CDPR Analysis in CASPR

1 Analysis in CASPR

CASPR supports a number of different analysis techniques to be conducted on cable robots including

- Inverse Kinematics
- Forward Kinematics
- Forward Dynamics
- Inverse Dynamics
- Control
- Workspace Analysis
- Design Optimisation

The table below provides a brief summary of the implementations for each of these analysis problems. The sections that follow then provide a more detailed description of how to perform each of these analyses for a single point in CASPR. In order to perform these analyses over a trajectory and/or set of points please refer to the documentation for `Simulators`.

Analysis Problem	**Folder** (from ∼/src/)	**Abstract Classes**	**Evaluation Functions**
Inverse Kinematics	Model	N/A	SystemModel.update
Forward Kinematics	Analysis/ForwardKinematics	FKAnalysisBase	compute
Forward Dynamics	Analysis/ForwardDynamics	N/A	ForwardDynamics.compute
Inverse Dynamics	Analysis/InverseDynamics	IDSolverBase	resolve
Control	Analysis/Control	ControllerBase	execute
Workspace Analysis	Analysis/Workspace	WorkspaceConditionBase WorkspaceMetricBase	evaluate evaluate
Design Optimisation	Analysis/DesignOptimisation	CableAttachmentOptimisationFnBase AttachmentPointParamBase	evaluate update

Table 1: Summary of Implementation for the Different Analysis Problems

2 Inverse Kinematics

In cable robotics analysis, the inverse kinematics (IK) problem consists of the problem of determining the cable lengths $\mathbf{l}$ given the joint positions $\mathbf{q}$ where these two variables are related to one another by the inverse kinematics mapping

$$\mathbf{l} = \mathbf{g}(\mathbf{q}). \tag{1}$$

In CDPR analysis, the cable lengths $\mathbf{l}$ are often assumed to be known uniquely provided that the base and rigid link attachments locations are known with respect to the inertial base frame. Since these attachment locations are themselves uniquely determined as a function of the joint pose $\mathbf{q}$, the mapping $\mathbf{g}$ is generally a known many to one mapping. This means that solving the IK problem is typically trivial when compared to other CDPR analysis problems such as forward kinematics and inverse dynamics.

2.1 CASPR Inverse Kinematics Implementation

In CASPR, the IK problem is solved for all model types using the approach proposed in [1]. Specifically, CASPR solves the inverse kinematics problem within the `update(...)` functions of the classes contained within the ∼`/src/Model/Cables` directory. These methods are called when the `update(...)` function of the `SystemModel` (∼`/src/Model/SystemModel.m`) object is called. The resulting length vector $\mathbf{l}$ and its derivative $\dot{\mathbf{l}}$ are stored within the variables `cableLengths` and `cableLengthsDot`, respectively. These variables can be accessed directory from the `SystemModel` class.

3 Forward Kinematics

The forward kinematics (FK) problem looks to determine the CDPR pose $\mathbf{q}$ for a given set of cable lengths $\mathbf{l}$. As a result FK analysis looks to determine the solution to the inverse of (1). Since the mapping from $\mathbf{l}$ to $\mathbf{q}$ is not unique, the FK problem represents an area of active research interest. In its current state, CASPR has two different implemented forward kinematics solvers from the literature [2, 3]. CASPR also supports the addition of new FK solvers which can be added by implementing the forward kinematics base class.

3.1 Base Forward Kinematics Solver

In CASPR, different forward kinematics solvers can be added through the implementation of the abstract `FKAnalysisBase` (∼/`src/Analysis/ForwardKinematics/FKAnalysisBase.m`) class. Code Sample 1 shows the `FKAnalysisBase` class. It can be seen that this class consists of the property `model` which represents the model of the system as well as the protected properties `q_previous` and `l_previous` which act as internal variables which can used if necessary by forward kinematic solvers.

