avoidance

We implement differentiable collision avoidance using multiple shape primitives.

### 1.1 Primitives

The library supports the following primitives with the mentioned states and parameters:

- Sphere, where the state is the center of the sphere.
- Capsule, where the state is the two ends of the capsule, stacked.
- Rectangle, where the state is the center of mass and the orientation (expressed in exponential coordinates).
- Box, where the state is the center of mass and the orientation (expressed in exponential coordinates).

Furthermore, the size of the parameters is:

- 0 for a sphere
- 1 for a capsule
- 2 for a rectangle
- 3 for a box

### 1.2 Example

We refer the reader to the [example](#).

### 1.3 API

The API can be found inside the [API header](#). Generally speaking, one passes two primitives and gets back either the distance or the first or second derivative. If the states of the two primitives do not change, one can also pass the computed parameterization ( $\tau$ ) using the specialized functions. The parameterization can also be computed with a single API call and the two given primitives.

### 1.4 Documentation

The documentation can be downloaded [here](#).



## Chapter 2

# Module Index

### 2.1 Modules

Here is a list of all modules:

Partial Derivatives of Rotation Matrices . . . . .	13
--	----



## Chapter 3

# Namespace Index

### 3.1 Namespace List

Here is a list of all namespaces with brief descriptions:

DCA	.....	21
-----	-------	----





## Chapter 4

# Hierarchical Index

### 4.1 Class Hierarchy

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

DCA::API . . . . .	25
DCA::ExpCoords . . . . .	41
DCA::FiniteDifference . . . . .	48
DCA::Objective . . . . .	57
DCA::Primitive . . . . .	69
DCA::Box . . . . .	30
DCA::Capsule . . . . .	36
DCA::Rectangle . . . . .	75
DCA::Sphere . . . . .	91
DCA::Solver . . . . .	86
DCA::Logger . . . . .	49
DCA::NeighborsPairGenerator . . . . .	50
DCA::NewtonOptimizer . . . . .	53
DCA::PermutationPairGenerator . . . . .	68
DCA::SoftUnilateralConstraint . . . . .	81



## Chapter 5

# Class Index

### 5.1 Class List

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

<a href="#">DCA::API</a>	
Public API . . . . .	25
<a href="#">DCA::Box</a>	
Definition of a Box . . . . .	30
<a href="#">DCA::Capsule</a>	
Definition of a Capsule . . . . .	36
<a href="#">DCA::ExpCoords</a>	
Represents Rotations using exponential coordinates . . . . .	41
<a href="#">DCA::FiniteDifference</a>	
Helper for finite differences (to test derivatives) . . . . .	48
<a href="#">DCA::Logger</a>	
Logger class . . . . .	49
<a href="#">DCA::NeighborsPairGenerator</a>	
This generator computes the pairs which are in a certain threshold from each other. It does so by computing a single position for each primitive and selecting pairs based on the distance . .	50
<a href="#">DCA::NewtonOptimizer</a>	
Simply newton optimizer class . . . . .	53
<a href="#">DCA::Objective</a>	
The main objective which used for computing distances and their derivatives . . . . .	57
<a href="#">DCA::PermutationPairGenerator</a>	
This generator creates all possible permutations of pairs. This means, the amount of pairs created is $n^2/2$ , where n is the number of primitives. Pairs are not returned twice (0, 1) and (1, 0)	68
<a href="#">DCA::Primitive</a>	
Definition of a Primitive . . . . .	69
<a href="#">DCA::Rectangle</a>	
Definition of a Rectangle . . . . .	75
<a href="#">DCA::SoftUnilateralConstraint</a>	
This class is a soft constraint, meaning a constraint is softfied . . . . .	81
<a href="#">DCA::Solver</a>	
Wrapper around a newton optimizer and an objective . . . . .	86
<a href="#">DCA::Sphere</a>	
Definition of a Sphere . . . . .	91



## Chapter 6

# File Index

### 6.1 File List

Here is a list of all files with brief descriptions:

<a href="#">API.h</a>	97
<a href="#">ddR.h</a>	97
<a href="#">ExpCoords.h</a>	98
<a href="#">FiniteDifference.h</a>	98
<a href="#">Logger.h</a>	99
<a href="#">NewtonOptimizer.h</a>	99
<a href="#">Objective.h</a>	100
<a href="#">Pair.h</a>	100
<a href="#">Primitives.h</a>	100
<a href="#">SoftUnilateralConstraint.h</a>	101
<a href="#">Solver.h</a>	101
<a href="#">Utils.h</a>	102



## Chapter 7

# Module Documentation

### 7.1 Partial Derivatives of Rotation Matrices

Second derivatives of  $R(\theta)$ .

#### Functions

- Eigen::Matrix3d [DCA::ddR\\_0\\_0](#) (const Eigen::Vector3d &v)  
*Compute  $\frac{d^2 R}{d\theta_0 d\theta_0}$ .*
- Eigen::Matrix3d [DCA::ddR\\_0\\_1](#) (const Eigen::Vector3d &v)  
*Compute  $\frac{d^2 R}{d\theta_0 d\theta_1}$ .*
- Eigen::Matrix3d [DCA::ddR\\_0\\_2](#) (const Eigen::Vector3d &v)  
*Compute  $\frac{d^2 R}{d\theta_0 d\theta_2}$ .*
- Eigen::Matrix3d [DCA::ddR\\_1\\_0](#) (const Eigen::Vector3d &v)  
*Compute  $\frac{d^2 R}{d\theta_1 d\theta_0}$ .*
- Eigen::Matrix3d [DCA::ddR\\_1\\_1](#) (const Eigen::Vector3d &v)  
*Compute  $\frac{d^2 R}{d\theta_1 d\theta_1}$ .*
- Eigen::Matrix3d [DCA::ddR\\_1\\_2](#) (const Eigen::Vector3d &v)  
*Compute  $\frac{d^2 R}{d\theta_1 d\theta_2}$ .*
- Eigen::Matrix3d [DCA::ddR\\_2\\_0](#) (const Eigen::Vector3d &v)  
*Compute  $\frac{d^2 R}{d\theta_2 d\theta_0}$ .*
- Eigen::Matrix3d [DCA::ddR\\_2\\_1](#) (const Eigen::Vector3d &v)  
*Compute  $\frac{d^2 R}{d\theta_2 d\theta_1}$ .*
- Eigen::Matrix3d [DCA::ddR\\_2\\_2](#) (const Eigen::Vector3d &v)  
*Compute  $\frac{d^2 R}{d\theta_2 d\theta_2}$ .*

#### 7.1.1 Detailed Description

Second derivatives of  $R(\theta)$ .

$\theta$  represents the exponential map parameter vector of the rotation.

$\frac{dR}{d\theta_i}$  is the derivative of the rotation matrix with respect to the i-th exponential map parameter  $\frac{d^2 R}{d\theta_i d\theta_j}$  is the derivative of  $\frac{dR}{d\theta_i}$  with respect to the j-th exponential map parameter

## 7.1.2 Function Documentation

### 7.1.2.1 ddR\_0\_0()

```
Eigen::Matrix3d DCA::ddR_0_0 (
    const Eigen::Vector3d & v ) [inline]
```

Compute  $\frac{d^2 R}{d\theta_0 d\theta_0}$ .

#### Parameters

in	v	$\theta$
----	---	----------

#### Returns

The partial second derivative.

### 7.1.2.2 ddR\_0\_1()

```
Eigen::Matrix3d DCA::ddR_0_1 (
    const Eigen::Vector3d & v ) [inline]
```

Compute  $\frac{d^2 R}{d\theta_0 d\theta_1}$ .

#### Parameters

in	v	$\theta$
----	---	----------

#### Returns

The partial second derivative.

### 7.1.2.3 ddR\_0\_2()

```
Eigen::Matrix3d DCA::ddR_0_2 (
    const Eigen::Vector3d & v ) [inline]
```

Compute  $\frac{d^2 R}{d\theta_0 d\theta_2}$ .



**Parameters**

in	$v$	$\theta$
----	-----	----------

**Returns**

The partial second derivative.

**7.1.2.4 ddR\_1\_0()**

```
Eigen::Matrix3d DCA::ddR_1_0 (
    const Eigen::Vector3d & v ) [inline]
```

Compute  $\frac{d^2 R}{d\theta_1 d\theta_0}$ .

**Parameters**

in	$v$	$\theta$
----	-----	----------

**Returns**

The partial second derivative.

**7.1.2.5 ddR\_1\_1()**

```
Eigen::Matrix3d DCA::ddR_1_1 (
    const Eigen::Vector3d & v ) [inline]
```

Compute  $\frac{d^2 R}{d\theta_1 d\theta_1}$ .

**Parameters**

in	$v$	$\theta$
----	-----	----------

**Returns**

The partial second derivative.

**7.1.2.6 ddR\_1\_2()**

```
Eigen::Matrix3d DCA::ddR_1_2 (
    const Eigen::Vector3d & v ) [inline]
```

Compute  $\frac{d^2 R}{d\theta_1 d\theta_2}$ .

**Parameters**

in	$v$	$\theta$
----	-----	----------

**Returns**

The partial second derivative.

**7.1.2.7 ddR\_2\_0()**

```
Eigen::Matrix3d DCA::ddR_2_0 (
    const Eigen::Vector3d & v ) [inline]
```

Compute  $\frac{d^2 R}{d\theta_2 d\theta_0}$ .

**Parameters**

in	$v$	$\theta$
----	-----	----------

**Returns**

The partial second derivative.

**7.1.2.8 ddR\_2\_1()**

```
Eigen::Matrix3d DCA::ddR_2_1 (
    const Eigen::Vector3d & v ) [inline]
```

Compute  $\frac{d^2 R}{d\theta_2 d\theta_1}$ .

**Parameters**

in	$v$	$\theta$
----	-----	----------

**Returns**

The partial second derivative.

**7.1.2.9 ddR\_2\_2()**

```
Eigen::Matrix3d DCA::ddR_2_2 (
    const Eigen::Vector3d & v ) [inline]
```

Compute  $\frac{d^2 R}{d\theta_2 d\theta_2}$ .

**Parameters**

$\text{in}$	$v$	$\theta$
-------------	-----	----------

**Returns**

The partial second derivative.



## Chapter 8

# Namespace Documentation

### 8.1 DCA Namespace Reference

#### Classes

- class [API](#)  
*Public API.*
- class [ExpCoords](#)  
*Represents Rotations using exponential coordinates.*
- class [FiniteDifference](#)  
*Helper for finite differences (to test derivatives).*
- class [Logger](#)  
*Logger class.*
- class [NewtonOptimizer](#)  
*Simply newton optimizer class.*
- class [Objective](#)  
*The main objective which used for computing distances and their derivatives.*
- class [PermutationPairGenerator](#)  
*This generator creates all possible permutations of pairs. This means, the amount of pairs created is  $n^2/2$ , where  $n$  is the number of primitives. Pairs are not returned twice (0, 1) and (1, 0).*
- class [NeighborsPairGenerator](#)  
*This generator computes the pairs which are in a certain threshold from each other. It does so by computing a single position for each primitive and selecting pairs based on the distance.*
- class [Primitive](#)  
*Definition of a Primitive.*
- class [Sphere](#)  
*Definition of a Sphere.*
- class [Capsule](#)  
*Definition of a Capsule.*
- class [Rectangle](#)  
*Definition of a Rectangle.*
- class [Box](#)  
*Definition of a Box.*
- class [SoftUnilateralConstraint](#)  
*This class is a soft constraint, meaning a constraint is softfied.*
- class [Solver](#)  
*Wrapper around a newton optimizer and an objective.*

## Typedefs

- using `primitive_t` = `std::variant< Sphere, Capsule, Rectangle, Box >`  
*All possible primitives.*
- typedef unsigned int `uint`  
*Easier access to a unsigned int.*
- using `Vector0d` = `Eigen::Matrix< double, 0, 1 >`  
*Easier access to a 0 vector of type double.*
- using `Vector1d` = `Eigen::Matrix< double, 1, 1 >`  
*Easier access to a 1 vector of type double.*
- using `Vector6d` = `Eigen::Matrix< double, 6, 1 >`  
*Easier access to a 6 vector of type double.*
- using `Vector12d` = `Eigen::Matrix< double, 12, 1 >`  
*Easier access to a 12 vector.*
- using `Matrix12d` = `Eigen::Matrix< double, 12, 12 >`  
*Easier access to a 12 by 12 matrix.*
- using `pair_t` = `std::pair< size_t, size_t >`  
*Easier access to a pair (corresponding of two indices)*

## Functions

- `Eigen::Matrix3d ddR_0_0` (const `Eigen::Vector3d` &`v`)  
*Compute  $\frac{d^2 R}{d\theta_0 d\theta_0}$ .*
- `Eigen::Matrix3d ddR_0_1` (const `Eigen::Vector3d` &`v`)  
*Compute  $\frac{d^2 R}{d\theta_0 d\theta_1}$ .*
- `Eigen::Matrix3d ddR_0_2` (const `Eigen::Vector3d` &`v`)  
*Compute  $\frac{d^2 R}{d\theta_0 d\theta_2}$ .*
- `Eigen::Matrix3d ddR_1_0` (const `Eigen::Vector3d` &`v`)  
*Compute  $\frac{d^2 R}{d\theta_1 d\theta_0}$ .*
- `Eigen::Matrix3d ddR_1_1` (const `Eigen::Vector3d` &`v`)  
*Compute  $\frac{d^2 R}{d\theta_1 d\theta_1}$ .*
- `Eigen::Matrix3d ddR_1_2` (const `Eigen::Vector3d` &`v`)  
*Compute  $\frac{d^2 R}{d\theta_1 d\theta_2}$ .*
- `Eigen::Matrix3d ddR_2_0` (const `Eigen::Vector3d` &`v`)  
*Compute  $\frac{d^2 R}{d\theta_2 d\theta_0}$ .*
- `Eigen::Matrix3d ddR_2_1` (const `Eigen::Vector3d` &`v`)  
*Compute  $\frac{d^2 R}{d\theta_2 d\theta_1}$ .*
- `Eigen::Matrix3d ddR_2_2` (const `Eigen::Vector3d` &`v`)  
*Compute  $\frac{d^2 R}{d\theta_2 d\theta_2}$ .*

### 8.1.1 Typedef Documentation

#### 8.1.1.1 `primitive_t`

```
using DCA::primitive_t = typedef std::variant<Sphere, Capsule, Rectangle, Box>
```

All possible primitives.



### 8.1.1.2 uint

```
typedef unsigned int DCA::uint
```

Easier access to a unsigned int.

### 8.1.1.3 Vector0d

```
using DCA::Vector0d = typedef Eigen::Matrix<double, 0, 1>
```

Easier access to a 0 vector of type double.

### 8.1.1.4 Vector1d

```
using DCA::Vector1d = typedef Eigen::Matrix<double, 1, 1>
```

Easier access to a 1 vector of type double.

### 8.1.1.5 Vector6d

```
using DCA::Vector6d = typedef Eigen::Matrix<double, 6, 1>
```

Easier access to a 6 vector of type double.

### 8.1.1.6 Vector12d

```
using DCA::Vector12d = typedef Eigen::Matrix<double, 12, 1>
```

Easier access to a 12 vector.

### 8.1.1.7 Matrix12d

```
using DCA::Matrix12d = typedef Eigen::Matrix<double, 12, 12>
```

Easier access to a 12 by 12 matrix.

