Gradient-Based Path Optimization

June 20, 2018

Contents

1 Assumptions

The path optimizer reduces the length of a given path with some assumptions:

- The cost is defined in a vectorial space (i.e., rotations must be defined by bounded rotations, SO_3 joint can not be directly used). This provides a quadratic cost.
- The cost can be weighted by the proportionality of the initial segments lengths (see weighted cost definition). You can force to one weights in the code to cancel weighting.
- we do not have convergence proof of solving a quadratic cost under non-linear constraints.

2 Collision Constraints

2.1 Notations

$c(x)$: cost x : path defined by the waypoints.
$\mathbf{p}$: gradient direction of iteration (obtained with BFGS algorithm $p = -H^{-1}\nabla c(x)$).
α : tuning parameter.
J_f : collision constraints jacobian

2.2 Constraints

Affine constraints:
Every path apply previous constraints.
x_{2i} : paths on which computed constraints are backtracked.
x_{2i+1} : paths with high probability of collisions on which J_f is computed.
Newton algorithm : $x_{2i+1} = x_{2i} + \alpha_{2i}\mathbf{p_{2i}}$

Constraint definition : $f(x_{i+2}) = f(x_i)$

$$
\begin{aligned}
f(x_{i+2}) - f(x_{i+1}) &= f(x_i) - f(x_{i+1}) && (1)\\
\frac{\partial f(x_{i+1})}{\partial x}(x_{i+2} - x_{i+1}) &= -\frac{\partial f(x_i)}{\partial x}(x_{i+1} - x_i) && (2)\\
J_{f_i}(x_{i+2} - x_{i+1}) &= J_{f_i}(x_{i+1} - x_i) && (3)\\
\alpha_{i+1} J_{f_i}\mathbf{p_{i+1}} &= -\alpha_i J_{f_i}\mathbf{p_i} && (4)\\
& && (5)
\end{aligned}
$$

Because the constraint (jacobian) is computed only once every two iterations,

$$\frac{\partial f(x_{i+1})}{\partial x} = \frac{\partial f(x_i)}{\partial x} = J_{f_i}$$

So the affine constraint can be globally written as:

$$J_{f_i}\mathbf{p} = b$$

with the second term (which is zero for a linear constraint)

$$b = \frac{\alpha}{\alpha_{previous}} J_f \mathbf{p_{previous}}$$

That formula avoids to directly compute the constraint ($f(q_{Constr}) = b$) using forward kinematics, all terms are already available.

$\mathbf{p_{previous}}$ depends how affine constraints are used. Section **??** presents an example of "'compute and apply constraint one step over two", so $\mathbf{p_{previous}}$ is basically the previous performed step.

2.3 Example of algorithm using naive step projection on affine constraints null space

Projection of $\mathbf{p_{i+1}}$ direction:

$$\overline{\mathbf{p_{i+1}}} = (I - J_{f_i}^{+} J_{f_i})\mathbf{p_{i+1}} - \frac{\alpha_i}{\alpha_{i+1}} J_{f_i}^{+} J_{f_i} \mathbf{p_i}$$

x_0 : paths with no constraint and collision-free.

$x_1 = x_0 + \alpha_0 \mathbf{p_0}$: path with collision q_{Coll_0} and no constraint, computes future constraint q_{Constr_0}, add it to J_f.

x_2 : path which project the gradient descent $\mathbf{p_1}$ to comply with the constraint q_{Constr_0}. Works also with several constraints added at the same time.
$x_2 = x_1 + \alpha_1 \overline{\mathbf{p_1}}$
$\overline{\mathbf{p_1}} = (I - J_f^{+} J_f)\mathbf{p_1} - \frac{\alpha_0}{\alpha_1} J_f^{+} J_f \mathbf{p_0}$

x_3 : path which project the gradient descent $\mathbf{p_2}$ to comply with the constraint q_{Constr_0}, in collision q_{Coll_1}, so create q_{Constr_1} and update J_f.
$x_3 = x_2 + \alpha_2 \overline{\mathbf{p_2}}$
$\overline{\mathbf{p_2}} = (I - J_f^{+} J_f)\mathbf{p_2}$
No affine term since x_2 already applied the constraint.

x_4 : path which project the gradient descent $\mathbf{p_3}$ to comply with the constraints q_{Constr_0} and q_{Constr_1}.
$x_4 = x_3 + \alpha_3 \overline{\mathbf{p_3}}$
$\overline{\mathbf{p_3}} = (I - J_f^{+} J_f)\mathbf{p_3} - \frac{\alpha_2}{\alpha_3} J_f^{+} J_f \mathbf{p_2}$

Problem encountered : we will get the solution of the constraint sub-space that is orthogonal to $-H^{-1}\nabla c(x)$ but it will not minimize the quadratic cost. This is why the section **??** is rather solving a QP under constraints.