Code Sample 1: FKAnalysisBase Class.

```
classdef FKAnalysisBase < handle
    properties
        model;      % The model of the system
    end

    properties (SetAccess = protected, GetAccess = protected)
        q_previous = []     % The previous joint positions
        l_previous = []     % The previous cable lengths
    end

    methods
        % Constructor for forward kinematics objects
        function fk = FKAnalysisBase(kin_model)
            fk.model = kin_model;
        end

        % Computes the joint position information given the cable
        % information.
        function [q, q_dot, comp_time] = compute(obj, len, len_prev_2, q_prev, q_d_prev,
            delta_t)
            start_tic = tic;
            [q, q_dot] = obj.computeFunction(len, len_prev_2, q_prev, q_d_prev, delta_t);
            comp_time = toc(start_tic);
        end
    end

    methods (Abstract)
        % An abstract function for computation.
        [q,q_dot] = computeFunction(obj, len, len_prev_2, q_prev, q_d_prev, delta_t);
    end

    methods (Static)
        % Computation of the length error as a 1 norm.
        function [errorVector, jacobian] = ComputeLengthErrorVector(q, l, model)
            model.update(q, zeros(model.numDofs,1), zeros(model.numDofs,1),zeros(model.
                numDofs,1));
            errorVector = l - model.cableLengths;
            jacobian = - model.L;
        end
    end
end
```

From Code Sample 1 it can also be seen that the class possesses the methods `compute(...)`, `computeFunction(...)` and `ComputeLengthErrorVector(...)`. The method `computeFunction(...)` is an abstract method whose implementation corresponds to the different possible forward kinematics solvers. The method `compute(...)` then acts as a wrapper for this method by storing the computational time information. Finally the method `ComputeLengthErrorVector(...)` represents an additional function for measuring errors, which different solvers may make use of.

3.2 Forward Kinematics Implementations

Two different forward kinematics implementations are provided in CASPR (within the folder ∼/`src/Analysis/ForwardKinematics`): the Jacobian pseudoinverse integration method [2] and the non-linear

least squares method [3]. The Jacobian pseudoinverse integration method is written in the class `FKDifferential`. This method computes the FK solution via numerical integration of $\dot{\mathbf{q}}$ which is determined as

$$\dot{\mathbf{q}} = L^{\dagger}\dot{\mathbf{l}}, \tag{2}$$

where † denotes the Moore-Penrose pseudo-inverse.

The non-linear least squares method is written in the class `FKLeastSquares`. In this class, the FK problem is solved as a least square problem using the error computed from the function `ComputeLengthErrorVector(...)`.

4 Forward Dynamics

The forward dynamics (FD) problem for CDPRs at time $t = t_k$ is to determine $\mathbf{q}(t_k)$ and $\dot{\mathbf{q}}(t_k)$ given knowledge of the cable forces $\mathbf{f}(t_k)$. One typical means with which to solve the FD problem is through the use of numerical double integration. These methods solve the FD problem by integrating the joint acceleration $\ddot{\mathbf{q}}$, which can be determined using the equation of motion expression

$$\ddot{\mathbf{q}}(t_k) = -M^{-1}\left(L^T\mathbf{f} - \mathbf{C} - \mathbf{G} - \mathbf{w_{ext}}\right). \tag{3}$$

4.1 CASPR Forward Dynamics Solver

In CASPR, the FD problem is solved using the numerical double integration and (3). This procedure is implemented in the class `ForwardDynamics` which is located in the `~/src/Analysis/ForwardDynamics` directory. Within the `ForwardDynamics` class, the FD problem is solved within the operation `compute(...)`. This operation takes in a given MATLAB numerical integrator (identified by the class `FDSolverType`) and then integrates the defined `eom` function which is equivalent to (3).

5 Inverse Dynamics

The inverse dynamics for CDPRs refers to the resolution of a set of positive cable forces $\mathbf{f}$ to achieve a desired joint space motion described by $\mathbf{q}$, $\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$. The resulting cable forces must satisfy the equation of motion

$$M(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}}) + \mathbf{G}(\mathbf{q} + \mathbf{w_{ext}} = -L(\mathbf{q})^T\mathbf{f}, \tag{4}$$

as well as any other constraints that are acting on the system.