### 8.1.1.8 pair\_t

```
using DCA::pair_t = typedef std::pair<size_t, size_t>
```

Easier access to a pair (corresponding of two indices)



## Chapter 9

# Class Documentation

### 9.1 DCA::API Class Reference

Public API.

```
#include <API.h>
```

#### Static Public Member Functions

- static double `compute_D` (const `primitive_t` &p\_a, const `primitive_t` &p\_b)  
*Computes the shortest distance between two primitives.*
- static void `compute_dDdS` (`Vector12d` &dDdS, const `primitive_t` &p\_a, const `primitive_t` &p\_b)  
*Computes the first derivative of the shortest distance.*
- static void `compute_d2DdS2` (`Matrix12d` &d2DdS2, const `primitive_t` &p\_a, const `primitive_t` &p\_b)  
*Computes the second derivative of the shortest distance.*
- static std::pair< `Vector3d`, `Vector3d` > `compute_closest_points` (const `primitive_t` &p\_a, const `primitive_t` &p\_b)  
*Compute the points which yield the shortest distance.*
- static double `compute_D` (const `primitive_t` &p\_a, const `primitive_t` &p\_b, const `VectorXd` &t)  
*Computes the shortest distance between two primitives given  $t$ .*
- static void `compute_dDdS` (`Vector12d` &dDdS, const `primitive_t` &p\_a, const `primitive_t` &p\_b, const `VectorXd` &t)  
*Computes the first derivative of the shortest distance given  $t$ .*
- static void `compute_d2DdS2` (`Matrix12d` &d2DdS2, const `primitive_t` &p\_a, const `primitive_t` &p\_b, const `VectorXd` &t)  
*Computes the second derivative of the shortest distance given  $t$ .*
- static std::pair< `Vector3d`, `Vector3d` > `compute_closest_points` (const `primitive_t` &p\_a, const `primitive_t` &p\_b, const `VectorXd` &t)  
*Compute the points which yield the shortest distance given  $t$ .*
- static void `compute_t` (`VectorXd` &t, const `primitive_t` &p\_a, const `primitive_t` &p\_b)  
*Compute the  $t$  values based on the primitives (and their state).*

#### 9.1.1 Detailed Description

Public API.

This is the main public API which should be used.

## 9.1.2 Member Function Documentation

### 9.1.2.1 compute\_D() [1/2]

```
static double DCA::API::compute_D (
    const primitive_t & p_a,
    const primitive_t & p_b ) [static]
```

Computes the shortest distance between two primitives.

#### Parameters

in	$p \leftarrow$ _a	The first primitive
in	$p \leftarrow$ _b	The second primitive

#### Returns

The shortest distance between both primitives.

### 9.1.2.2 compute\_dDdS() [1/2]

```
static void DCA::API::compute_dDdS (
    Vector12d & dDdS,
    const primitive_t & p_a,
    const primitive_t & p_b ) [static]
```

Computes the first derivative of the shortest distance.

Computes the first derivative of the shortest distance between two primitives with respect to the state of the primitives.

#### Parameters

out	dDdS	The first derivative $\frac{dD}{dS}$
in	p_a	The first primitive
in	p_b	The second primitive

### 9.1.2.3 compute\_d2DdS2() [1/2]

```
static void DCA::API::compute_d2DdS2 (
    Matrix12d & d2DdS2,
```

```
const primitive_t & p_a,
const primitive_t & p_b ) [static]
```

Computes the second derivative of the shortest distance.

Computes the second derivative of the shortest distance between two primitives with respect to the state of the primitives.

#### Parameters

out	$d^2D/dS^2$	The second derivative $\frac{d^2D}{dS^2}$
in	$p_a$	The first primitive
in	$p_b$	The second primitive

#### 9.1.2.4 compute\_closest\_points() [1/2]

```
static std::pair<Vector3d, Vector3d> DCA::API::compute_closest_points (
    const primitive_t & p_a,
    const primitive_t & p_b ) [static]
```

Compute the points which yield the shortest distance.

Returns a pair of points, where the distance between those points is the shortest between the primitives. The first returned point belongs to primitive  $p_a$ , the second belongs to primitive  $p_b$ .

#### Parameters

in	$p_a$	The first primitive
in	$p_b$	The second primitive

#### Returns

A pair of points, where the distance between those points is the shortest between the two primitives.

#### 9.1.2.5 compute\_D() [2/2]

```
static double DCA::API::compute_D (
    const primitive_t & p_a,
    const primitive_t & p_b,
    const VectorXd & t ) [static]
```

Computes the shortest distance between two primitives given  $t$ .

**Parameters**

in	$p \leftrightarrow$ _a	The first primitive
in	$p \leftrightarrow$ _b	The second primitive
in	$t$	The $t$ values for the parameterization.

**Returns**

The shortest distance between both primitives.

**9.1.2.6 compute\_dDdS() [2/2]**

```
static void DCA::API::compute_dDdS (
    Vector12d & dDdS,
    const primitive_t & p_a,
    const primitive_t & p_b,
    const VectorXd & t ) [static]
```

Computes the first derivative of the shortest distance given  $t$ .

Computes the first derivative of the shortest distance between two primitives with respect to the state of the primitives.

**Parameters**

out	$dDdS$	The first derivative $\frac{dD}{dS}$
in	$p_a$	The first primitive
in	$p_b$	The second primitive
in	$t$	The $t$ values for the parameterization.

**9.1.2.7 compute\_d2DdS2() [2/2]**

```
static void DCA::API::compute_d2DdS2 (
    Matrix12d & d2DdS2,
    const primitive_t & p_a,
    const primitive_t & p_b,
    const VectorXd & t ) [static]
```

Computes the second derivative of the shortest distance given  $t$ .

Computes the second derivative of the shortest distance between two primitives with respect to the state of the primitives.

## Parameters

out	$d^2D/dS^2$	The second derivative $\frac{d^2D}{dS^2}$
in	$p_a$	The first primitive
in	$p_b$	The second primitive
in	$t$	The $t$ values for the parameterization.

## 9.1.2.8 compute\_closest\_points() [2/2]

```
static std::pair<Vector3d, Vector3d> DCA::API::compute_closest_points (
    const primitive_t & p_a,
    const primitive_t & p_b,
    const VectorXd & t ) [static]
```

Compute the points which yield the shortest distance given  $t$ .

Returns a pair of points, where the distance between those points is the shortest between the primitives. The first returned point belongs to primitive  $p_a$ , the second belongs to primitive  $p_b$ .

## Parameters

in	$p_a$	The first primitive
in	$p_b$	The second primitive
in	$t$	The $t$ values for the parameterization.

## Returns

A pair of points, where the distance between those points is the shortest between the two primitives.

## 9.1.2.9 compute\_t()

```
static void DCA::API::compute_t (
    VectorXd & t,
    const primitive_t & p_a,
    const primitive_t & p_b ) [static]
```

Compute the  $t$  values based on the primitives (and their state).

Computes the  $t$  values based on the state of the primitives. The values represent the points which will yield the shortest distance between the two primitives.

## Parameters

out	$t$	The $t$ values which represent the parameterization of the shortest distance.
in	$p_a$	The first primitive
in	$p_b$	The second primitive

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

- [API.h](#)

## 9.2 DCA::Box Class Reference

Definition of a Box.

```
#include <Primitives.h>
```

### Public Member Functions

- [Box](#) (const [Vector3d](#) &[center](#), const [Matrix3d](#) &[orientation](#), const [Vector3d](#) &[dimensions](#), const double &[safetyMargin](#)=0.001)
- [~Box](#) ()=default  
*Default destructor.*
- [Vector3d compute\\_P](#) (const [Vector6d](#) &s, const [VectorXd](#) &t) const override  
*Compute a point on the primitive.*
- [Eigen::Matrix< double, 3, 6 > compute\\_dPdS](#) (const [Vector6d](#) &s, const [VectorXd](#) &t) const override  
*Compute the derivative of a point on the primitive with respect to s.*
- [Eigen::Matrix< double, 3, -1 > compute\\_dPdT](#) (const [Vector6d](#) &s, const [VectorXd](#) &t) const override  
*Compute the derivative of a point on the primitive with respect to t.*
- [std::array< Eigen::Matrix< double, 3, 6 >, 6 > compute\\_d2PdS2](#) (const [Vector6d](#) &s, const [VectorXd](#) &t) const override  
*Compute the second derivative of a point on the primitive with respect to s.*
- [std::vector< Eigen::Matrix< double, 3, 6 > > compute\\_d2PdSdT](#) (const [Vector6d](#) &s, const [VectorXd](#) &t) const override  
*Compute the mixed second derivative of a point on the primitive with respect to s and t.*
- [Vector6d get\\_s](#) () const override  
*Helper to get the state.*
- [int SIZE\\_T](#) () const override  
*Helper to get the size of t.*
- [double get\\_largest\\_dimension\\_from\\_center](#) () const override  
*Get the largest dimension.*

### Public Attributes

- [Vector3d center](#)  
*The center of mass of the box.*
- [Matrix3d orientation](#)  
*The orientation of the box.*
- [Vector3d dimensions](#)  
*Convention: local dimensions in x, y, z direction.*

### Private Member Functions

- [std::tuple< Vector3d, Vector3d, Vector3d > getLocalVectors](#) () const  
*Get vectors from the coordinates.*



## Additional Inherited Members

### 9.2.1 Detailed Description

Definition of a Box.

The state of a box is as follows: (x1, y1, z1, th1, th2, th3), which means that it is center of mass and the exponential map theta.

The parameterization of a box is as follows: (t1, t2, t3), which means, the parameterisation of a box.

### 9.2.2 Constructor & Destructor Documentation

#### 9.2.2.1 Box()

```
DCA::Box::Box (
    const Vector3d & center,
    const Matrix3d & orientation,
    const Vector3d & dimensions,
    const double & safetyMargin = 0.001 )
```

Create a box primitive.

##### Parameters

in	<i>center</i>	The center of mass.
in	<i>orientation</i>	The orientation of the box.
in	<i>dimensions</i>	The three dimensions of the box in x, y and z direction.
in	<i>safetyMargin</i>	The safety margin of the box.

##### Exceptions

<i>std::runtime_error</i>	If safety margin < 0.
<i>std::runtime_error</i>	If any dimension < 0.
<i>std::runtime_error</i>	If the orientation is invalid.

#### 9.2.2.2 ~Box()

```
DCA::Box::~Box ( ) [default]
```

Default destructor.

## 9.2.3 Member Function Documentation

### 9.2.3.1 compute\_P()

```
Vector3d DCA::Box::compute_P (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```

Compute a point on the primitive.

Compute a point on the primitive based on s and t.

#### Parameters

in	<i>s</i>	The state of the primitive.
in	<i>t</i>	The parameterization of the point.

#### Returns

The point.

Implements [DCA::Primitive](#).

### 9.2.3.2 compute\_dPdS()

```
Eigen::Matrix<double, 3, 6> DCA::Box::compute_dPdS (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```

Compute the derivative of a point on the primitive with respect to s.

Compute the derivative of a point on the primitive based on s and t.

#### Parameters

in	<i>s</i>	The state of the primitive.
in	<i>t</i>	The parameterization of the point.

#### Returns

The derivative  $\frac{dP}{ds}$ .

Implements [DCA::Primitive](#).

### 9.2.3.3 compute\_dPdT()

```
Eigen::Matrix<double, 3, -1> DCA::Box::compute_dPdT (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```

Compute the derivative of a point on the primitive with respect to t.

Compute the derivative of a point on the primitive based on s and t.

#### Parameters

in	<i>s</i>	The state of the primitive.
in	<i>t</i>	The parameterization of the point.

#### Returns

The derivative  $\frac{dP}{dt}$ .

In the case of a box, this has size 3x3.

Implements [DCA::Primitive](#).

### 9.2.3.4 compute\_d2PdS2()

```
std::array<Eigen::Matrix<double, 3, 6>, 6> DCA::Box::compute_d2PdS2 (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```

Compute the second derivative of a point on the primitive with respect to s.

Compute the second derivative of a point on the primitive based on s and t.

#### Parameters

in	<i>s</i>	The state of the primitive.
in	<i>t</i>	The parameterization of the point.

#### Returns

The second derivative  $\frac{d^2P}{ds^2}$  as an array (size 6).

Implements [DCA::Primitive](#).

### 9.2.3.5 compute\_d2PdSdT()

```
std::vector<Eigen::Matrix<double, 3, 6> > DCA::Box::compute_d2PdSdT (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```

Compute the mixed second derivative of a point on the primitive with respect to s and t.

Compute the mixed second derivative of a point on the primitive based on s and t.

#### Parameters

in	<i>s</i>	The state of the primitive.
in	<i>t</i>	The parameterization of the point.

#### Returns

The derivative  $\frac{d^2 P}{ds dt}$  as a vector (size t).

In the case of a box, this returns a vector with size 3.

Implements [DCA::Primitive](#).

### 9.2.3.6 get\_s()

```
Vector6d DCA::Box::get_s ( ) const [override], [virtual]
```

Helper to get the state.

#### Returns

The state of the primitive.

Implements [DCA::Primitive](#).

### 9.2.3.7 SIZE\_T()

```
int DCA::Box::SIZE_T ( ) const [override], [virtual]
```

Helper to get the size of t.

#### Returns

The number of dimensions needed for parameterization.

In the case of a rectangle, this returns 3.

Implements [DCA::Primitive](#).

### 9.2.3.8 get\_largest\_dimension\_from\_center()

```
double DCA::Box::get_largest_dimension_from_center ( ) const [override], [virtual]
```

Get the largest dimension.

Helper for pair generation.

#### Returns

The largest dimension from the center point.

Implements [DCA::Primitive](#).

### 9.2.3.9 getLocalVectors()

```
std::tuple<Vector3d, Vector3d, Vector3d> DCA::Box::getLocalVectors ( ) const [private]
```

Get vectors from the coordinates.

#### Returns

The local vectors.

## 9.2.4 Member Data Documentation

### 9.2.4.1 center

```
Vector3d DCA::Box::center
```

The center of mass of the box.

### 9.2.4.2 orientation

```
Matrix3d DCA::Box::orientation
```

The orientation of the box.

### 9.2.4.3 dimensions

Vector3d DCA::Box::dimensions

Convention: local dimensions in x, y, z direction.

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

- [Primitives.h](#)

## 9.3 DCA::Capsule Class Reference

Definition of a Capsule.

```
#include <Primitives.h>
```

### Public Member Functions

- [Capsule](#) (const Vector3d &startPosition, const Vector3d &endPosition, const double &radius)
- [~Capsule](#) ()=default  
*Default deconstructor.*
- Vector3d [compute\\_P](#) (const [Vector6d](#) &s, const VectorXd &t) const override  
*Compute a point on the primitive.*
- Eigen::Matrix< double, 3, 6 > [compute\\_dPdS](#) (const [Vector6d](#) &s, const VectorXd &t) const override  
*Compute the derivative of a point on the primitive with respect to s.*
- Eigen::Matrix< double, 3, -1 > [compute\\_dPdT](#) (const [Vector6d](#) &s, const VectorXd &t) const override  
*Compute the derivative of a point on the primitive with respect to t.*
- std::array< Eigen::Matrix< double, 3, 6 >, 6 > [compute\\_d2PdS2](#) (const [Vector6d](#) &s, const VectorXd &t) const override  
*Compute the second derivative of a point on the primitive with respect to s.*
- std::vector< Eigen::Matrix< double, 3, 6 > > [compute\\_d2PdSdT](#) (const [Vector6d](#) &s, const VectorXd &t) const override  
*Compute the mixed second derivative of a point on the primitive with respect to s and t.*
- [Vector6d get\\_s](#) () const override  
*Helper to get the state.*
- int [SIZE\\_T](#) () const override  
*Helper to get the size of t.*
- double [get\\_largest\\_dimension\\_from\\_center](#) () const override  
*Get the largest dimension.*

### Public Attributes

- Vector3d [startPosition](#)  
*The start position of the capsule in 3d space.*
- Vector3d [endPosition](#)  
*The end position of the capsule in 3d space.*

## Additional Inherited Members

### 9.3.1 Detailed Description

Definition of a Capsule.

The state of a capsule is as follows: (x1, y1, z1, x2, y2, z2), which means that it is the start and end point of the underlying line segment, stacked.

The parameterization of a capsule is as follows: (t1), which means, the position on the underlying line segment from -1 to 1.

### 9.3.2 Constructor & Destructor Documentation

#### 9.3.2.1 Capsule()

```
DCA::Capsule::Capsule (
    const Vector3d & startPosition,
    const Vector3d & endPosition,
    const double & radius )
```

Create a capsule primitive.

##### Parameters

in	<i>startPosition</i>	The start position of the underlying line segment.
in	<i>endPosition</i>	The end position of the underlying line segment.
in	<i>radius</i>	The safety margin around the center.

##### Exceptions

<i>std::runtime_error</i>	If the radius is smaller than the distance between start and end.
---------------------------	---

#### 9.3.2.2 ~Capsule()

```
DCA::Capsule::~~Capsule ( ) [default]
```

Default destructor.

### 9.3.3 Member Function Documentation

### 9.3.3.1 compute\_P()

```
Vector3d DCA::Capsule::compute_P (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```

Compute a point on the primitive.