2.4 Explaination of the step projection on the constraints null space

Reminder of the problem. The quadratic cost on a path x can be written as:

$$c(x) = \frac{1}{2} x^T H_i x - g_i^T \mathbf{p_i} + cste$$

But since collision-constrainted are defined for $\mathbf{p_i}$ step, the cost becomes:

$$\tilde{c}(\mathbf{p_i}) = \frac{1}{2} \mathbf{p_i}^T H_i \mathbf{p_i} - \tilde{g_i}^T \mathbf{p_i} + \tilde{cste}$$

H_i is the estimated Hessian at i^{th} step, found by the BFGS approximation:

$$H_{i+1} = H_i (I - \frac{\mathbf{p_i} \mathbf{p_i}^T H_i}{\mathbf{p_i}^T H_i \mathbf{p_i}}) + \frac{y_i y_i^T}{y_i^T \alpha_i \mathbf{p_i}}$$

with $y_i = \nabla c(x_{i+1}) - \nabla c(x_i)$.
The second term of the cost ${\tilde{g}_i}^T$ can be computed thanks to an initial condition:

$$\frac{\partial \tilde{c}}{\partial \mathbf{p}}(\mathbf{p_i}) = H\mathbf{p_i} - \tilde{g}_i$$

$$\frac{\partial \tilde{c}}{\partial \mathbf{p}}(0) = \frac{\partial c}{\partial x}(x_0) = \nabla c(x_0)$$

$$\tilde{g}_i = -\nabla c(x_0)$$

So, when constraints are found, the QP is expressed as follows:

$$\min_{\mathbf{p_i}} \tilde{c}(\mathbf{p_i}) \quad \text{such that} \quad J_{f_i}\,\mathbf{p_i} = b_i$$

using the singular value decomposition of J_{f_i}

$$J_{f_i} = \begin{pmatrix} U_1 & U_0 \end{pmatrix} \Sigma \begin{pmatrix} V_1 & V_0 \end{pmatrix}^T$$

we get a parameterization of the affine sub-space defined by the constraint:

$$\mathbf{p_i} = J_{f_i}^{+} b_i + V_0 \mathbf{z} \quad \mathbf{z} \in \mathbb{R}^{n - rank(J_{f_i})}$$

Solving the constrained QP consists in finding $\mathbf{z}$ that minimizes

$$\begin{aligned}
& \frac{1}{2}(J_{f_i}^{+} b_i + V_0\mathbf{z})^T H_i (J_{f_i}^{+} b_i + V_0\mathbf{z}) - {\tilde{g}_i}^T (J_{f_i}^{+} b_i + V_0\mathbf{z}) \\
= & \frac{1}{2}\mathbf{z}^T V_0^T H_i V_0 \mathbf{z} + (J_{f_i}^{+} b_i)^T H_i V_0 \mathbf{z} - {\tilde{g}_i}^T V_0 \mathbf{z} + Cste \\
= & \frac{1}{2}\mathbf{z}^T V_0^T H_i V_0 \mathbf{z} + (V_0^T H_i J_{f_i}^{+} b_i - V_0^T \tilde{g}_i)^T \mathbf{z} + Cste
\end{aligned}$$

The value of $\mathbf{z}$ that minimizes the above expression is given by

$$\mathbf{z_i}^* = (V_0^T H_i V_0)^{-1} (V_0^T \tilde{g}_i - V_0^T H_i J_{f_i}^{+} b_i)$$

which can be directly computed with the LLT library of Eigen.

Then, the found step is use in the descent algorithm:

$$x_1 = x_0 + \alpha \mathbf{p_i}$$

α parameter is handled by the following finite state machine fig. **??**. Globally, α is only changed when a collision is detected. If α was previously non-equal to 1 then it is set to 1 to directly converge to the solution of the QP. If α was equal to 1 then it is set to an arbitrary value < 1, typically 0.5 or 0.2 (slower convergence, less risks).

A different behaviour can be implemented if we use the α-tuning heuristic (be-gining from 0.5) while no-collision is detected. If a collision is detected but the QP solution is not collision-free, α will be reset to its initial value, and not the last value returned by the heuristic.