For completely and redundantly restrained systems, there exists a greater number of cables than the number of degrees of freedom of the system. Hence there may exist infinitely many cable force solutions to the inverse dynamics problem. As such, the resolution of positive cable forces can be formulated as an optimisation problem to resolve the redundancy. This redundancy resolution takes the form

$$\begin{aligned} \mathbf{f}^* = & \underset{\mathbf{f_{min}} \leq \mathbf{f} \leq \mathbf{f_{max}}}{\arg\min} \quad Q(\mathbf{f}) \\ s.t.\ & M(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}}) + \mathbf{G}(\mathbf{q} + \mathbf{w_{ext}} = -L(\mathbf{q})^T\mathbf{f}, \\ & C(\mathbf{q}, \dot{\mathbf{q}}, \mathbf{f}) \leq \mathbf{0}. \end{aligned} \tag{5}$$

where $\mathbf{f}^*$ represents the optimum cable force solution, $Q(\mathbf{f})$ represents the objective function and $C(\mathbf{q}, \dot{\mathbf{q}}, \mathbf{f})$ represents any additional system constraints.

5.1 Base Inverse Dynamics Solver

In CASPR, different inverse dynamics solvers can be added through the implementation of the abstract `IDSolverBase` (∼/`src/Analysis/InverseDynamics/IDSolverBase.m`) class. Code Sample 2 shows the `IDSolverBase` class. It can be seen that this class consists of the property `model` which represents the dynamic model of the system and the protected properties `f_previous` and `active_set` which act as internal variables used by some inverse dynamics solvers.

Code Sample 2: IDSolverBase Class.

```
classdef IDSolverBase < handle
    properties
        model           % The model of the system
    end

    properties (SetAccess = protected, GetAccess = protected)
        f_previous = [] % The previous instance of cable forces
        active_set = [] % The previous active set for optimisation based methods
    end

    methods
        % The constructor for the class.
        function id = IDSolverBase(dyn_model)
            id.model = dyn_model;
        end

        % Resolves the system kinematics into the next set of cable forces.
        function [cable_forces, Q_opt, id_exit_type, comp_time, model] = resolve(obj, q, q_d,
             q_dd, w_ext)
            obj.model.update(q, q_d, q_dd, w_ext);
            start_tic = tic;
            [obj.model.cableForces, Q_opt, id_exit_type] = obj.resolveFunction(obj.model);
            comp_time = toc(start_tic);
            model = obj.model;
            cable_forces = obj.model.cableForces;
        end
    end

    methods (Abstract)
        % The abstract resolution function to be implemented by concrete
        % implementations of this base class.
        [cable_forces,Q_opt, id_exit_type] = resolveFunction(obj, dynamics);
    end

    methods (Static)
        % The equation of motion constraints in linear terms.
        function [A, b] = GetEoMConstraints(dynamics)
            A = -dynamics.L';
            b = dynamics.M*dynamics.q_ddot + dynamics.C + dynamics.G + dynamics.W_e;
        end
```

```
    end
end
```

From Code Sample 2 it can also be seen that the class possesses the methods `resolve(...)`, `resolveFunction(...)` and `GetEOMConstraints(...)`. The method `resolveFunction` is an abstract method whose implementation allows for different ID method to be added to CASPR. The method `resolve` then acts as a wrapper for this method by updating the internal dynamic model and storing the computational time information. Finally the method `GetEOMConstraints` converts the equation of motion constraint (4) into the linear constraint form

$$A\mathbf{f} \leq \mathbf{b}. \tag{6}$$

This form of constraint is more suitable for many optimisation based solvers.

5.2 Inverse Dynamics Implementations

A number of different inverse dynamics solvers have been implemented into CASPR (in the directory `~/src/Analysis/InverseDynamics/Solvers`). Broadly speaking these solvers can be broken into two categories: *optimisation based solvers* which explicitly solve a problem in the form of (5) and *miscellaneous* solvers which resolve the redundancy using algorithms which do not make explicit use of generic optimisation tools.

5.2.1 Miscellaneous Solvers

Table 2 lists the different miscellaneous inverse dynamics solvers implemented and any restrictions that the implemented solver is subject to.

Solve Name	Restrictions	Cite
`IDSolverOptimallySafe`		
∞-norm implementation	N/A	[4]
null space implementation	N/A	[4]
`IDSolverFeasiblePolygon`		
1-norm implementation	m = n+2	[5]
2-norm implementation	m = n+2	[5]
centroid implementation	m = n+2	[5]
`IDSolverClosedForm`		
Closed Form Method	may not find solution	[6]
Puncture Method	may not find solution	[7]
Improved Closed Form Method	may not find solution	[8]
Improved Puncture Method	may not find solution	[7]

Table 2: Miscellaneous Solver Table

5.2.2 Optimisation Based Solvers

In CASPR, the implemented optimisation based solvers consider the 1,2 and ∞ norm minimisation problems subject to linear constraints. The different possible cost functions and constraints are provided in the `~/src/Analysis/InverseDynamics/Formulation` directory.