Compute a point on the primitive based on s and t.

#### Parameters

in	<i>s</i>	The state of the primitive.
in	<i>t</i>	The parameterization of the point.

#### Returns

The point.

Implements [DCA::Primitive](#).

### 9.3.3.2 compute\_dPdS()

```
Eigen::Matrix<double, 3, 6> DCA::Capsule::compute_dPdS (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```

Compute the derivative of a point on the primitive with respect to s.

Compute the derivative of a point on the primitive based on s and t.

#### Parameters

in	<i>s</i>	The state of the primitive.
in	<i>t</i>	The parameterization of the point.

#### Returns

The derivative  $\frac{dP}{ds}$ .

Implements [DCA::Primitive](#).

### 9.3.3.3 compute\_dPdT()

```
Eigen::Matrix<double, 3, -1> DCA::Capsule::compute_dPdT (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```



Compute the derivative of a point on the primitive with respect to  $t$ .

Compute the derivative of a point on the primitive based on  $s$  and  $t$ .

#### Parameters

in	$s$	The state of the primitive.
in	$t$	The parameterization of the point.

#### Returns

The derivative  $\frac{dP}{dt}$ .

In the case of a capsule, this has size 3x1.

Implements [DCA::Primitive](#).

#### 9.3.3.4 compute\_d2PdS2()

```
std::array<Eigen::Matrix<double, 3, 6>, 6> DCA::Capsule::compute_d2PdS2 (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```

Compute the second derivative of a point on the primitive with respect to  $s$ .

Compute the second derivative of a point on the primitive based on  $s$  and  $t$ .

#### Parameters

in	$s$	The state of the primitive.
in	$t$	The parameterization of the point.

#### Returns

The second derivative  $\frac{d^2P}{ds^2}$  as an array (size 6).

Implements [DCA::Primitive](#).

#### 9.3.3.5 compute\_d2PdSdT()

```
std::vector<Eigen::Matrix<double, 3, 6> > DCA::Capsule::compute_d2PdSdT (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```

Compute the mixed second derivative of a point on the primitive with respect to  $s$  and  $t$ .

Compute the mixed second derivative of a point on the primitive based on  $s$  and  $t$ .

**Parameters**

in	<i>s</i>	The state of the primitive.
in	<i>t</i>	The parameterization of the point.

**Returns**

The derivative  $\frac{d^2 P}{ds dt}$  as a vector (size t).

In the case of a capsule, this returns a vector with one matrix.

Implements [DCA::Primitive](#).

**9.3.3.6 get\_s()**

```
Vector6d DCA::Capsule::get_s ( ) const [override], [virtual]
```

Helper to get the state.

**Returns**

The state of the primitive.

Implements [DCA::Primitive](#).

**9.3.3.7 SIZE\_T()**

```
int DCA::Capsule::SIZE_T ( ) const [override], [virtual]
```

Helper to get the size of t.

**Returns**

The number of dimensions needed for parameterization.

In the case of a capsule, this returns 1.

Implements [DCA::Primitive](#).

### 9.3.3.8 get\_largest\_dimension\_from\_center()

```
double DCA::Capsule::get_largest_dimension_from_center ( ) const [override], [virtual]
```

Get the largest dimension.

Helper for pair generation.

#### Returns

The largest dimension from the center point.

Implements [DCA::Primitive](#).

## 9.3.4 Member Data Documentation

### 9.3.4.1 startPosition

```
Vector3d DCA::Capsule::startPosition
```

The start position of the capsule in 3d space.

### 9.3.4.2 endPosition

```
Vector3d DCA::Capsule::endPosition
```

The end position of the capsule in 3d space.

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

- [Primitives.h](#)

## 9.4 DCA::ExpCoords Class Reference

Represents Rotations using exponential coordinates.

```
#include <ExpCoords.h>
```

## Static Public Member Functions

- static Vector3d [get\\_theta](#) (const Matrix3d &R)  
*Get  $\theta$  from a rotation matrix.*
- static Matrix3d [get\\_dThetadRi](#) (const Matrix3d &R, const int &index)  
*Get  $\frac{d\theta}{dR_i}$ .*
- static Matrix3d [get\\_R](#) (const Vector3d &theta)  
*Get a rotation matrix from given exponential map.*
- static Matrix3d [get\\_dRi](#) (const Vector3d &theta, int i)  
*Returns the Jacobian of R.*
- static Matrix3d [get\\_ddR\\_ij](#) (const Vector3d &theta, int i, int j)  
*Returns the derivative of the Jacobian of R at an index.*
- static Vector3d [get\\_w](#) (const Vector3d &theta, const Vector3d &x)  
*Returns world coordinates for a vector.*
- static Matrix3d [get\\_dwdr](#) (const Vector3d &theta, const Vector3d &x)  
*Returns the first derivative of [ExpCoords::get\\_w](#).*
- static Matrix3d [get\\_ddwdr\\_dr1](#) (const Vector3d &theta, const Vector3d &x)  
*Returns  $\frac{d^2 w}{dR dR_1}$ .*
- static Matrix3d [get\\_ddwdr\\_dr2](#) (const Vector3d &theta, const Vector3d &x)  
*Returns  $\frac{d^2 w}{dR dR_2}$ .*
- static Matrix3d [get\\_ddwdr\\_dr3](#) (const Vector3d &theta, const Vector3d &x)  
*Returns  $\frac{d^2 w}{dR dR_3}$ .*

## Static Private Member Functions

- static Matrix3d [getSkewSymmetricMatrix](#) (const Vector3d &v)  
*Get the skew symmetric matrix for a given vector.*
- static double [safeACOS](#) (double x)  
*Safe arcus cosinus.*
- static double [safeDACOS](#) (double x)  
*Safe derivative of arcus cosinus.*

## Static Private Attributes

- constexpr static const double [tol](#) = 1e-8  
*Tolerance for avoiding singularities of Rodrigues' formula.*
- constexpr static const double [\\_PI](#) = 3.14159265359  
*Helper.*

### 9.4.1 Detailed Description

Represents Rotations using exponential coordinates.

See e.g. <https://arxiv.org/pdf/1312.0788.pdf>

### 9.4.2 Member Function Documentation

### 9.4.2.1 get\_theta()

```
static Vector3d DCA::ExpCoords::get_theta (
    const Matrix3d & R ) [inline], [static]
```

Get  $\theta$  from a rotation matrix.

Returns the exponential map parameterization

#### Parameters

in	$R$	The rotation matrix
----	-----	---------------------

#### Returns

The exponential map parameterization.

### 9.4.2.2 get\_dThetadRi()

```
static Matrix3d DCA::ExpCoords::get_dThetadRi (
    const Matrix3d & R,
    const int & index ) [inline], [static]
```

Get  $\frac{d\theta}{dR_i}$ .

Returns the derivative of exponential coordinates with respect to a column of R

#### Parameters

in	$R$	The rotation matrix
in	$index$	The column to use. Must be either 0, 1 or 2.

#### Returns

The exponential map parameterization.

#### Exceptions

<code>std::runtime_error</code>	If the index is out of bounds.
---------------------------------	--------------------------------

### 9.4.2.3 get\_R()

```
static Matrix3d DCA::ExpCoords::get_R (
    const Vector3d & theta ) [inline], [static]
```

Get a rotation matrix from given exponential map.

**Parameters**

in	<i>theta</i>	$\theta$
----	--------------	----------

**Returns**

The rotation matrix which is represented by  $\theta$ .

**9.4.2.4 get\_dRi()**

```
static Matrix3d DCA::ExpCoords::get_dRi (
    const Vector3d & theta,
    int i ) [inline], [static]
```

Returns the Jacobian of R.

Returns a Jacobian that tells us how the rotation matrix changes with respect to the exponential map parameters that define it.

**Parameters**

in	<i>theta</i>	$\theta$
in	<i>i</i>	The index (column).

**Returns**

$$\frac{dR}{d\theta_i}$$

**9.4.2.5 get\_ddR\_i\_j()**

```
static Matrix3d DCA::ExpCoords::get_ddR_i_j (
    const Vector3d & theta,
    int i,
    int j ) [inline], [static]
```

Returns the derivative of the Jacobian of R at an index.

Returns the derivative of the Jacobian  $\frac{dR}{d\theta_i}$  with respect to the exponential map parameters at index j.

**Parameters**

in	<i>theta</i>	$\theta$
in	<i>i</i>	The first index.
in	<i>j</i>	The second index.

**Returns**

$$\frac{d^2 R}{d\theta_i d\theta_j}$$

**Exceptions**

<code>std::runtime_error</code>	(only release) if either i or j is not in [0,2].
---------------------------------	--

**9.4.2.6 get\_w()**

```
static Vector3d DCA::ExpCoords::get_w (
    const Vector3d & theta,
    const Vector3d & x ) [inline], [static]
```

Returns world coordinates for a vector.

Returns the world coordinates for the vector x that is expressed in local coordinates.

**Parameters**

in	<i>theta</i>	The exponential map representation.
in	<i>x</i>	The vector in local coordinates.

**Returns**

The world coordinates for x.

**9.4.2.7 get\_dwdr()**

```
static Matrix3d DCA::ExpCoords::get_dwdr (
    const Vector3d & theta,
    const Vector3d & x ) [inline], [static]
```

Returns the first derivative of [ExpCoords::get\\_w](#).

Returns a matrix that tells us how w changes with respect to the orientation.

**Parameters**

in	<i>theta</i>	The exponential map representation.
in	<i>x</i>	The vector in local coordinates.

**Returns**

The derivative of  $w$  with respect to the orientation.

**9.4.2.8 get\_ddwdr\_dr1()**

```
static Matrix3d DCA::ExpCoords::get_ddwdr_dr1 (
    const Vector3d & theta,
    const Vector3d & x ) [inline], [static]
```

Returns  $\frac{d^2 w}{dRdR_1}$ .

Returns the second derivative of [ExpCoords::get\\_w](#) with respect to the first component of the orientation.

**Parameters**

in	<i>theta</i>	The exponential map representation.
in	<i>x</i>	The local coordinates of the point.

**Returns**

The second derivative.

**9.4.2.9 get\_ddwdr\_dr2()**

```
static Matrix3d DCA::ExpCoords::get_ddwdr_dr2 (
    const Vector3d & theta,
    const Vector3d & x ) [inline], [static]
```

Returns  $\frac{d^2 w}{dRdR_2}$ .

Returns the second derivative of [ExpCoords::get\\_w](#) with respect to the second component of the orientation.

**Parameters**

in	<i>theta</i>	The exponential map representation.
in	<i>x</i>	The local coordinates of the point.

**Returns**

The second derivative.



**9.4.2.10 get\_ddwdr\_dr3()**

```
static Matrix3d DCA::ExpCoords::get_ddwdr_dr3 (
    const Vector3d & theta,
    const Vector3d & x ) [inline], [static]
```

Returns  $\frac{d^2 w}{dR dR_3}$ .

Returns the second derivative of [ExpCoords::get\\_w](#) with respect to the third component of the orientation.

**Parameters**

in	<i>theta</i>	The exponential map representation.
in	<i>x</i>	The local coordinates of the point.

**Returns**

The second derivative.

**9.4.2.11 getSkewSymmetricMatrix()**

```
static Matrix3d DCA::ExpCoords::getSkewSymmetricMatrix (
    const Vector3d & v ) [inline], [static], [private]
```

Get the skew symmetric matrix for a given vector.

**Parameters**

in	<i>v</i>	The vector.
----	----------	-------------

**Returns**

The skew symmetric matrix.

**9.4.2.12 safeACOS()**

```
static double DCA::ExpCoords::safeACOS (
    double x ) [inline], [static], [private]
```

Safe arcus cosinus.

**Parameters**

in	<i>x</i>	The value.
----	----------	------------

**Returns**

safe arcus cosinus.

**9.4.2.13 safeDACOS()**

```
static double DCA::ExpCoords::safeDACOS (
    double x ) [inline], [static], [private]
```

Safe derivative of arcus cosinus.

**Parameters**

in	x	The value.
----	---	------------

**Returns**

Safe derivative of the arcus cosinus.

**9.4.3 Member Data Documentation****9.4.3.1 tol**

```
constexpr static const double DCA::ExpCoords::tol = 1e-8 [static], [constexpr], [private]
```

Tolerance for avoiding singularities of Rodrigues' formula.

**9.4.3.2 \_PI**

```
constexpr static const double DCA::ExpCoords::_PI = 3.14159265359 [static], [constexpr],
[private]
```

Helper.

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

- [ExpCoords.h](#)

**9.5 DCA::FiniteDifference Class Reference**

Helper for finite differences (to test derivatives).

```
#include <FiniteDifference.h>
```

### 9.5.1 Detailed Description

Helper for finite differences (to test derivatives).

#### Attention

Not documented.

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

- [FiniteDifference.h](#)

## 9.6 DCA::Logger Class Reference

Logger class.

```
#include <Logger.h>
```

### Public Types

- enum [PRINT\\_COLOR](#) {  
    [RED](#) , [GREEN](#) , [YELLOW](#) , [BLUE](#) ,  
    [MAGENTA](#) , [CYAN](#) , [DEFAULT](#) }

*Possible color values.*

### Static Public Member Functions

- static void [print](#) ([PRINT\\_COLOR](#) color, const char \*fmt,...)

*Print a message.*

### 9.6.1 Detailed Description

Logger class.

Writes to stout.

### 9.6.2 Member Enumeration Documentation

#### 9.6.2.1 PRINT\_COLOR

```
enum DCA::Logger::PRINT\_COLOR
```

Possible color values.

## Enumerator

RED	
GREEN	
YELLOW	
BLUE	
MAGENTA	
CYAN	
DEFAULT	

### 9.6.3 Member Function Documentation

#### 9.6.3.1 print()

```
static void DCA::Logger::print (
    PRINT_COLOR color,
    const char * fmt,
    ... ) [inline], [static]
```

Print a message.

## Parameters

in	<i>color</i>	The color to use.
in	<i>fmt</i>	The thing to print, potentially containing format strings.

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

- [Logger.h](#)

## 9.7 DCA::NeighborsPairGenerator Class Reference

This generator computes the pairs which are in a certain threshold from each other. It does so by computing a single position for each primitive and selecting pairs based on the distance.

```
#include <Pair.h>
```

### Public Member Functions

- [NeighborsPairGenerator](#) (const double &radius)  
*Construct this generator with a given radius.*
- std::vector< [pair\\_t](#) > [generate](#) (const std::vector< [primitive\\_t](#) > &primitives) const  
*This function generates the pairs given the primitives, where the pairs are in a certain distance from each other.*
- std::vector< [pair\\_t](#) > [generate](#) (const std::vector< [primitive\\_t](#) > &primitives\_a, const std::vector< [primitive\\_t](#) > &primitives\_b) const  
*This function generates pairs given two vectors of primitives, where the pairs are in a certain distance from each other and are in separate vectors.*

## Private Member Functions

- double `estimate_distance` (const `primitive_t` &p\_A, const `primitive_t` &p\_B) const  
*Helper function for generate.*

## Private Attributes

- double `m_radius`  
*The radius.*

### 9.7.1 Detailed Description

This generator computes the pairs which are in a certain threshold from each other. It does so by computing a single position for each primitive and selecting pairs based on the distance.