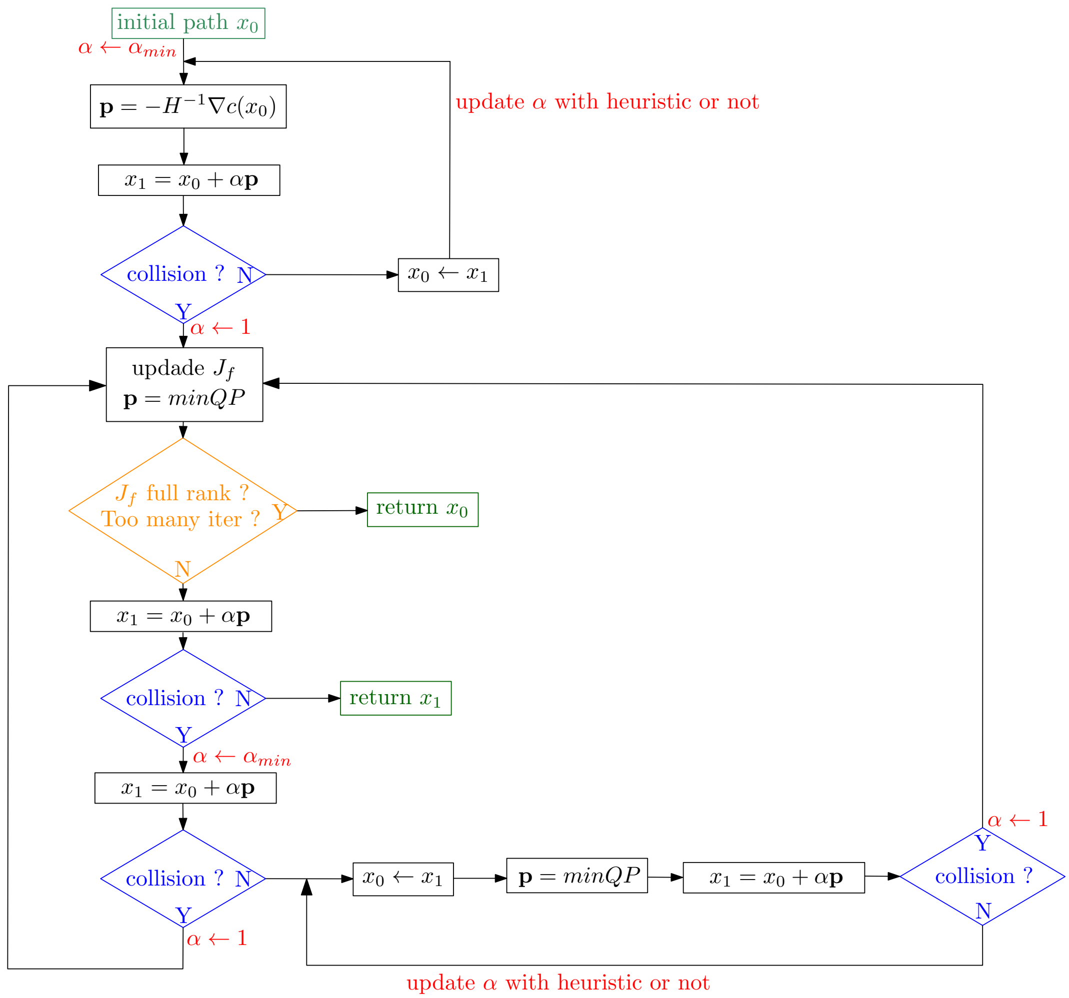

Figure 1: Algorithm and alpha tuning. Number of algorithm iterations is in-creasing for each collision-test box. When reaching the command "update Jf", collision information are gathered and constraints are computed, next paths will apply these constraints.

3 Weighted Cost

3.1 New cost definition

Notation of length in configuration space: i is the path (or iteration) index, k is the waypoint (or sub-path) index. n is the number of waypoints.

$$L_{k,i} = ||q_{k+1,i} - q_{k,i}||$$

Previously, we used the following quadratic cost:

$$c(x_i) = \frac{1}{2}\sum_{k=0}^{n} L_{k,i}^2$$

Problem: reducing this cost is reducing the length of the path BUT also equidistantly allocating the waypoints. This represents a problem for long trajectories with local very-constrained paths as the puzzle and the 2D path passing through a box.