Table 3 lists the different optimisation based inverse dynamics solvers considered as well as the different formulations and optimisation tool implementations.

Solve Type	Formulations	Implementations
`IDSolverLinProg`	- `IDObjectiveMinLinCableForce`	- MATLAB linprog - OptiToolbox LP_SOLVE
`IDSolverQuadProg`	- `IDObjectiveMinQuadCableForce` - `IDObjectiveMinInteractions` - `IDObjectiveQuadOptimallySafe`	- MATLAB quadprog - MATLAB quadprog warm start - OptiToolbox IPOPT - OptiToolbox OOQP
`IDSolverMinInfNorm`	- `IDObjectiveMinInfCableForce` - `IDObjectiveInfOptimallySafe`	- MATLAB linprog - OptiToolbox LP_SOLVE

Table 3: Optimisation Based Solver Table

6 Control

The control problem for CDPRs refers to the resolution of a set of positive cable forces $\mathbf{f}$ in order to track a desired joint space motion described by $\mathbf{q}_{ref}$, $\dot{\mathbf{q}}_{ref}$ and $\ddot{\mathbf{q}}_{ref}$ given the current joint space dynamics $\mathbf{q}$, $\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$. The resulting cable force must therefore satisfy the relationship

$$\tau_{control} = -L(\mathbf{q})^T \mathbf{f}, \tag{7}$$

where $\tau_{control}$ is the control force of an equivalent fully actuated system, as well as any other constraints that are acting on the system.

6.1 Controller Base

In CASPR, different controllers can be added for simulation through the implementation of the abstract `ControllerBase` (∼/`src/Control/ControllerBase.m`) class. Code Sample 3 shows the `ControllerBase` class. It can be seen that this class consists of the property `dynModel` which represents the dynamic model of the system and the abstract method `evaluateFunction` which takes in the simulated mechanism motion (`q`, `q_d` and `q_dd`) as well as the reference mechanism motion (`q_ref`, `q_ref_d` and `q_ref_dd`) and returns the necessary cable forces (`cable_forces`) for the mechanism.

Code Sample 3: ControllerBase Class.

```
classdef ControllerBase < handle
    properties
        dynModel            % The model of the system
    end

    methods
        % A constructor for the controller base.
        function cb = ControllerBase(dyn_model)
            cb.dynModel = dyn_model;
        end

    end

    methods (Abstract)
        % An abstract executeFunction for all controllers. This should take
        % in the generalised coordinate information and produces a control
        % input.
        [cable_forces] = executeFunction(obj, q, q_d, q_dd, q_ref, q_ref_d, q_ref_dd,t);
    end

end
```

6.2 Controller Implementations

A number of different CDPR controllers have been implemented into CASPR. Table 4 lists the different controllers implemented, the tuning parameters of the controllers, and the form of the implementation for the `executeFunction`.

Folder Name	Parameters	executeFunction	cite
ComputedTorque	`id_solver,Kp,Kd`	$\mathbf{e} = K_p(\mathbf{q_{ref}} - \mathbf{q}) + K_d(\dot{\mathbf{q}}_{\mathbf{ref}} - \dot{\mathbf{q}})$ $\mathbf{f} =$ `id_solver.resolve`$(\mathbf{q}, \mathbf{0}, \mathbf{0}, \mathbf{e})$	[9]
LyapunovStaticCompensation	`id_solver,Kp,Kd`	$\ddot{\mathbf{q}}_{\mathbf{cmd}} = \ddot{\mathbf{q}}_{\mathbf{cmd}} K_p(\mathbf{q_{ref}} - \mathbf{q}) + K_d(\dot{\mathbf{q}}_{\mathbf{ref}} - \dot{\mathbf{q}})$ $\mathbf{f} =$ `id_solver.resolve`$(\mathbf{q}, \mathbf{0}, \ddot{\mathbf{q}}_{\mathbf{cmd}}, \mathbf{0})$	[9]

Table 4: Implemented Controller Table

7 Workspace Analysis

Workspace analysis evaluates the capability of a mechanism to produce a desired set of wrenches/accelerations at given poses. Due to the actuation redundancy and positive force constraints, CDPR workspace analysis is typically conducted at a kinetics level. Workspace analysis can be performed as both a boolean evaluation of a given condition (such as wrench closure [10, 11], wrench feasibility [12] etc.) or as a quantitative measure of CDPR capability through the use of workspace metrics. CASPR supports both of these approaches through operations conducted on the class `WorkspacePoint` (∼/`src/Analysis/Workspace/WorkspacePoint.m`).