### 9.7.2 Constructor & Destructor Documentation

#### 9.7.2.1 NeighborsPairGenerator()

```
DCA::NeighborsPairGenerator::NeighborsPairGenerator (
    const double & radius )
```

Construct this generator with a given radius.

##### Parameters

in	<i>radius</i>	The radius to search other primitives in.
----	---------------	---

### 9.7.3 Member Function Documentation

#### 9.7.3.1 generate() [1/2]

```
std::vector<pair_t> DCA::NeighborsPairGenerator::generate (
    const std::vector< primitive_t > & primitives ) const
```

This function generates the pairs given the primitives, where the pairs are in a certain distance from each other.

The returned vector consists of pairs, where each pair holds two numbers: The indices of the corresponding primitives which were given.

**Parameters**

in	<i>primitives</i>	All primitives to generate the pairs from.
----	-------------------	--

**Returns**

A vector of pairs of indices, where each index corresponds to a primitive in the primitives vector.

**9.7.3.2 generate() [2/2]**

```
std::vector<pair_t> DCA::NeighborsPairGenerator::generate (
    const std::vector< primitive_t > & primitives_a,
    const std::vector< primitive_t > & primitives_b ) const
```

This function generates pairs given two vectors of primitives, where the pairs are in a certain distance from each other and are in separate vectors.

The returned vector consists of pairs, where each pair holds two numbers: The indices of the corresponding primitives which were given. The first index corresponds to the primitives\_a vector, the second to the primitives\_b vector.

**Parameters**

in	<i>primitives↔ _a</i>	The first set of primitives.
in	<i>primitives↔ _b</i>	The second set of primitives.

**Returns**

A vector of pairs of indices, where the first index corresponds to a primitive in the primitives\_a vector and the second index to the primitives\_b vector.

**Attention**

Not all pairs are reported, only those between primitives\_a and primitives\_b!

**9.7.3.3 estimate\_distance()**

```
double DCA::NeighborsPairGenerator::estimate_distance (
    const primitive_t & p_A,
    const primitive_t & p_B ) const [private]
```

Helper function for generate.

## 9.7.4 Member Data Documentation

### 9.7.4.1 m\_radius

```
double DCA::NeighborsPairGenerator::m_radius [private]
```

The radius.

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

- [Pair.h](#)

## 9.8 DCA::NewtonOptimizer Class Reference

Simply newton optimizer class.

```
#include <NewtonOptimizer.h>
```

### Public Member Functions

- [NewtonOptimizer](#) (const std::string &name="NewtonOptimizer", double [solverResidual](#)=1e-5, int [maxLineSearchIterations](#)=15)  
*Construct a newton optimizer.*
- [~NewtonOptimizer](#) ()=default  
*Default destructor.*
- bool [optimize](#) (VectorXd &t, const VectorXd &s, const [Objective](#) &objective, int maxIterations)  
*Perform the optimization.*

### Public Attributes

- std::string [name](#)  
*Name of the solver (mainly used for printing)*
- double [solverResidual](#)  
*Accuracy when solver states that it has converged (compared to the norm of the gradient)*
- int [maxLineSearchIterations](#)  
*Max number of line search iterations before giving up.*
- double [lineSearchStartValue](#) = 1.0  
*Start value of line search. If unsuccessful, this value is cut in half until a suitable solution is found.*
- bool [useDynamicRegularization](#) = true  
*If activated, we regularize the hessian in case we don't have a descending search direction.*
- bool [printInfo](#) = false  
*Print infos to console to observe solver progress.*
- bool [checkLinearSystemSolve](#) = false  
*Check if linear system has been solved successfully (i.e. the correct linear solver has been used)*
- bool [checkHessianRank](#) = false  
*Check if hessian is invertible.*

## Private Attributes

- double [objectiveValue](#)  
*The objective value.*
- VectorXd [t\\_tmp](#)  
*Temporary t variable.*
- VectorXd [searchDir](#)  
*Current search direction.*
- VectorXd [gradient](#)  
*Current gradient.*
- MatrixXd [hessian](#)  
*Current hessian.*

### 9.8.1 Detailed Description

Simply newton optimizer class.

### 9.8.2 Constructor & Destructor Documentation

#### 9.8.2.1 NewtonOptimizer()

```
DCA::NewtonOptimizer::NewtonOptimizer (
    const std::string & name = "NewtonOptimizer",
    double solverResidual = 1e-5,
    int maxLineSearchIterations = 15 ) [inline]
```

Construct a newton optimizer.

#### Parameters

in	<i>name</i>	The name of the optimizer.
in	<i>solverResidual</i>	The solver residual.
in	<i>maxLineSearchIterations</i>	The maximum number of line search iterations.

#### 9.8.2.2 ~NewtonOptimizer()

```
DCA::NewtonOptimizer::~~NewtonOptimizer ( ) [default]
```

Default destructor.

### 9.8.3 Member Function Documentation



### 9.8.3.1 optimize()

```
bool DCA::NewtonOptimizer::optimize (
    VectorXd & t,
    const VectorXd & s,
    const Objective & objective,
    int maxIterations )
```

Perform the optimization.

#### Parameters

out	<i>t</i>	The resulting t values.
in	<i>s</i>	The state of two primitives.
in	<i>objective</i>	The objective to optimize.
in	<i>maxIterations</i>	The maximum number of iterations.

#### Returns

True, if the solver has converged.

## 9.8.4 Member Data Documentation

### 9.8.4.1 name

```
std::string DCA::NewtonOptimizer::name
```

Name of the solver (mainly used for printing)

### 9.8.4.2 solverResidual

```
double DCA::NewtonOptimizer::solverResidual
```

Accuracy when solver states that it has converged (compared to the norm of the gradient)

### 9.8.4.3 maxLineSearchIterations

```
int DCA::NewtonOptimizer::maxLineSearchIterations
```

Max number of line search iterations before giving up.

#### 9.8.4.4 lineSearchStartValue

```
double DCA::NewtonOptimizer::lineSearchStartValue = 1.0
```

Start value of line search. If unsuccessful, this value is cut in half until a suitable solution is found.

#### 9.8.4.5 useDynamicRegularization

```
bool DCA::NewtonOptimizer::useDynamicRegularization = true
```

If activated, we regularize the hessian in case we don't have a decending search direction.

#### 9.8.4.6 printInfo

```
bool DCA::NewtonOptimizer::printInfo = false
```

Print infos to console to observe solver progress.

#### 9.8.4.7 checkLinearSystemSolve

```
bool DCA::NewtonOptimizer::checkLinearSystemSolve = false
```

Check if linear system has been solved successfully (i.e. the correct linear solver has been used)

#### 9.8.4.8 checkHessianRank

```
bool DCA::NewtonOptimizer::checkHessianRank = false
```

Check if hessian is invertible.

#### 9.8.4.9 objectiveValue

```
double DCA::NewtonOptimizer::objectiveValue [private]
```

The objective value.

#### 9.8.4.10 t\_tmp

`VectorXd DCA::NewtonOptimizer::t_tmp [private]`

Temporary t variable.

#### 9.8.4.11 searchDir

`VectorXd DCA::NewtonOptimizer::searchDir [private]`

Current search direction.

#### 9.8.4.12 gradient

`VectorXd DCA::NewtonOptimizer::gradient [private]`

Current gradient.

#### 9.8.4.13 hessian

`MatrixXd DCA::NewtonOptimizer::hessian [private]`

Current hessian.

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

- [NewtonOptimizer.h](#)

## 9.9 DCA::Objective Class Reference

The main objective which used for computing distances and their derivatives.

```
#include <Objective.h>
```

## Public Member Functions

- [Objective](#) ([primitive\\_t primitive\\_A](#), [primitive\\_t primitive\\_B](#))  
*Create an objective for a pair of primitives.*
- [~Objective](#) ()=default  
*Default destructor.*
- `std::pair< Vector3d, Vector3d >` [compute\\_Ps](#) (const VectorXd &s, const VectorXd &t) const  
*Compute points on the primitives based on s and t.*
- `std::pair< int, int >` [get\\_sizes\\_t](#) () const  
*Return both sizes of t.*
- `int` [get\\_total\\_size\\_t](#) () const  
*Return the sum of t sizes.*

## D

*D and its derivatives*

- `double` [compute\\_D](#) (const VectorXd &s, const VectorXd &t) const  
*Compute the distance.*
- `void` [compute\\_pDpT](#) (VectorXd &pDpT, const VectorXd &s, const VectorXd &t) const  
*Compute the partial derivative of the distance w.r.t. t.*
- `void` [compute\\_pDpS](#) (VectorXd &pDpS, const VectorXd &s, const VectorXd &t) const  
*Compute the partial derivative of the distance w.r.t. s.*
- `void` [compute\\_p2DpT2](#) (MatrixXd &p2DpT2, const VectorXd &s, const VectorXd &t) const  
*Compute the second partial derivative of the distance w.r.t. t.*
- `void` [compute\\_p2DpS2](#) (MatrixXd &p2DpS2, const VectorXd &s, const VectorXd &t) const  
*Compute the second partial derivative of the distance w.r.t. s.*
- `void` [compute\\_p2DpTpS](#) (MatrixXd &p2DpTpS, const VectorXd &s, const VectorXd &t) const  
*Compute the mixed partial derivative of the distance w.r.t. t and s.*

## O

*O and its derivatives*

- `double` [compute\\_O](#) (const VectorXd &s, const VectorXd &t) const  
*Compute the objective value.*
- `void` [compute\\_pOpT](#) (VectorXd &pOpT, const VectorXd &s, const VectorXd &t) const  
*Compute the partial derivative of the objective value w.r.t. t.*
- `void` [compute\\_pOpS](#) (VectorXd &pOpS, const VectorXd &s, const VectorXd &t) const  
*Compute the partial derivative of the objective value w.r.t. s.*
- `void` [compute\\_p2OpT2](#) (MatrixXd &p2OpT2, const VectorXd &s, const VectorXd &t) const  
*Compute the second partial derivative of the objective value w.r.t. t.*
- `void` [compute\\_p2OpS2](#) (MatrixXd &p2OpS2, const VectorXd &s, const VectorXd &t) const  
*Compute the second partial derivative of the objective value w.r.t. s.*
- `void` [compute\\_p2OpTpS](#) (MatrixXd &p2OpTpS, const VectorXd &s, const VectorXd &t) const  
*Compute the mixed partial derivative of the objective value w.r.t. t and s.*

## Static Public Member Functions

- `static std::string` [get\\_primitives\\_description](#) ([primitive\\_t primitive\\_A](#), [primitive\\_t primitive\\_B](#))  
*Get a string of both primitive descriptions.*

## Public Attributes

- double `regularizerWeight` = 0.01  
*Regularitaion wheight for the objective.*
- double `constraintWeight` = 10.0  
*Constraint weight for the objective.*
- `SoftUnilateralConstraint unilateralConstraint` = `SoftUnilateralConstraint(0.0, 1.0, 1e-3)`  
*The soft unilateral constraint.*
- const `primitive_t primitive_A`  
*The first primitive.*
- const `primitive_t primitive_B`  
*The seonc primitive.*

## Private Types

- enum `PRIMITIVE` { `A`, `B` }  
*Helper for the first or second primitive.*

## Private Member Functions

- `Vector6d get_s` (const `VectorXd` &s, `PRIMITIVE` P) const  
*Get s for a primitive.*
- `VectorXd get_t` (const `VectorXd` &t, `PRIMITIVE` P) const  
*Get s for a primitive.*
- void `check_inputs` (const `VectorXd` &s, const `VectorXd` &t) const  
*Check s and t.*

### 9.9.1 Detailed Description

The main objective which used for computing distances and their derivatives.

### 9.9.2 Member Enumeration Documentation

#### 9.9.2.1 PRIMITIVE

```
enum DCA::Objective::PRIMITIVE [private]
```

Helper for the first or second primitive.

Enumerator

A	
B	

### 9.9.3 Constructor & Destructor Documentation

#### 9.9.3.1 Objective()

```
DCA::Objective::Objective (
    primitive_t primitive_A,
    primitive_t primitive_B ) [inline]
```

Create an objective for a pair of primitives.

##### Parameters

in	<i>primitive</i> ↔ _A	The first primitive.
in	<i>primitive</i> ↔ _B	The second primitive.

#### 9.9.3.2 ~Objective()

```
DCA::Objective::~~Objective ( ) [default]
```

Default destructor.

### 9.9.4 Member Function Documentation

#### 9.9.4.1 compute\_Ps()

```
std::pair<Vector3d, Vector3d> DCA::Objective::compute_Ps (
    const VectorXd & s,
    const VectorXd & t ) const
```

Compute points on the primitives based on s and t.

##### Parameters

in	<i>s</i>	The state of both primitives, stacked.
in	<i>t</i>	The parameterization of both primitives, stacked.

**9.9.4.2 compute\_D()**

```
double DCA::Objective::compute_D (
    const VectorXd & s,
    const VectorXd & t ) const
```

Compute the distance.

**Parameters**

in	$s$	The state of the two primitives.
in	$t$	The parameterization of the two primitives.

**Returns**

The distance between the two primitives.

**9.9.4.3 compute\_pDpT()**

```
void DCA::Objective::compute_pDpT (
    VectorXd & pDpT,
    const VectorXd & s,
    const VectorXd & t ) const
```

Compute the partial derivative of the distance w.r.t.  $t$ .

**Parameters**

out	$pDpT$	$\frac{\partial D}{\partial t}$
in	$s$	The state of the two primitives.
in	$t$	The parameterization of the two primitives.

**9.9.4.4 compute\_pDpS()**

```
void DCA::Objective::compute_pDpS (
    VectorXd & pDpS,
    const VectorXd & s,
    const VectorXd & t ) const
```

Compute the partial derivative of the distance w.r.t.  $s$ .

**Parameters**

out	$pDpS$	$\frac{\partial D}{\partial s}$
in	$s$	The state of the two primitives.
in	$t$	The parameterization of the two primitives.

#### 9.9.4.5 compute\_p2DpT2()

```
void DCA::Objective::compute_p2DpT2 (
    MatrixXd & p2DpT2,
    const VectorXd & s,
    const VectorXd & t ) const
```

Compute the second partial derivative of the distance w.r.t. t.

##### Parameters

out	$p2DpT2$	$\frac{\partial^2 D}{\partial t^2}$
in	$s$	The state of the two primitives.
in	$t$	The parameterization of the two primitives.

#### 9.9.4.6 compute\_p2DpS2()

```
void DCA::Objective::compute_p2DpS2 (
    MatrixXd & p2DpS2,
    const VectorXd & s,
    const VectorXd & t ) const
```

Compute the second partial derivative of the distance w.r.t. s.

##### Parameters

out	$p2DpS2$	$\frac{\partial^2 D}{\partial s^2}$
in	$s$	The state of the two primitives.
in	$t$	The parameterization of the two primitives.

#### 9.9.4.7 compute\_p2DpTpS()

```
void DCA::Objective::compute_p2DpTpS (
    MatrixXd & p2DpTpS,
    const VectorXd & s,
    const VectorXd & t ) const
```

Compute the mixed partial derivative of the distance w.r.t. t and s.

##### Parameters

out	$p2DpTpS$	$\frac{\partial^2 D}{\partial t \partial s}$
in	$s$	The state of the two primitives.
in	$t$	The parameterization of the two primitives.



**9.9.4.8 compute\_O()**

```
double DCA::Objective::compute_O (
    const VectorXd & s,
    const VectorXd & t ) const
```

Compute the objective value.

**Parameters**

in	<i>s</i>	The state of the two primitives.
in	<i>t</i>	The parameterization of the two primitives.

**Returns**

The objective value.