Therefore, we can use a weighted cost, according to the initial segments lengths :

$$\forall k \in [0, n], \quad w_k = \frac{1}{L_{k,0}}$$

$$c(x_i) = \frac{1}{2}\sum_{k=0}^{n} w_k L_{k,i}^2$$

This cost has to propriety to conservate the proportionality (Fig. **??**) between of each segment length over total length between initial path x_0 and the ideal path without obstacle (so when the weighted gradient is zero) x_{min}.
The proportionality can be written as:

$$\forall k \in [0, n], \quad L_{k,min} = \frac{L_{min}}{L_0} L_{k,0}$$

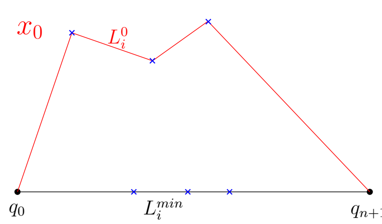

Figure 2: Illustration of an optimized path obtained by weighted cost. Note the conservation of lengths proportionalites.

3.2 Gradient implementation

Lets $k \in [0; n]$ represents the waypoints index including initial and final configurations. Implemented previously:

$$u1 = \begin{pmatrix} \vdots \\ q_{k+1} - q_k \\ \vdots \end{pmatrix}^{k \in [0;n-1]}$$

$$u2 = \begin{pmatrix} \vdots \\ q_{k+2} - q_{k+1} \\ \vdots \end{pmatrix}^{k \in [0;n-1]}$$

$$\nabla c(x) = u1 - u2 \ \in \mathbb{R}^n$$

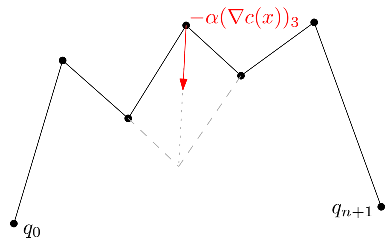

Figure 3: Illustration (without weights) of the third component of the path gradient.

Now with weights (with the same notations):

$$\nabla c(x) = \begin{pmatrix} w_0(q_1 - q_0) - w_1(q_2 - q_1) \\ \vdots \\ w_{k-1}(q_k - q_{k-1}) - w_k(q_{k+1} - q_k) \\ \vdots \\ w_{n-1}(q_n - q_{n-1}) - w_n(q_{n+1} - q_n) \end{pmatrix}^{k \in [1;n]}$$

Note: for the ideal path where all points are aligned, the gradient is zero. Therefore:

$$\forall k \in [1, n], \quad w_{k-1} L_{k-1}^{min} = w_k L_k^{min}$$

$$\forall k \in [1, n], \quad w_{k-1} \frac{L^{min}}{L^0} L_{k-1}^0 = w_k \frac{L^{min}}{L^0} L_k^0$$

And finally the equation that defines the cost weights:

$$\forall k \in [1, n], \quad w_{k-1}\, L_{k-1}^0 = w_k\, L_k^0$$

3.3 Hessian implementation

$$H(x) = \frac{\partial(\nabla c(x))}{\partial q}$$

In practice:

$$\frac{\partial term_i}{\partial q_j} \begin{cases} w_{i-1} + w_i & \text{if } j = i \\ -w_i & \text{if } j = i+1 \\ -w_{i+1} & \text{if } j = i-1 \end{cases} \tag{6}$$

So finally:

$$H(x) = \begin{pmatrix} (w_0+w_1)W & -w_1\,W & 0 & & \\ -w_1\,W & (w_1+w_2)W & -w_2\,W & 0 & \\ 0 & -w_2 W & (w_2+w_3)W & -w_3\,W & 0 \\ & & & \ddots & \\ & & 0 & -w_{n-1}\,W & (w_{n-1}+w_n)W \end{pmatrix}$$

Where W is a diagonal matrix made of the robot weights.

3.4 Example of results in 2D

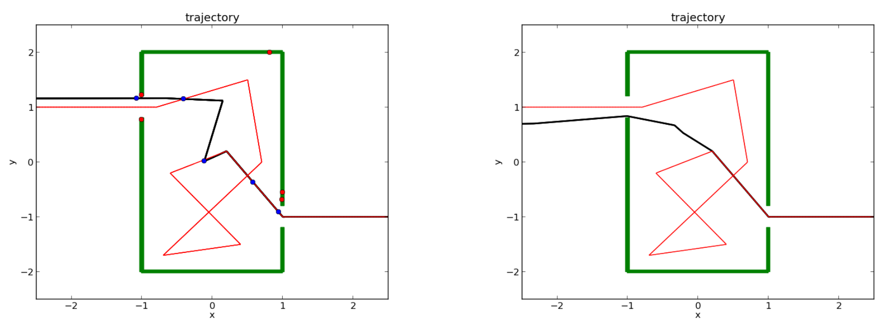

Table 1: Result for a very long path from $(-100;1)$ to $(100;-1)$ which Random Shortcut fails to optimize. Left: for $i_{max}=30$ iterations (local min obtained for 41 iterations exactly). Right: for $i_{max}=30$ iterations + second run of 14 iterations based on the first result. So we see the interest of relaxing/canceling old constraints...