7.1 Workspace Point Class

Code Sample 4 shows the `WorkspacePoint` class (∼/`src/Analysis/Workspace/WorkspacePoint`). It can be seen that this CASPR class acts as a container class which holds the properties `pose`, `metrics` and `conditions`. The property `pose` stores the information about the pose at which the workspace conditions/metrics are to be evaluated at. The properties `metrics` and `conditions` stores lists corresponding to an identifier for each metric and condition that has been evaluated at the given workspace pose as well as the value obtained through that evaluation.

Code Sample 4: WorkspacePoint Class.

```
classdef WorkspacePoint < handle
    properties(SetAccess = protected)
        pose        % The pose for the workspace condition to be evaluated at
        metrics     % A cell array of different metrics (enum and value)
        conditions  % A cell array of different workspace conditions (enum and value)
    end

    methods
        % Constructor for the class
        function wp = WorkspacePoint(pose,n_metrics,n_constraints)
            wp.pose         = pose;
            wp.metrics      = cell(n_metrics,2);
            wp.conditions   = cell(n_constraints,2);
        end

        % A function to add the metric information to the point
        function addMetric(obj,metric_type,metric_value,index)
            obj.metrics{index,1} = metric_type;
            obj.metrics{index,2} = metric_value;
        end

        % A function to add a new condition to the point
        function addCondition(obj,condition_type,index)
            obj.conditions{index,1} = condition_type;
            obj.conditions{index,2} = true;
        end
    end
end
```

It can also be seen that the class consists of two methods `addMetric` and`addCondition`. These two methods both act to facilitate the addition of new workspace evaluations to the workspace point.

7.2 Workspace Conditions

Workspace conditions refer to a binary requirement that defines the capability of a CDPR to produce a set of desired wrenches and/or kinematics. In CASPR, the abstract class `WorkspaceConditionBase` (∼/`src/Workspace/Conditions/WorkspaceConditionBase.m`) is provided in order for different workspace conditions to be inherited. Code Sample 5 shows the contents of the `WorkspaceConditionBase` class. It can be seen that the class possesses two properties: `method` and `type`. The property `type` represents an enum which identifies the particular workspace condition that is being evaluated while `method` then denotes the method of evaluation for that condition that is being used.

Code Sample 5: Base Workspace Condition Class.

```
classdef WorkspaceConditionBase < handle
    properties
        method          % Method of implementation (an enum)
        type            % Type of joint from JointType enum
    end

    methods
```

```
        % The implementation of evaluate. This evaluates the object
        % dynamics to determine if the workspace condition is satisfied.
        function [condition_type, condition_value, comp_time] = evaluate(obj,dynamics,
            workspace_point)
            start_tic       = tic;
            condition_value = obj.evaluateFunction(dynamics, workspace_point);
            condition_type  = obj.type;
            comp_time = toc(start_tic);
        end
    end

    methods (Abstract)
        % evalute - This function takes in the workspace dynamics and
        % returns a boolean for whether the point lies in the workspace.
        f = evaluateFunction(obj, dynamics, workspace_point);
    end

    methods (Static)
        % Creates a new condition
        function wc = CreateWorkspaceCondition(conditionType,method,desired_set)
            switch conditionType
                case WorkspaceConditionType.WRENCH_CLOSURE
                    wc = WrenchClosure(method);
                case WorkspaceConditionType.WRENCH_FEASIBLE
                    wc = WrenchFeasible(method,desired_set);
                case WorkspaceConditionType.STATIC
                    wc = WorkspaceStatic(method);
                otherwise
                    error('Workspace metric type is not defined');
            end
            wc.type = conditionType;
        end
    end
end
```

From Code Sample 5 it can also be observed that the `WorkspaceConditionBase` class possesses a number of different methods. To produce a new workspace condition it is necessary that the abstract method `evaluteFunction` which evaluates the workspace condition is implemented. Furthermore the user also needs to add an enum for the condition into the class `WorkspaceConditionType` and then use that enum in the object creation function `CreateWorkspaceCondition`.