**9.9.4.9 compute\_pOpT()**

```
void DCA::Objective::compute_pOpT (
    VectorXd & pOpT,
    const VectorXd & s,
    const VectorXd & t ) const
```

Compute the partial derivative of the objective value w.r.t. *t*.

**Parameters**

out	<i>pOpT</i>	$\frac{\partial O}{\partial t}$
in	<i>s</i>	The state of the two primitives.
in	<i>t</i>	The parameterization of the two primitives.

**9.9.4.10 compute\_pOpS()**

```
void DCA::Objective::compute_pOpS (
    VectorXd & pOpS,
    const VectorXd & s,
    const VectorXd & t ) const
```

Compute the partial derivative of the objective value w.r.t. *s*.

**Parameters**

out	$pOpS$	$\frac{\partial O}{\partial s}$
in	$s$	The state of the two primitives.
in	$t$	The parameterization of the two primitives.

**9.9.4.11 compute\_p2OpT2()**

```
void DCA::Objective::compute_p2OpT2 (
    MatrixXd & p2OpT2,
    const VectorXd & s,
    const VectorXd & t ) const
```

Compute the second partial derivative of the objective value w.r.t.  $t$ .

**Parameters**

out	$p2OpT2$	$\frac{\partial^2 O}{\partial t^2}$
in	$s$	The state of the two primitives.
in	$t$	The parameterization of the two primitives.

**9.9.4.12 compute\_p2OpS2()**

```
void DCA::Objective::compute_p2OpS2 (
    MatrixXd & p2OpS2,
    const VectorXd & s,
    const VectorXd & t ) const
```

Compute the second partial derivative of the objective value w.r.t.  $s$ .

**Parameters**

out	$p2OpS2$	$\frac{\partial^2 O}{\partial s^2}$
in	$s$	The state of the two primitives.
in	$t$	The parameterization of the two primitives.

**9.9.4.13 compute\_p2OpTpS()**

```
void DCA::Objective::compute_p2OpTpS (
    MatrixXd & p2OpTpS,
```

```
const VectorXd & s,
const VectorXd & t ) const
```

Compute the mixed partial derivative of the objective value w.r.t. t and s.

#### Parameters

out	$p2OpTpS$	$\frac{\partial^2 O}{\partial t \partial s}$
in	$s$	The state of the two primitives.
in	$t$	The parameterization of the two primitives.

#### 9.9.4.14 get\_sizes\_t()

```
std::pair<int, int> DCA::Objective::get_sizes_t ( ) const
```

Return both sizes of t.

#### Returns

Both sizes of t.

#### 9.9.4.15 get\_total\_size\_t()

```
int DCA::Objective::get_total_size_t ( ) const
```

Return the sum of t sizes.

#### Returns

The sum of t sizes.

#### 9.9.4.16 get\_primitives\_description()

```
static std::string DCA::Objective::get_primitives_description (
    primitive_t primitive_A,
    primitive_t primitive_B ) [static]
```

Get a string of both primitive descriptions.

#### Parameters

in	$primitive\_A$	The first primitive.
in	$primitive\_B$	The second primitive.

**Returns**

The string.

**9.9.4.17 get\_s()**

```
Vector6d DCA::Objective::get_s (
    const VectorXd & s,
    PRIMITIVE P ) const [private]
```

Get s for a primitive.

**Parameters**

in	$s$	The stacked s vector.
in	$P$	the primitive to use.

**Returns**

The state belonging to the given primitive.

**9.9.4.18 get\_t()**

```
VectorXd DCA::Objective::get_t (
    const VectorXd & t,
    PRIMITIVE P ) const [private]
```

Get s for a primitive.

**Parameters**

in	$t$	The stacked s vector.
in	$P$	the primitive to use.

**Returns**

The parameterization belonging to the given primitive.

**9.9.4.19 check\_inputs()**

```
void DCA::Objective::check_inputs (
    const VectorXd & s,
    const VectorXd & t ) const [private]
```

Check s and t.

**Parameters**

in	<i>s</i>	The stacked s vector.
in	<i>t</i>	the stacked t vector.

**Exceptions**

<code>std::runtime_error</code>	If the size is wrong.
---------------------------------	-----------------------

**9.9.5 Member Data Documentation****9.9.5.1 regularizerWeight**

```
double DCA::Objective::regularizerWeight = 0.01
```

Regularitaion wheight for the objective.

**9.9.5.2 constraintWeight**

```
double DCA::Objective::constraintWeight = 10.0
```

Constraint weight for the objective.

**9.9.5.3 unilateralConstraint**

```
SoftUnilateralConstraint DCA::Objective::unilateralConstraint = SoftUnilateralConstraint(0.0, 1.0, 1e-3)
```

The soft unilateral constraint.

**9.9.5.4 primitive\_A**

```
const primitive_t DCA::Objective::primitive_A
```

The first primitive.

### 9.9.5.5 primitive\_B

```
const primitive_t DCA::Objective::primitive_B
```

The seonc primitive.

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

- [Objective.h](#)

## 9.10 DCA::PermutationPairGenerator Class Reference

This generator creates all possible permutations of pairs. This means, the amount of pairs created is  $n^2/2$ , where  $n$  is the number of primitives. Pairs are not returned twice (0, 1) and (1, 0).

```
#include <Pair.h>
```

### Public Member Functions

- `std::vector< pair_t > generate (const std::vector< primitive_t > &primitives) const`  
*This function generates all pairs given the primitives.*

#### 9.10.1 Detailed Description

This generator creates all possible permutations of pairs. This means, the amount of pairs created is  $n^2/2$ , where  $n$  is the number of primitives. Pairs are not returned twice (0, 1) and (1, 0).

#### 9.10.2 Member Function Documentation

##### 9.10.2.1 generate()

```
std::vector<pair_t> DCA::PermutationPairGenerator::generate (
    const std::vector< primitive_t > & primitives ) const
```

This function generates all pairs given the primitives.

The returned vector consists of pairs, where each pair holds two numbers: The indices of the corresponding primitives which were given.

#### Parameters

in	<i>primitives</i>	All primitives to generate the pairs from.
----	-------------------	--

**Returns**

A vector of pairs of indices, where each index corresponds to a primitive in the primitives vector.

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

- [Pair.h](#)

## 9.11 DCA::Primitive Class Reference

Definition of a Primitive.

```
#include <Primitives.h>
```

### Public Member Functions

- [Primitive](#) (const std::string &description, const double &safetyMargin)  
*Create a primitive.*
- virtual [~Primitive](#) ()=default  
*Default destructor.*
- virtual Vector3d [compute\\_P](#) (const [Vector6d](#) &s, const VectorXd &t) const =0  
*Compute a point on the primitive.*
- virtual Eigen::Matrix< double, 3, 6 > [compute\\_dPdS](#) (const [Vector6d](#) &s, const VectorXd &t) const =0  
*Compute the derivative of a point on the primitive with respect to s.*
- virtual Eigen::Matrix< double, 3, -1 > [compute\\_dPdT](#) (const [Vector6d](#) &s, const VectorXd &t) const =0  
*Compute the derivative of a point on the primitive with respect to t.*
- virtual std::array< Eigen::Matrix< double, 3, 6 >, 6 > [compute\\_d2PdS2](#) (const [Vector6d](#) &s, const VectorXd &t) const =0  
*Compute the second derivative of a point on the primitive with respect to s.*
- virtual std::vector< Eigen::Matrix< double, 3, 6 >, 6 > [compute\\_d2PdSdT](#) (const [Vector6d](#) &s, const VectorXd &t) const =0  
*Compute the mixed second derivative of a point on the primitive with respect to s and t.*
- virtual [Vector6d](#) [get\\_s](#) () const =0  
*Helper to get the state.*
- virtual int [SIZE\\_T](#) () const =0  
*Helper to get the size of t.*
- Vector3d [get\\_center\\_point](#) () const  
*Get the center point.*
- virtual double [get\\_largest\\_dimension\\_from\\_center](#) () const =0  
*Get the largest dimension.*

### Public Attributes

- double [safetyMargin](#)  
*Internal storage of the safety margin.*

## Protected Member Functions

- void [check\\_t](#) (const VectorXd &t) const  
*Check the size of t.*

### 9.11.1 Detailed Description

Definition of a Primitive.

Each Primitive can be described by a 6-dimensional state and a n-dimensional parameterization vector.

This means, any convex primitive which can be described in a continuously differentiable manner can be implemented.

The inheritance on [FiniteDifference](#) is just to make the automatic derivative checking easier.

### 9.11.2 Constructor & Destructor Documentation

#### 9.11.2.1 Primitive()

```
DCA::Primitive::Primitive (
    const std::string & description,
    const double & safetyMargin )
```

Create a primitive.

##### Parameters

in	<i>description</i>	The description of the primitive.
in	<i>safetyMargin</i>	The safety margin of the primitive.

##### Exceptions

<i>std::runtime_error</i>	if safety margin < 0
---------------------------	----------------------

#### 9.11.2.2 ~Primitive()

```
virtual DCA::Primitive::~~Primitive ( ) [virtual], [default]
```

Default destructor.



### 9.11.3 Member Function Documentation

#### 9.11.3.1 compute\_P()

```
virtual Vector3d DCA::Primitive::compute_P (
    const Vector6d & s,
    const VectorXd & t ) const [pure virtual]
```

Compute a point on the primitive.

Compute a point on the primitive based on s and t.

##### Parameters

in	<i>s</i>	The state of the primitive.
in	<i>t</i>	The parameterization of the point.

##### Returns

The point.

Implemented in [DCA::Box](#), [DCA::Rectangle](#), [DCA::Capsule](#), and [DCA::Sphere](#).

#### 9.11.3.2 compute\_dPdS()

```
virtual Eigen::Matrix<double, 3, 6> DCA::Primitive::compute_dPdS (
    const Vector6d & s,
    const VectorXd & t ) const [pure virtual]
```

Compute the derivative of a point on the primitive with respect to s.

Compute the derivative of a point on the primitive based on s and t.

##### Parameters

in	<i>s</i>	The state of the primitive.
in	<i>t</i>	The parameterization of the point.

##### Returns

The derivative  $\frac{dP}{ds}$ .

Implemented in [DCA::Box](#), [DCA::Rectangle](#), [DCA::Capsule](#), and [DCA::Sphere](#).

### 9.11.3.3 compute\_dPdT()

```
virtual Eigen::Matrix<double, 3, -1> DCA::Primitive::compute_dPdT (
    const Vector6d & s,
    const VectorXd & t ) const [pure virtual]
```

Compute the derivative of a point on the primitive with respect to t.

Compute the derivative of a point on the primitive based on s and t.

#### Parameters

in	$s$	The state of the primitive.
in	$t$	The parameterization of the point.

#### Returns

The derivative  $\frac{dP}{dt}$ .

Implemented in [DCA::Box](#), [DCA::Rectangle](#), [DCA::Capsule](#), and [DCA::Sphere](#).

### 9.11.3.4 compute\_d2PdS2()

```
virtual std::array<Eigen::Matrix<double, 3, 6>, 6> DCA::Primitive::compute_d2PdS2 (
    const Vector6d & s,
    const VectorXd & t ) const [pure virtual]
```

Compute the second derivative of a point on the primitive with respect to s.

Compute the second derivative of a point on the primitive based on s and t.

#### Parameters

in	$s$	The state of the primitive.
in	$t$	The parameterization of the point.

#### Returns

The second derivative  $\frac{d^2P}{ds^2}$  as an array (size 6).

Implemented in [DCA::Box](#), [DCA::Rectangle](#), [DCA::Capsule](#), and [DCA::Sphere](#).

### 9.11.3.5 compute\_d2PdSdT()

```
virtual std::vector<Eigen::Matrix<double, 3, 6> > DCA::Primitive::compute_d2PdSdT (
    const Vector6d & s,
    const VectorXd & t ) const [pure virtual]
```

Compute the mixed second derivative of a point on the primitive with respect to s and t.

Compute the mixed second derivative of a point on the primitive based on s and t.

#### Parameters

in	s	The state of the primitive.
in	t	The parameterization of the point.

#### Returns

The derivative  $\frac{d^2 P}{ds dt}$  as a vector (size t).

Implemented in [DCA::Box](#), [DCA::Rectangle](#), [DCA::Capsule](#), and [DCA::Sphere](#).

#### 9.11.3.6 get\_s()

```
virtual Vector6d DCA::Primitive::get_s ( ) const [pure virtual]
```

Helper to get the state.

#### Returns

The state of the primitive.

Implemented in [DCA::Box](#), [DCA::Rectangle](#), [DCA::Capsule](#), and [DCA::Sphere](#).

#### 9.11.3.7 SIZE\_T()

```
virtual int DCA::Primitive::SIZE_T ( ) const [pure virtual]
```

Helper to get the size of t.

#### Returns

The number of dimensions needed for parameterization.

Implemented in [DCA::Box](#), [DCA::Rectangle](#), [DCA::Capsule](#), and [DCA::Sphere](#).

### 9.11.3.8 `get_center_point()`

```
Vector3d DCA::Primitive::get_center_point ( ) const
```

Get the center point.

Helper for pair generation.

#### Returns

The center point of the primitive.

### 9.11.3.9 `get_largest_dimension_from_center()`

```
virtual double DCA::Primitive::get_largest_dimension_from_center ( ) const [pure virtual]
```

Get the largest dimension.

Helper for pair generation.

#### Returns

The largest dimension from the center point.

Implemented in [DCA::Box](#), [DCA::Rectangle](#), [DCA::Capsule](#), and [DCA::Sphere](#).

### 9.11.3.10 `check_t()`

```
void DCA::Primitive::check_t (
    const VectorXd & t ) const [protected]
```

Check the size of t.

#### Parameters

in	<i>t</i>	The t vector.
----	----------	---------------

#### Exceptions

<code>std::runtime_error</code>	If the size is wrong.
---------------------------------	-----------------------

## 9.11.4 Member Data Documentation

### 9.11.4.1 safetyMargin

```
double DCA::Primitive::safetyMargin
```

Internal storage of the safety margin.

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

- [Primitives.h](#)

## 9.12 DCA::Rectangle Class Reference

Definition of a Rectangle.

```
#include <Primitives.h>
```

### Public Member Functions

- [Rectangle](#) (const Vector3d &center, const Matrix3d &orientation, const Vector2d &dimensions, const double &safetyMargin=0.001)
- [~Rectangle](#) ()=default  
*Default destructor.*
- Vector3d [compute\\_P](#) (const Vector6d &s, const VectorXd &t) const override  
*Compute a point on the primitive.*
- Eigen::Matrix< double, 3, 6 > [compute\\_dPdS](#) (const Vector6d &s, const VectorXd &t) const override  
*Compute the derivative of a point on the primitive with respect to s.*
- Eigen::Matrix< double, 3, -1 > [compute\\_dPdT](#) (const Vector6d &s, const VectorXd &t) const override  
*Compute the derivative of a point on the primitive with respect to t.*
- std::array< Eigen::Matrix< double, 3, 6 >, 6 > [compute\\_d2PdS2](#) (const Vector6d &s, const VectorXd &t) const override  
*Compute the second derivative of a point on the primitive with respect to s.*
- std::vector< Eigen::Matrix< double, 3, 6 > > [compute\\_d2PdSdT](#) (const Vector6d &s, const VectorXd &t) const override  
*Compute the mixed second derivative of a point on the primitive with respect to s and t.*
- Vector6d [get\\_s](#) () const override  
*Helper to get the state.*
- int [SIZE\\_T](#) () const override  
*Helper to get the size of t.*
- double [get\\_largest\\_dimension\\_from\\_center](#) () const override  
*Get the largest dimension.*

### Public Attributes

- Vector3d [center](#)  
*The center of mass of the rectangle.*
- Matrix3d [orientation](#)  
*The orientation of the rectangle.*
- Vector2d [dimensions](#)  
*Convention: local dimensions in x and z direction.*

## Private Member Functions

- `std::pair< Vector3d, Vector3d > getLocalVectors () const`  
*Get vectors from the coordinates.*

## Additional Inherited Members

### 9.12.1 Detailed Description

Definition of a Rectangle.

The state of a rectangle is as follows: (x1, y1, z1, th1, th2, th3), which means that it is center of mass and the exponential map theta.

The parameterization of a rectangle is as follows: (t1, t2), which means, the parameterisation of a bounded plane.

### 9.12.2 Constructor & Destructor Documentation

#### 9.12.2.1 Rectangle()

```
DCA::Rectangle::Rectangle (
    const Vector3d & center,
    const Matrix3d & orientation,
    const Vector2d & dimensions,
    const double & safetyMargin = 0.001 )
```

Create a rectangle primitive.

#### Parameters

in	<i>center</i>	The center of mass.
in	<i>orientation</i>	The orientation of the rectangle.
in	<i>dimensions</i>	The two dimensions of the rectangle in x and z direction.
in	<i>safetyMargin</i>	The safety margin of the rectangle.

#### Exceptions

<i>std::runtime_error</i>	If safety margin < 0.
<i>std::runtime_error</i>	If any dimension < 0.
<i>std::runtime_error</i>	If the orientation is invalid.

### 9.12.2.2 ~Rectangle()

```
DCA::Rectangle::~~Rectangle ( ) [default]
```

Default destructor.

## 9.12.3 Member Function Documentation

### 9.12.3.1 compute\_P()

```
Vector3d DCA::Rectangle::compute_P (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```

Compute a point on the primitive.

Compute a point on the primitive based on s and t.

#### Parameters

in	<i>s</i>	The state of the primitive.
in	<i>t</i>	The parameterization of the point.

#### Returns

The point.

Implements [DCA::Primitive](#).

### 9.12.3.2 compute\_dPdS()

```
Eigen::Matrix<double, 3, 6> DCA::Rectangle::compute_dPdS (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```

Compute the derivative of a point on the primitive with respect to s.

Compute the derivative of a point on the primitive based on s and t.

#### Parameters

in	<i>s</i>	The state of the primitive.
in	<i>t</i>	The parameterization of the point.

**Returns**

The derivative  $\frac{dP}{ds}$ .

Implements [DCA::Primitive](#).

**9.12.3.3 compute\_dPdT()**

```
Eigen::Matrix<double, 3, -1> DCA::Rectangle::compute_dPdT (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```

Compute the derivative of a point on the primitive with respect to t.

Compute the derivative of a point on the primitive based on s and t.

**Parameters**

in	<i>s</i>	The state of the primitive.
in	<i>t</i>	The parameterization of the point.

**Returns**

The derivative  $\frac{dP}{dt}$ .

In the case of a rectangle, this has size 3x2.

Implements [DCA::Primitive](#).

**9.12.3.4 compute\_d2PdS2()**

```
std::array<Eigen::Matrix<double, 3, 6>, 6> DCA::Rectangle::compute_d2PdS2 (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```

Compute the second derivative of a point on the primitive with respect to s.

Compute the second derivative of a point on the primitive based on s and t.

**Parameters**

in	<i>s</i>	The state of the primitive.
in	<i>t</i>	The parameterization of the point.



**Returns**

The second derivative  $\frac{d^2 P}{ds^2}$  as an array (size 6).

Implements [DCA::Primitive](#).

**9.12.3.5 compute\_d2PdSdT()**

```
std::vector<Eigen::Matrix<double, 3, 6> > DCA::Rectangle::compute_d2PdSdT (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```

Compute the mixed second derivative of a point on the primitive with respect to s and t.

Compute the mixed second derivative of a point on the primitive based on s and t.

**Parameters**

in	s	The state of the primitive.
in	t	The parameterization of the point.

**Returns**

The derivative  $\frac{d^2 P}{dsdt}$  as a vector (size t).

In the case of a rectangle, this returns a vector with size 2.

Implements [DCA::Primitive](#).

**9.12.3.6 get\_s()**

```
Vector6d DCA::Rectangle::get_s ( ) const [override], [virtual]
```

Helper to get the state.

**Returns**

The state of the primitive.

Implements [DCA::Primitive](#).

### 9.12.3.7 SIZE\_T()

```
int DCA::Rectangle::SIZE_T ( ) const [override], [virtual]
```

Helper to get the size of t.

#### Returns

The number of dimensions needed for parameterization.

In the case of a rectangle, this returns 2.

Implements [DCA::Primitive](#).

### 9.12.3.8 get\_largest\_dimension\_from\_center()

```
double DCA::Rectangle::get_largest_dimension_from_center ( ) const [override], [virtual]
```

Get the largest dimension.

Helper for pair generation.

#### Returns

The largest dimension from the center point.

Implements [DCA::Primitive](#).

### 9.12.3.9 getLocalVectors()

```
std::pair<Vector3d, Vector3d> DCA::Rectangle::getLocalVectors ( ) const [private]
```

Get vectors from the coordinates.

#### Returns

The local vectors.

## 9.12.4 Member Data Documentation

#### 9.12.4.1 center

```
Vector3d DCA::Rectangle::center
```

The center of mass of the rectangle.

#### 9.12.4.2 orientation

```
Matrix3d DCA::Rectangle::orientation
```

The orientation of the rectangle.

#### 9.12.4.3 dimensions

```
Vector2d DCA::Rectangle::dimensions
```

Convention: local dimensions in x and z direction.

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

- [Primitives.h](#)

## 9.13 DCA::SoftUnilateralConstraint Class Reference

This class is a soft constraint, meaning a constraint is softfied.

```
#include <SoftUnilateralConstraint.h>
```

### Public Member Functions

- [SoftUnilateralConstraint](#) (double [limit](#), double stiffness, double [epsilon](#))  
*Create a constraint.*
- [~SoftUnilateralConstraint](#) ()=default  
*Default deconstructor.*
- void [setLimit](#) (double [limit](#))  
*Set a new limit.*
- void [setEpsilon](#) (double eps)  
*Set a new epsilon.*
- void [setStiffness](#) (double s)  
*Set a new stiffness.*
- double [evaluate](#) (double x) const  
*Compute the force acting on x.*
- double [computeDerivative](#) (double x) const  
*Compute first derivative of the force acting on x.*
- double [computeSecondDerivative](#) (double x) const  
*Compute the second derivative of the force acting on x.*

## Private Attributes

- double `a1`
- double `b1`
- double `c1`
- double `a2`
- double `b2`
- double `c2`
- double `d2`
- double `epsilon`
- double `limit`

*Internal values.*

### 9.13.1 Detailed Description

This class is a soft constraint, meaning a constraint is softened.

used to model unilateral constraints of the type  $x < u$  using a C2 penalty energy  $f(x)$ .

- $u$  is the upper limit that  $x$  needs to be less than
- if  $x < u$ , then the energy of the constraint, its gradient and hessian are all 0 (i.e. inactive)
- `epsilon` is the value away from the limit (how much smaller should  $x$  be compared to  $u$ ) after which  $f(x) = 0$
- `stiffness` controls the rate at which  $f(x)$  increases if  $x > u$

### 9.13.2 Constructor & Destructor Documentation

#### 9.13.2.1 SoftUnilateralConstraint()

```
DCA::SoftUnilateralConstraint::SoftUnilateralConstraint (
    double limit,
    double stiffness,
    double epsilon )
```

Create a constraint.

#### Parameters

in	<i>limit</i>	The limit of the constraint.
in	<i>stiffness</i>	The stiffness of the softifying.
in	<i>epsilon</i>	The limit (how close to the limit should the value be able to be).

### 9.13.2.2 ~SoftUnilateralConstraint()

```
DCA::SoftUnilateralConstraint::~~SoftUnilateralConstraint ( ) [default]
```

Default destructor.

## 9.13.3 Member Function Documentation

### 9.13.3.1 setLimit()

```
void DCA::SoftUnilateralConstraint::setLimit (
    double limit )
```

Set a new limit.

#### Parameters

in	<i>limit</i>	The new limit.
----	--------------	----------------

### 9.13.3.2 setEpsilon()

```
void DCA::SoftUnilateralConstraint::setEpsilon (
    double eps )
```

Set a new epsilon.

#### Parameters

in	<i>eps</i>	The new epsilon.
----	------------	------------------

### 9.13.3.3 setStiffness()

```
void DCA::SoftUnilateralConstraint::setStiffness (
    double s )
```

Set a new stiffness.

#### Parameters

in	<i>s</i>	The new stiffness.
----	----------	--------------------

#### 9.13.3.4 evaluate()

```
double DCA::SoftUnilateralConstraint::evaluate (  
    double x ) const
```

Compute the force acting on x.

##### Parameters

in	x	The current position
----	---	----------------------

##### Returns

The force F acting on x.

#### 9.13.3.5 computeDerivative()

```
double DCA::SoftUnilateralConstraint::computeDerivative (  
    double x ) const
```

Compute first derivative of the force acting on x.

##### Parameters

in	x	The current position
----	---	----------------------

##### Returns

The first derivative of F evaluated at x.

#### 9.13.3.6 computeSecondDerivative()

```
double DCA::SoftUnilateralConstraint::computeSecondDerivative (  
    double x ) const
```

Compute the second derivative of the force acting on x.

##### Parameters

in	x	The current position
----	---	----------------------

#### Returns

The second derivative of F evaluated at x.

### 9.13.4 Member Data Documentation

#### 9.13.4.1 a1

```
double DCA::SoftUnilateralConstraint::a1 [private]
```

#### 9.13.4.2 b1

```
double DCA::SoftUnilateralConstraint::b1 [private]
```

#### 9.13.4.3 c1

```
double DCA::SoftUnilateralConstraint::c1 [private]
```

#### 9.13.4.4 a2

```
double DCA::SoftUnilateralConstraint::a2 [private]
```

#### 9.13.4.5 b2

```
double DCA::SoftUnilateralConstraint::b2 [private]
```

#### 9.13.4.6 c2

```
double DCA::SoftUnilateralConstraint::c2 [private]
```

#### 9.13.4.7 d2

```
double DCA::SoftUnilateralConstraint::d2 [private]
```

#### 9.13.4.8 epsilon

```
double DCA::SoftUnilateralConstraint::epsilon [private]
```

#### 9.13.4.9 limit

```
double DCA::SoftUnilateralConstraint::limit [private]
```

Internal values.

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

- [SoftUnilateralConstraint.h](#)

## 9.14 DCA::Solver Class Reference

Wrapper around a newton optimizer and an objective.

```
#include <Solver.h>
```

### Public Member Functions

- [Solver](#) ([primitive\\_t](#) primitive\_A, [primitive\\_t](#) primitive\_B)  
*Create a solver.*
- [~Solver](#) ()=default  
*Default destructor.*
- void [compute\\_t](#) (VectorXd &t, const VectorXd &s, bool forFD=false) const  
*Compute the parameterization.*
- void [compute\\_dtds](#) (MatrixXd &dtds, const VectorXd &s, const VectorXd &t) const  
*Compute the derivative.*
- double [compute\\_D](#) (const VectorXd &s, const VectorXd &t) const  
*Compute the distance between two primitives.*
- void [compute\\_dDdS](#) (VectorXd &dDdS, const VectorXd &s, const VectorXd &t) const  
*Compute the derivative of the distance between two primitives w.r.t. s.*
- void [compute\\_d2DdS2](#) (MatrixXd &d2DdS2, const VectorXd &s, const VectorXd &t) const  
*Compute the second derivative of the distance between two primitives w.r.t. s.*
- std::pair< Vector3d, Vector3d > [compute\\_closest\\_points](#) (const VectorXd &s, const VectorXd &t) const  
*Compute the closest points which yield the shortest distance.*
- bool [are\\_t\\_and\\_s\\_in\\_sync](#) (const VectorXd &s, const VectorXd &t) const  
*Check if t and s are in sync.*
- [Vector12d get\\_s\\_from\\_primitives](#) () const  
*Get the state of both primitives, stacked.*
- int [get\\_total\\_size\\_t](#) () const  
*Get the total size of t.*



## Private Attributes

- [Objective](#) `objective`  
*The objective to optimize.*
- [NewtonOptimizer](#) `optimizer`  
*The newton optimizer.*

### 9.14.1 Detailed Description

Wrapper around a newton optimizer and an objective.

This class actually finds the shortest distance.

### 9.14.2 Constructor & Destructor Documentation

#### 9.14.2.1 Solver()

```
DCA::Solver::Solver (
    primitive_t primitive_A,
    primitive_t primitive_B )
```

Create a solver.

##### Parameters

in	<code>primitive↵ _A</code>	The first primitive.
in	<code>primitive↵ _B</code>	The second primitive.

#### 9.14.2.2 ~Solver()

```
DCA::Solver::~Solver ( ) [default]
```

Default destructor.

### 9.14.3 Member Function Documentation

### 9.14.3.1 compute\_t()

```
void DCA::Solver::compute_t (
    VectorXd & t,
    const VectorXd & s,
    bool forFD = false ) const
```

Compute the parameterization.

#### Parameters

out	<i>t</i>	The computed parameterization, stacked.
in	<i>s</i>	The state of the two primitives, stacked.
in	<i>forFD</i>	Should be used with "false". True will lower residual, which makes it slower but more accurate.

### 9.14.3.2 compute\_dtds()

```
void DCA::Solver::compute_dtds (
    MatrixXd & dtds,
    const VectorXd & s,
    const VectorXd & t ) const
```

Compute the derivative.

Computes  $\frac{dt}{ds}$ .

#### Parameters

out	<i>dtds</i>	$\frac{dt}{ds}$ .
in	<i>s</i>	The state of both primitives, stacked.
in	<i>t</i>	The parameterization of both primitives, stacked.

### 9.14.3.3 compute\_D()

```
double DCA::Solver::compute_D (
    const VectorXd & s,
    const VectorXd & t ) const
```

Compute the distance between two primitives.

#### Parameters

in	<i>s</i>	The state of the two primitives.
in	<i>t</i>	The parameterization of the two primitives, stacked.

**Returns**

The shortest distance between both primitives, stacked.

**9.14.3.4 compute\_dDdS()**

```
void DCA::Solver::compute_dDdS (
    VectorXd & dDdS,
    const VectorXd & s,
    const VectorXd & t ) const
```

Compute the derivative of the distance between two primitives w.r.t.  $s$ .

**Parameters**

out	$dDdS$	The derivative $\frac{dD}{ds}$ .
in	$s$	The state of the two primitives, stacked.
in	$t$	The parameterization of the two primitives, stacked.

**9.14.3.5 compute\_d2DdS2()**

```
void DCA::Solver::compute_d2DdS2 (
    MatrixXd & d2DdS2,
    const VectorXd & s,
    const VectorXd & t ) const
```

Compute the second derivative of the distance between two primitives w.r.t.  $s$ .

**Parameters**

out	$d2DdS2$	The derivative $\frac{d^2D}{ds^2}$ .
in	$s$	The state of the two primitives, stacked.
in	$t$	The parameterization of the two primitives, stacked.

**9.14.3.6 compute\_closest\_points()**

```
std::pair<Vector3d, Vector3d> DCA::Solver::compute_closest_points (
    const VectorXd & s,
    const VectorXd & t ) const
```

Compute the closest points which yield the shortest distance.

**Parameters**

in	<i>s</i>	The state of the two primitives, stacked.
in	<i>t</i>	The parameterization of the two primitives, stacked.

**Returns**

A pair of points, where the distance between those points is the shortest between the two primitives.

**9.14.3.7 are\_t\_and\_s\_in\_sync()**

```
bool DCA::Solver::are_t_and_s_in_sync (
    const VectorXd & s,
    const VectorXd & t ) const
```

Check if *t* and *s* are in sync.

**Parameters**

in	<i>s</i>	The state of the two primitives, stacked.
in	<i>t</i>	The parameterization of the two primitives, stacked.

**Returns**

True, if *t* and *s* are in sync. False, otherwise.

**9.14.3.8 get\_s\_from\_primitives()**

```
Vector12d DCA::Solver::get_s_from_primitives ( ) const
```

Get the state of both primitives, stacked.

**Returns**

The state of the two primitives, stacked.

**9.14.3.9 get\_total\_size\_t()**

```
int DCA::Solver::get_total_size_t ( ) const
```

Get the total size of *t*.

**Returns**

The total size of *t*.

## 9.14.4 Member Data Documentation

### 9.14.4.1 objective

`Objective` DCA::Solver::objective [private]

The objective to optimize.

### 9.14.4.2 optimizer

`NewtonOptimizer` DCA::Solver::optimizer [mutable], [private]

The newton optimizer.

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

- [Solver.h](#)

## 9.15 DCA::Sphere Class Reference

Definition of a Sphere.