As an additional note, new methods can also be applied to existing workspace conditions in CASPR. To do this the user needs to simply add the written evaluation to the associated methods enum class for the particular condition and then add that function call into the condition's `evaluateFunction` implementation.

The following conditions (and implementations in the citations) have currently been implemented in CASPR: `StaticWorkspace` [12, 13], `WrenchClosure` [10–12, 14] and `WrenchFeasible` [12].

7.3 Workspace Metrics

Compared with workspace conditions which are boolean, workspace metrics provide a measure workspace quality at different CDPR poses. These qualities can be useful in the design and synthesis of CDPRs. Like the workspace conditions, the abstract class `WorkspaceMetricBase` (∼/`src/Workspace/Metrics/WorkspaceMetricBase.m`) is provided for different workspace metrics to be inherited. Code Sample 6 shows the `WorkspaceMetricBase` class. It can be seen that this class is fundamentally similar to that of the `WorkspaceConditionBase` class. However, the class has the additional properties `metricMin` and `metricMax` which represent the boundary values for the metric.

Code Sample 6: Base Workspace Metric Class.

```
classdef (Abstract) WorkspaceMetricBase < handle
    properties
        type            % Type of joint from JointType enum
    end

    properties (SetAccess = protected)
        % Minimum and maximum allowable metric values
        metricMin
        metricMax
    end

    methods
        % Evalute function returns a quantitative evaluation of a metric
        % given dynamics information
```

```
        function [metric_type,metric_value, comp_time] = evaluate(obj,dynamics,options)
            start_tic    =    tic;
            metric_type =    obj.type;
            f = obj.evaluateFunction(dynamics,options);
            if(f<obj.metricMin)
                metric_value = obj.metricMin;
            elseif(f>obj.metricMax)
                metric_value = obj.metricMax;
            else
                metric_value = f;
            end
            comp_time = toc(start_tic);
        end

        % Overrides the metricMin and MetricMax values
        function setMetricLimits(obj,metric_min,metric_max)
            assert(metric_max >= metric_min,'Maximum must be greater than minimum');
            obj.metricMin = metric_min;
            obj.metricMax = metric_max;
        end
    end

    methods (Abstract)
        % evalute - This function takes in the workspace dynamics and
        % returns the metric value
        f = evaluateFunction(obj,dynamics,options);
    end

    methods (Static)
        % Creates a new metric
        function wm = CreateWorkspaceMetric(metricType,desired_set)
            switch metricType
                case WorkspaceMetricType.SEACM
                    wm = SEACM;
                case WorkspaceMetricType.CAPACITY_MARGIN
                    wm = CapacityMarginMetric(desired_set);
                case WorkspaceMetricType.CONDITION_NUMBER
                    wm = ConditionNumberMetric;
                case WorkspaceMetricType.TENSION_FACTOR
                    wm = TensionFactorMetric;
                case WorkspaceMetricType.TENSION_FACTOR_MODIFIED
                    wm = TensionFactorModifiedMetric;
                case WorkspaceMetricType.UNILATERAL_DEXTERITY
                    wm = UnilateralDexterityMetric;
                otherwise
                    error('Workspace metric type is not defined');
            end
            wm.type = metricType;
        end
    end
end
```

Currently, the following metrics have been implemented:

- `ConditionNumberMetric`: The standard condition number metric used as a measure of dexterity for serial and parallel manipulators.
- `CapacityMarginMetric`: The robustness of the CDPR in producing a given workspace condition defined by the minimum signed distance between the available wrench set and the desired wrench set [15]
- `TensionFactorMetric`: A measure of the relative tension distribution for the cables which can evaluate the quality of wrench closure at a specific pose [10]
- `TensionFactorModifiedMetric`: A modification of the tension factor metric to incorporate singular values.
- `UnilateralDexterityMetric`: An approximation of the dexterity measure for CDPRs taking into account the positive cable force constraint [16]

8 Design Optimisation

To be added.