```
#include <Primitives.h>
```

### Public Member Functions

- [Sphere](#) (const Vector3d &position, const double &radius)
- [~Sphere](#) ()=default  
*Default destructor.*
- Vector3d [compute\\_P](#) (const [Vector6d](#) &s, const VectorXd &t) const override  
*Compute a point on the primitive.*
- Eigen::Matrix< double, 3, 6 > [compute\\_dPdS](#) (const [Vector6d](#) &s, const VectorXd &t) const override  
*Compute the derivative of a point on the primitive with respect to s.*
- Eigen::Matrix< double, 3, -1 > [compute\\_dPdT](#) (const [Vector6d](#) &s, const VectorXd &t) const override  
*Compute the derivative of a point on the primitive with respect to t.*
- std::array< Eigen::Matrix< double, 3, 6 >, 6 > [compute\\_d2PdS2](#) (const [Vector6d](#) &s, const VectorXd &t) const override  
*Compute the second derivative of a point on the primitive with respect to s.*
- std::vector< Eigen::Matrix< double, 3, 6 > > [compute\\_d2PdSdT](#) (const [Vector6d](#) &s, const VectorXd &t) const override  
*Compute the mixed second derivative of a point on the primitive with respect to s and t.*
- [Vector6d get\\_s](#) () const override  
*Helper to get the state.*
- int [SIZE\\_T](#) () const override  
*Helper to get the size of t.*
- double [get\\_largest\\_dimension\\_from\\_center](#) () const override  
*Get the largest dimension.*

## Public Attributes

- Vector3d [position](#)

*The position of the sphere in 3d space.*

## Additional Inherited Members

### 9.15.1 Detailed Description

Definition of a Sphere.

The state of a sphere is as follows: (x, y, z, -, -, -), which means that it is the center of the sphere and the other three parameters are not used.

The parameterization of a sphere is as follows: (-), which means, it does not need any (0-dimensional).

### 9.15.2 Constructor & Destructor Documentation

#### 9.15.2.1 Sphere()

```
DCA::Sphere::Sphere (
    const Vector3d & position,
    const double & radius )
```

Create a sphere primitive.

##### Parameters

in	<i>position</i>	The position of the sphere.
in	<i>radius</i>	The safety margin around the center.

#### 9.15.2.2 ~Sphere()

```
DCA::Sphere::~~Sphere ( ) [default]
```

Default destructor.

### 9.15.3 Member Function Documentation

### 9.15.3.1 compute\_P()

```
Vector3d DCA::Sphere::compute_P (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```

Compute a point on the primitive.

Compute a point on the primitive based on s and t.

#### Parameters

in	<i>s</i>	The state of the primitive.
in	<i>t</i>	The parameterization of the point.

#### Returns

The point.

The parameterization *t* is not used, since it has size 0!

Implements [DCA::Primitive](#).

### 9.15.3.2 compute\_dPdS()

```
Eigen::Matrix<double, 3, 6> DCA::Sphere::compute_dPdS (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```

Compute the derivative of a point on the primitive with respect to s.

Compute the derivative of a point on the primitive based on s and t.

#### Parameters

in	<i>s</i>	The state of the primitive.
in	<i>t</i>	The parameterization of the point.

#### Returns

The derivative  $\frac{dP}{ds}$ .

Implements [DCA::Primitive](#).

### 9.15.3.3 compute\_dPdT()

```
Eigen::Matrix<double, 3, -1> DCA::Sphere::compute_dPdT (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```

Compute the derivative of a point on the primitive with respect to t.

Compute the derivative of a point on the primitive based on s and t.

#### Parameters

in	<i>s</i>	The state of the primitive.
in	<i>t</i>	The parameterization of the point.

#### Returns

The derivative  $\frac{dP}{dt}$ .

In the case of a sphere, this has size 3x0.

Implements [DCA::Primitive](#).

### 9.15.3.4 compute\_d2PdS2()

```
std::array<Eigen::Matrix<double, 3, 6>, 6> DCA::Sphere::compute_d2PdS2 (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```

Compute the second derivative of a point on the primitive with respect to s.

Compute the second derivative of a point on the primitive based on s and t.

#### Parameters

in	<i>s</i>	The state of the primitive.
in	<i>t</i>	The parameterization of the point.

#### Returns

The second derivative  $\frac{d^2P}{ds^2}$  as an array (size 6).

Implements [DCA::Primitive](#).



### 9.15.3.5 compute\_d2PdSdT()

```
std::vector<Eigen::Matrix<double, 3, 6> > DCA::Sphere::compute_d2PdSdT (
    const Vector6d & s,
    const VectorXd & t ) const [override], [virtual]
```

Compute the mixed second derivative of a point on the primitive with respect to s and t.

Compute the mixed second derivative of a point on the primitive based on s and t.

#### Parameters

in	$s$	The state of the primitive.
in	$t$	The parameterization of the point.

#### Returns

The derivative  $\frac{d^2 P}{ds dt}$  as a vector (size t).

In the case of a sphere, this returns an empty vector.

Implements [DCA::Primitive](#).

### 9.15.3.6 get\_s()

```
Vector6d DCA::Sphere::get_s ( ) const [override], [virtual]
```

Helper to get the state.

#### Returns

The state of the primitive.

Implements [DCA::Primitive](#).

### 9.15.3.7 SIZE\_T()

```
int DCA::Sphere::SIZE_T ( ) const [override], [virtual]
```

Helper to get the size of t.

#### Returns

The number of dimensions needed for parameterization.

In the case of a sphere, this returns 0.

Implements [DCA::Primitive](#).

#### 9.15.3.8 `get_largest_dimension_from_center()`

```
double DCA::Sphere::get_largest_dimension_from_center ( ) const [override], [virtual]
```

Get the largest dimension.

Helper for pair generation.

##### Returns

The largest dimension from the center point.

In the case of a sphere, this returns the radius.

Implements [DCA::Primitive](#).

### 9.15.4 Member Data Documentation

#### 9.15.4.1 `position`

```
Vector3d DCA::Sphere::position
```

The position of the sphere in 3d space.

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

- [Primitives.h](#)

## Chapter 10

# File Documentation

### 10.1 API.h File Reference

```
#include <DCA/Pair.h>
```

#### Classes

- class [DCA::API](#)  
*Public API.*

#### Namespaces

- [DCA](#)

### 10.2 ddR.h File Reference

```
#include <Eigen/Core>
```

#### Namespaces

- [DCA](#)

## Functions

- Eigen::Matrix3d [DCA::ddR\\_0\\_0](#) (const Eigen::Vector3d &v)  

$$\text{Compute } \frac{d^2 R}{d\theta_0 d\theta_0}.$$
- Eigen::Matrix3d [DCA::ddR\\_0\\_1](#) (const Eigen::Vector3d &v)  

$$\text{Compute } \frac{d^2 R}{d\theta_0 d\theta_1}.$$
- Eigen::Matrix3d [DCA::ddR\\_0\\_2](#) (const Eigen::Vector3d &v)  

$$\text{Compute } \frac{d^2 R}{d\theta_0 d\theta_2}.$$
- Eigen::Matrix3d [DCA::ddR\\_1\\_0](#) (const Eigen::Vector3d &v)  

$$\text{Compute } \frac{d^2 R}{d\theta_1 d\theta_0}.$$
- Eigen::Matrix3d [DCA::ddR\\_1\\_1](#) (const Eigen::Vector3d &v)  

$$\text{Compute } \frac{d^2 R}{d\theta_1 d\theta_1}.$$
- Eigen::Matrix3d [DCA::ddR\\_1\\_2](#) (const Eigen::Vector3d &v)  

$$\text{Compute } \frac{d^2 R}{d\theta_1 d\theta_2}.$$
- Eigen::Matrix3d [DCA::ddR\\_2\\_0](#) (const Eigen::Vector3d &v)  

$$\text{Compute } \frac{d^2 R}{d\theta_2 d\theta_0}.$$
- Eigen::Matrix3d [DCA::ddR\\_2\\_1](#) (const Eigen::Vector3d &v)  

$$\text{Compute } \frac{d^2 R}{d\theta_2 d\theta_1}.$$
- Eigen::Matrix3d [DCA::ddR\\_2\\_2](#) (const Eigen::Vector3d &v)  

$$\text{Compute } \frac{d^2 R}{d\theta_2 d\theta_2}.$$

## 10.3 ExpCoords.h File Reference

```
#include <DCA/Logger.h>
#include <DCA/Utils.h>
#include <DCA/ddR.h>
```

## Classes

- class [DCA::ExpCoords](#)  
*Represents Rotations using exponential coordinates.*

## Namespaces

- [DCA](#)

## 10.4 FiniteDifference.h File Reference

```
#include <DCA/Utils.h>
#include <functional>
```

## Classes

- class [DCA::FiniteDifference](#)  
*Helper for finite differences (to test derivatives).*

## Namespaces

- [DCA](#)

## 10.5 Logger.h File Reference

```
#include <stdarg.h>
#include <stdio.h>
#include <string.h>
#include <string>
#include <vector>
```

## Classes

- class [DCA::Logger](#)  
*Logger class.*

## Namespaces

- [DCA](#)

## 10.6 NewtonOptimizer.h File Reference

```
#include <DCA/Objective.h>
```

## Classes

- class [DCA::NewtonOptimizer](#)  
*Simply newton optimizer class.*

## Namespaces

- [DCA](#)

## 10.7 Objective.h File Reference

```
#include <DCA/FiniteDifference.h>
#include <DCA/Primitives.h>
#include <DCA/SoftUnilateralConstraint.h>
```

### Classes

- class [DCA::Objective](#)  
*The main objective which used for computing distances and their derivatives.*

### Namespaces

- [DCA](#)

## 10.8 Pair.h File Reference

```
#include <DCA/Primitives.h>
```

### Classes

- class [DCA::PermutationPairGenerator](#)  
*This generator creates all possible permutations of pairs. This means, the amount of pairs created is  $n^2/2$ , where  $n$  is the number of primitives. Pairs are not returned twice (0, 1) and (1, 0).*
- class [DCA::NeighborsPairGenerator](#)  
*This generator computes the pairs which are in a certain threshold from each other. It does so by computing a single position for each primitive and selecting pairs based on the distance.*

### Namespaces

- [DCA](#)

## 10.9 Primitives.h File Reference

```
#include <DCA/FiniteDifference.h>
#include <DCA/Utils.h>
#include <array>
#include <variant>
#include <vector>
```

## Classes

- class [DCA::Primitive](#)  
*Definition of a Primitive.*
- class [DCA::Sphere](#)  
*Definition of a Sphere.*
- class [DCA::Capsule](#)  
*Definition of a Capsule.*
- class [DCA::Rectangle](#)  
*Definition of a Rectangle.*
- class [DCA::Box](#)  
*Definition of a Box.*

## Namespaces

- [DCA](#)

## Typedefs

- using [DCA::primitive\\_t](#) = std::variant< Sphere, Capsule, Rectangle, Box >  
*All possible primitives.*

## 10.10 README.md File Reference

## 10.11 SoftUnilateralConstraint.h File Reference

### Classes

- class [DCA::SoftUnilateralConstraint](#)  
*This class is a soft constraint, meaning a constraint is softfied.*

### Namespaces

- [DCA](#)

## 10.12 Solver.h File Reference

```
#include <DCA/NewtonOptimizer.h>
#include <DCA/Objective.h>
```

### Classes

- class [DCA::Solver](#)  
*Wrapper around a newton optimizer and an objective.*

## Namespaces

- [DCA](#)

## 10.13 Utils.h File Reference

```
#include <Eigen/Core>
#include <Eigen/Geometry>
#include <vector>
```

## Namespaces

- [DCA](#)

## Typedefs

- typedef unsigned int [DCA::uint](#)  
*Easier access to a unsigned int.*
- using [DCA::Vector0d](#) = Eigen::Matrix< double, 0, 1 >  
*Easier access to a 0 vector of type double.*
- using [DCA::Vector1d](#) = Eigen::Matrix< double, 1, 1 >  
*Easier access to a 1 vector of type double.*
- using [DCA::Vector6d](#) = Eigen::Matrix< double, 6, 1 >  
*Easier access to a 6 vector of type double.*
- using [DCA::Vector12d](#) = Eigen::Matrix< double, 12, 1 >  
*Easier access to a 12 vector.*
- using [DCA::Matrix12d](#) = Eigen::Matrix< double, 12, 12 >  
*Easier access to a 12 by 12 matrix.*
- using [DCA::pair\\_t](#) = std::pair< size\_t, size\_t >  
*Easier access to a pair (corresponding of two indices)*



# Index

- [\\_PI](#)
    - [DCA::ExpCoords, 48](#)
  - [~Box](#)
    - [DCA::Box, 31](#)
  - [~Capsule](#)
    - [DCA::Capsule, 37](#)
  - [~NewtonOptimizer](#)
    - [DCA::NewtonOptimizer, 54](#)
  - [~Objective](#)
    - [DCA::Objective, 60](#)
  - [~Primitive](#)
    - [DCA::Primitive, 70](#)
  - [~Rectangle](#)
    - [DCA::Rectangle, 76](#)
  - [~SoftUnilateralConstraint](#)
    - [DCA::SoftUnilateralConstraint, 82](#)
  - [~Solver](#)
    - [DCA::Solver, 87](#)
  - [~Sphere](#)
    - [DCA::Sphere, 92](#)
- [A](#)
  - [DCA::Objective, 59](#)
- [a1](#)
  - [DCA::SoftUnilateralConstraint, 85](#)
- [a2](#)
  - [DCA::SoftUnilateralConstraint, 85](#)
- [API.h, 97](#)
- [are\\_t\\_and\\_s\\_in\\_sync](#)
  - [DCA::Solver, 90](#)
- [B](#)
  - [DCA::Objective, 59](#)
- [b1](#)
  - [DCA::SoftUnilateralConstraint, 85](#)
- [b2](#)
  - [DCA::SoftUnilateralConstraint, 85](#)
- [BLUE](#)
  - [DCA::Logger, 50](#)
- [Box](#)
  - [DCA::Box, 31](#)
- [c1](#)
  - [DCA::SoftUnilateralConstraint, 85](#)
- [c2](#)
  - [DCA::SoftUnilateralConstraint, 85](#)
- [Capsule](#)
  - [DCA::Capsule, 37](#)
- [center](#)
  - [DCA::Box, 35](#)
- [DCA::Rectangle, 80](#)
- [check\\_inputs](#)
  - [DCA::Objective, 66](#)
- [check\\_t](#)
  - [DCA::Primitive, 74](#)
- [checkHessianRank](#)
  - [DCA::NewtonOptimizer, 56](#)
- [checkLinearSystemSolve](#)
  - [DCA::NewtonOptimizer, 56](#)
- [compute\\_closest\\_points](#)
  - [DCA::API, 27, 29](#)
  - [DCA::Solver, 89](#)
- [compute\\_D](#)
  - [DCA::API, 26, 27](#)
  - [DCA::Objective, 60](#)
  - [DCA::Solver, 88](#)
- [compute\\_d2DdS2](#)
  - [DCA::API, 26, 28](#)
  - [DCA::Solver, 89](#)
- [compute\\_d2PdS2](#)
  - [DCA::Box, 33](#)
  - [DCA::Capsule, 39](#)
  - [DCA::Primitive, 72](#)
  - [DCA::Rectangle, 78](#)
  - [DCA::Sphere, 94](#)
- [compute\\_d2PdSdT](#)
  - [DCA::Box, 33](#)
  - [DCA::Capsule, 39](#)
  - [DCA::Primitive, 72](#)
  - [DCA::Rectangle, 79](#)
  - [DCA::Sphere, 94](#)
- [compute\\_dDdS](#)
  - [DCA::API, 26, 28](#)
  - [DCA::Solver, 89](#)
- [compute\\_dPdS](#)
  - [DCA::Box, 32](#)
  - [DCA::Capsule, 38](#)
  - [DCA::Primitive, 71](#)
  - [DCA::Rectangle, 77](#)
  - [DCA::Sphere, 93](#)
- [compute\\_dPdT](#)
  - [DCA::Box, 32](#)
  - [DCA::Capsule, 38](#)
  - [DCA::Primitive, 71](#)
  - [DCA::Rectangle, 78](#)
  - [DCA::Sphere, 93](#)
- [compute\\_dtds](#)
  - [DCA::Solver, 88](#)
- [compute\\_O](#)

- DCA::Objective, 63
- compute\_P
  - DCA::Box, 32
  - DCA::Capsule, 37
  - DCA::Primitive, 71
  - DCA::Rectangle, 77
  - DCA::Sphere, 92
- compute\_p2DpS2
  - DCA::Objective, 62
- compute\_p2DpT2
  - DCA::Objective, 62
- compute\_p2DpTpS
  - DCA::Objective, 62
- compute\_p2OpS2
  - DCA::Objective, 64
- compute\_p2OpT2
  - DCA::Objective, 64
- compute\_p2OpTpS
  - DCA::Objective, 64
- compute\_pDpS
  - DCA::Objective, 61
- compute\_pDpT
  - DCA::Objective, 61
- compute\_pOpS
  - DCA::Objective, 63
- compute\_pOpT
  - DCA::Objective, 63
- compute\_Ps
  - DCA::Objective, 60
- compute\_t
  - DCA::API, 29
  - DCA::Solver, 87
- computeDerivative
  - DCA::SoftUnilateralConstraint, 84
- computeSecondDerivative
  - DCA::SoftUnilateralConstraint, 84
- constraintWeight
  - DCA::Objective, 67
- CYAN
  - DCA::Logger, 50
- d2
  - DCA::SoftUnilateralConstraint, 85
- DCA, 21
  - Matrix12d, 23
  - pair\_t, 23
  - primitive\_t, 22
  - uint, 22
  - Vector0d, 23
  - Vector12d, 23
  - Vector1d, 23
  - Vector6d, 23
- DCA::API, 25
  - compute\_closest\_points, 27, 29
  - compute\_D, 26, 27
  - compute\_d2DdS2, 26, 28
  - compute\_dDdS, 26, 28
  - compute\_t, 29
- DCA::Box, 30
  - ~Box, 31
  - Box, 31
  - center, 35
  - compute\_d2PdS2, 33
  - compute\_d2PdSdT, 33
  - compute\_dPdS, 32
  - compute\_dPdT, 32
  - compute\_P, 32
  - dimensions, 35
  - get\_largest\_dimension\_from\_center, 34
  - get\_s, 34
  - getLocalVectors, 35
  - orientation, 35
  - SIZE\_T, 34
- DCA::Capsule, 36
  - ~Capsule, 37
  - Capsule, 37
  - compute\_d2PdS2, 39
  - compute\_d2PdSdT, 39
  - compute\_dPdS, 38
  - compute\_dPdT, 38
  - compute\_P, 37
  - endPosition, 41
  - get\_largest\_dimension\_from\_center, 40
  - get\_s, 40
  - SIZE\_T, 40
  - startPosition, 41
- DCA::ExpCoords, 41
  - \_PI, 48
  - get\_ddR\_i\_j, 44
  - get\_ddwdr\_dr1, 46
  - get\_ddwdr\_dr2, 46
  - get\_ddwdr\_dr3, 46
  - get\_dRi, 44
  - get\_dThetadRi, 43
  - get\_dwdr, 45
  - get\_R, 43
  - get\_theta, 42
  - get\_w, 45
  - getSkewSymmetricMatrix, 47
  - safeACOS, 47
  - safeDACOS, 48
  - tol, 48
- DCA::FiniteDifference, 48
- DCA::Logger, 49
  - BLUE, 50
  - CYAN, 50
  - DEFAULT, 50
  - GREEN, 50
  - MAGENTA, 50
  - print, 50
  - PRINT\_COLOR, 49
  - RED, 50
  - YELLOW, 50
- DCA::NeighborsPairGenerator, 50
  - estimate\_distance, 52
  - generate, 51, 52
  - m\_radius, 53

- NeighborsPairGenerator, 51
- DCA::NewtonOptimizer, 53
  - ~NewtonOptimizer, 54
  - checkHessianRank, 56
  - checkLinearSystemSolve, 56
  - gradient, 57
  - hessian, 57
  - lineSearchStartValue, 55
  - maxLineSearchIterations, 55
  - name, 55
  - NewtonOptimizer, 54
  - objectiveValue, 56
  - optimize, 54
  - printInfo, 56
  - searchDir, 57
  - solverResidual, 55
  - t\_tmp, 56
  - useDynamicRegularization, 56
- DCA::Objective, 57
  - ~Objective, 60
  - A, 59
  - B, 59
  - check\_inputs, 66
  - compute\_D, 60
  - compute\_O, 63
  - compute\_p2DpS2, 62
  - compute\_p2DpT2, 62
  - compute\_p2DpTpS, 62
  - compute\_p2OpS2, 64
  - compute\_p2OpT2, 64
  - compute\_p2OpTpS, 64
  - compute\_pDpS, 61
  - compute\_pDpT, 61
  - compute\_pOpS, 63
  - compute\_pOpT, 63
  - compute\_Ps, 60
  - constraintWeight, 67
  - get\_primitives\_description, 65
  - get\_s, 66
  - get\_sizes\_t, 65
  - get\_t, 66
  - get\_total\_size\_t, 65
  - Objective, 60
  - PRIMITIVE, 59
  - primitive\_A, 67
  - primitive\_B, 67
  - regularizerWeight, 67
  - unilateralConstraint, 67
- DCA::PermutationPairGenerator, 68
  - generate, 68
- DCA::Primitive, 69
  - ~Primitive, 70
  - check\_t, 74
  - compute\_d2PdS2, 72
  - compute\_d2PdSdT, 72
  - compute\_dPdS, 71
  - compute\_dPdT, 71
  - compute\_P, 71
  - get\_center\_point, 73
  - get\_largest\_dimension\_from\_center, 74
  - get\_s, 73
  - Primitive, 70
  - safetyMargin, 74
  - SIZE\_T, 73
- DCA::Rectangle, 75
  - ~Rectangle, 76
  - center, 80
  - compute\_d2PdS2, 78
  - compute\_d2PdSdT, 79
  - compute\_dPdS, 77
  - compute\_dPdT, 78
  - compute\_P, 77
  - dimensions, 81
  - get\_largest\_dimension\_from\_center, 80
  - get\_s, 79
  - getLocalVectors, 80
  - orientation, 81
  - Rectangle, 76
  - SIZE\_T, 79
- DCA::SoftUnilateralConstraint, 81
  - ~SoftUnilateralConstraint, 82
  - a1, 85
  - a2, 85
  - b1, 85
  - b2, 85
  - c1, 85
  - c2, 85
  - computeDerivative, 84
  - computeSecondDerivative, 84
  - d2, 85
  - epsilon, 86
  - evaluate, 84
  - limit, 86
  - setEpsilon, 83
  - setLimit, 83
  - setStiffness, 83
  - SoftUnilateralConstraint, 82
- DCA::Solver, 86
  - ~Solver, 87
  - are\_t\_and\_s\_in\_sync, 90
  - compute\_closest\_points, 89
  - compute\_D, 88
  - compute\_d2DdS2, 89
  - compute\_dDdS, 89
  - compute\_dtds, 88
  - compute\_t, 87
  - get\_s\_from\_primitives, 90
  - get\_total\_size\_t, 90
  - objective, 91
  - optimizer, 91
  - Solver, 87
- DCA::Sphere, 91
  - ~Sphere, 92
  - compute\_d2PdS2, 94
  - compute\_d2PdSdT, 94
  - compute\_dPdS, 93

- compute\_dPdT, 93
- compute\_P, 92
- get\_largest\_dimension\_from\_center, 95
- get\_s, 95
- position, 96
- SIZE\_T, 95
- Sphere, 92
- ddR.h, 97
- ddR\_0\_0
  - Partial Derivatives of Rotation Matrices, 14
- ddR\_0\_1
  - Partial Derivatives of Rotation Matrices, 14
- ddR\_0\_2
  - Partial Derivatives of Rotation Matrices, 14
- ddR\_1\_0
  - Partial Derivatives of Rotation Matrices, 15
- ddR\_1\_1
  - Partial Derivatives of Rotation Matrices, 15
- ddR\_1\_2
  - Partial Derivatives of Rotation Matrices, 15
- ddR\_2\_0
  - Partial Derivatives of Rotation Matrices, 17
- ddR\_2\_1
  - Partial Derivatives of Rotation Matrices, 17
- ddR\_2\_2
  - Partial Derivatives of Rotation Matrices, 17
- DEFAULT
  - DCA::Logger, 50
- dimensions
  - DCA::Box, 35
  - DCA::Rectangle, 81
- endPosition
  - DCA::Capsule, 41
- epsilon
  - DCA::SoftUnilateralConstraint, 86
- estimate\_distance
  - DCA::NeighborsPairGenerator, 52
- evaluate
  - DCA::SoftUnilateralConstraint, 84
- ExpCoords.h, 98
- FiniteDifference.h, 98
- generate
  - DCA::NeighborsPairGenerator, 51, 52
  - DCA::PermutationPairGenerator, 68
- get\_center\_point
  - DCA::Primitive, 73
- get\_ddR\_i\_j
  - DCA::ExpCoords, 44
- get\_ddwdr\_dr1
  - DCA::ExpCoords, 46
- get\_ddwdr\_dr2
  - DCA::ExpCoords, 46
- get\_ddwdr\_dr3
  - DCA::ExpCoords, 46
- get\_dRi
  - DCA::ExpCoords, 44
- get\_dThetadRi
  - DCA::ExpCoords, 43
- get\_dwdr
  - DCA::ExpCoords, 45
- get\_largest\_dimension\_from\_center
  - DCA::Box, 34
  - DCA::Capsule, 40
  - DCA::Primitive, 74
  - DCA::Rectangle, 80
  - DCA::Sphere, 95
- get\_primitives\_description
  - DCA::Objective, 65
- get\_R
  - DCA::ExpCoords, 43
- get\_s
  - DCA::Box, 34
  - DCA::Capsule, 40
  - DCA::Objective, 66
  - DCA::Primitive, 73
  - DCA::Rectangle, 79
  - DCA::Sphere, 95
- get\_s\_from\_primitives
  - DCA::Solver, 90
- get\_sizes\_t
  - DCA::Objective, 65
- get\_t
  - DCA::Objective, 66
- get\_theta
  - DCA::ExpCoords, 42
- get\_total\_size\_t
  - DCA::Objective, 65
  - DCA::Solver, 90
- get\_w
  - DCA::ExpCoords, 45
- getLocalVectors
  - DCA::Box, 35
  - DCA::Rectangle, 80
- getSkewSymmetricMatrix
  - DCA::ExpCoords, 47
- gradient
  - DCA::NewtonOptimizer, 57
- GREEN
  - DCA::Logger, 50
- hessian
  - DCA::NewtonOptimizer, 57
- limit
  - DCA::SoftUnilateralConstraint, 86
- lineSearchStartValue
  - DCA::NewtonOptimizer, 55
- Logger.h, 99
- m\_radius
  - DCA::NeighborsPairGenerator, 53
- MAGENTA
  - DCA::Logger, 50
- Matrix12d
  - DCA, 23

- maxLineSearchIterations
  - DCA::NewtonOptimizer, [55](#)
- name
  - DCA::NewtonOptimizer, [55](#)
- NeighborsPairGenerator
  - DCA::NeighborsPairGenerator, [51](#)
- NewtonOptimizer
  - DCA::NewtonOptimizer, [54](#)
- NewtonOptimizer.h, [99](#)
- Objective
  - DCA::Objective, [60](#)
- objective
  - DCA::Solver, [91](#)
- Objective.h, [100](#)
- objectiveValue
  - DCA::NewtonOptimizer, [56](#)
- optimize
  - DCA::NewtonOptimizer, [54](#)
- optimizer
  - DCA::Solver, [91](#)
- orientation
  - DCA::Box, [35](#)
  - DCA::Rectangle, [81](#)
- Pair.h, [100](#)
- pair\_t
  - DCA, [23](#)
- Partial Derivatives of Rotation Matrices, [13](#)
  - ddR\_0\_0, [14](#)
  - ddR\_0\_1, [14](#)
  - ddR\_0\_2, [14](#)
  - ddR\_1\_0, [15](#)
  - ddR\_1\_1, [15](#)
  - ddR\_1\_2, [15](#)
  - ddR\_2\_0, [17](#)
  - ddR\_2\_1, [17](#)
  - ddR\_2\_2, [17](#)
- position
  - DCA::Sphere, [96](#)
- PRIMITIVE
  - DCA::Objective, [59](#)
- Primitive
  - DCA::Primitive, [70](#)
- primitive\_A
  - DCA::Objective, [67](#)
- primitive\_B
  - DCA::Objective, [67](#)
- primitive\_t
  - DCA, [22](#)
- Primitives.h, [100](#)
- print
  - DCA::Logger, [50](#)
- PRINT\_COLOR
  - DCA::Logger, [49](#)
- printlnInfo
  - DCA::NewtonOptimizer, [56](#)
- README.md, [101](#)
- Rectangle
  - DCA::Rectangle, [76](#)
- RED
  - DCA::Logger, [50](#)
- regularizerWeight
  - DCA::Objective, [67](#)
- safeACOS
  - DCA::ExpCoords, [47](#)
- safeDACOS
  - DCA::ExpCoords, [48](#)
- safetyMargin
  - DCA::Primitive, [74](#)
- searchDir
  - DCA::NewtonOptimizer, [57](#)
- setEpsilon
  - DCA::SoftUnilateralConstraint, [83](#)
- setLimit
  - DCA::SoftUnilateralConstraint, [83](#)
- setStiffness
  - DCA::SoftUnilateralConstraint, [83](#)
- SIZE\_T
  - DCA::Box, [34](#)
  - DCA::Capsule, [40](#)
  - DCA::Primitive, [73](#)
  - DCA::Rectangle, [79](#)
  - DCA::Sphere, [95](#)
- SoftUnilateralConstraint
  - DCA::SoftUnilateralConstraint, [82](#)
- SoftUnilateralConstraint.h, [101](#)
- Solver
  - DCA::Solver, [87](#)
- Solver.h, [101](#)
- solverResidual
  - DCA::NewtonOptimizer, [55](#)
- Sphere
  - DCA::Sphere, [92](#)
- startPosition
  - DCA::Capsule, [41](#)
- t\_tmp
  - DCA::NewtonOptimizer, [56](#)
- tol
  - DCA::ExpCoords, [48](#)
- uint
  - DCA, [22](#)
- unilateralConstraint
  - DCA::Objective, [67](#)
- useDynamicRegularization
  - DCA::NewtonOptimizer, [56](#)
- Utils.h, [102](#)
- Vector0d
  - DCA, [23](#)
- Vector12d
  - DCA, [23](#)
- Vector1d

DCA, [23](#)

Vector6d

DCA, [23](#)

YELLOW

DCA::Logger, [50](#)