References

[1] D. Lau, D. Oetomo, and S. K. Halgamuge, "Generalized modeling of multilink cable-driven manipulators with arbitrary routing using the cable-routing matrix," *IEEE Trans. Robot.*, vol. 29, no. 5, pp. 1102–1113, 2013.

[2] T. Bruckmann, L. Mikelsons, T. Brandt, M. Hiller, and D. Schramm, "Wire robots part I: Kinematics, analysis & design," in *Parallel Manipulators New Developments*, ser. ARS Robotic Books. I-Tech Education and Publishing, 2008.

[3] A. Pott and V. Schmidt, "On the forward kinematics of cable-driven parallel robots," in *Proc. IEEE/RSJ Int. Conf. Intell. Robot. Syst.*, 2015, pp. 3182–3187.

[4] P. H. Borgstrom, B. L. Jordan, G. S. Sukhatme, M. A. Batalin, and W. J. Kaiser, "Rapid computation of optimally safe tension distributions for parallel cable-driven robots," *IEEE Transactions on Robotics*, vol. 25, no. 6, pp. 1271–1281, 2009.

[5] M. Gouttefarde, J. Lamaury, C. Reichert, and T. Bruckmann, "A versatile tension distribution algorithm for n-DOF parallel robots driven by n+2 cables," *IEEE Trans. Robot.*, vol. 31, no. 6, pp. 1444–1457, 2015.

[6] A. Pott, T. Bruckmann, and L. Mikelsons, "Closed-form force distribution for parallel wire robots," in *Computational Kinematics*. Springer Netherlands, 2009, pp. 25–34.

[7] K. Müller, C. Reichert, and T. Bruckmann, "Analysis of a real-time capable cable force computation method," in *Cable-Driven Parallel Robots*, ser. Mechanisms and Machine Science, A. Pott and T. Bruckmann, Eds. Springer International Publishing, 2015, vol. 32, pp. 227–238.

[8] A. Pott, "An improved force distribution algorithm for over-constrained cable-driven parallel robots," in *Computational Kinematics*. Springer Netherlands, 2014, pp. 139–146.

[9] A. B. Alp and S. K. Agrawal, "Cable suspended robots: Design, planning and control," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2002, pp. 4275–4280.

[10] C. B. Pham, S. H. Yeo, G. Yang, and I.-M. Chen, "Workspace analysis of fully restrained cable-driven manipulators," *Robot. Auton. Syst.*, vol. 57, no. 9, pp. 901–912, 2009.

[11] B. Ouyang and W.-W. Shang, "A new computation method for the force-closure workspace of cable-driven parallel manipulators," *Robotica*, vol. 33, no. 3, pp. 537–547, 2015.

[12] A. L. C. Ruiz, S. Caro, P. Cardou, and F. Guay, "ARACHNIS: Analysis of robots actuated by cables with handy and neat interface software," in *Cable-Driven Parallel Robots*, ser. Mechanisms and Machine Science, A. Pott and T. Bruckmann, Eds. Springer International Publishing, vol. 32, pp. 293–305.

[13] D. Lau, J. Eden, D. Oetomo, and S. K. Halgamuge, "Musculoskeletal static workspace of the human shoulder as a cable-driven robot," *IEEE/ASME Trans. Mechatronics*, vol. 20, no. 2, pp. 978–984, 2015.

[14] W. B. Lim, G. Yang, S. H. Yeo, and S. K. Mustafa, "A generic force-closure analysis algorithm for cable-driven parallel manipulators," *Mech. Mach. Theory*, vol. 46, no. 9, pp. 1265–1275, 2011.

[15] F. Guay, P. Cardou, A. L. Cruz-Ruiz, and S. Caro, "Measuring how well a structure supports varying external wrenches," in *New Advances in Mechanisms, Transmissions and Applications: Proceedings of the Second Conference MeTrApp 2013*, V. Petuya, C. Pinto, and E.-C. Lovasz, Eds. Springer Netherlands, vol. 12, pp. 385–392.

[16] R. Kurtz and V. Hayward, "Dexterity measures with unilateral actuation constraints: the n+1 case," *Adv. Robot.*, vol. 9, no. 5, pp. 561–577, 1994.