4 Work in progress

4.1 Affine constraints to compensate linearization error

In fact, for most of problems, the constraint function $f(q)$ is not linear. So during linearization on f constraint submanifold to compute the constraint jacobian J_f a residual mistake may be encountered during the next iteration, if we re-check if the previous constraint is well-applied (Fig. **??**). We can also write this error as:

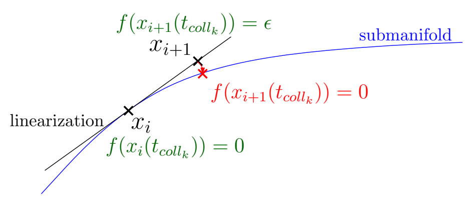

Figure 4: Linearization error.

$$f(x_0(t_{coll_k})) = 0 \text{ path on which constraints are computed}$$

$$f(x_1(t_{coll_k})) = \epsilon \text{ path on which constraints are applied}$$

with t_{coll_k} the instant when collision occurs in the k^{th} segment of the path.

But this approximation can be tolerated if there is no risk of collision. This criterion is obtained comparing the minimal distance $\mathscr{L}_d$ between the bodies that have been in collision at configuration $x_1(t_{coll_k})$ to the local range $\mathscr{R}$ of the constraint application error (similar to continuous collision checking):

$$f(x_1(t_{coll_k})) = \begin{pmatrix} t(M^{-1}\, M^*) \\ log(R^T\, R^*) \end{pmatrix}$$

$$\mathscr{R} = r\,||f(x_1(t_{coll_k}))\,[3:6]|| + ||f(x_1(t_{coll_k}))\,[0:3]||$$

where r is the `radius` of the joint, determined from the Device (see Fig. **??** for notations).

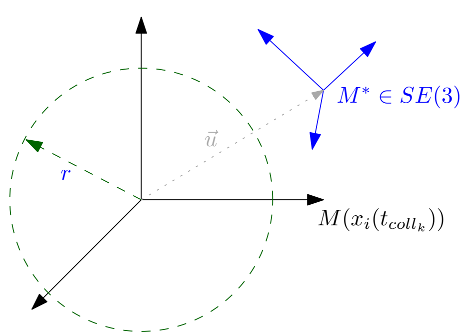

Figure 5: Collision-constraints frames notations.

Given a constraint f linearized around x_i:

$$f(x) = f(x_i) + J_f(x_i)\,(x_i) + o(x - x_i) \overset{\text{desired}}{=} 0$$

which gives (since $f(x_i) = 0$):

$$J_f(x_i)\,(x - x_i) \overset{\text{desired}}{=} 0$$

So we find our previous approach $J_f\,\mathbf{p_i} = 0$.

In case we want to re-linearize this constraint around an other path x_j:

$$f(x) = f(x_j) + J_f(x_j)\,(x_j) + o(x - x_j) \overset{\text{desired}}{=} 0$$

which gives

$$J_f(x_j)\,(x - x_j) \overset{\text{desired}}{=} -f(x_j)$$

So to re-linearize our constraint, we have to re-compute the constraint Jacobian on x_j and to introduce and affine term $-f(x_j)$.

The following algorithm presents what is implemented as linear-error-compensation in the path-optimizer. As soon as a collision-constraint exists,COMPENSATION is called after integrating the step and before testing path for collisions.

Algorithm 1 Raw version to summarize implemented work

1: **procedure** COMPENSATION
2:     a m^{th} constraint exists at subpath rank k and $t_{coll_{k,m}}$ parameter
3:     x_i is collision-free
4:     $x_{i+1} \leftarrow x_i + \alpha\mathbf{p_i}$
5:     $linErrorActive \leftarrow False$
6:     **for** $m \in collisionConstraintsNumber$ **do**
7:         $q_{collConstr_m} \leftarrow x_{i+1}(t_{coll_{k,m}})$
8:         $\mathscr{R} \leftarrow r\,||f(q_{collConstr_m})\,[3:6]|| + ||f(q_{collConstr_m})\,[0:3]||$
9:         $\mathscr{L}_d \leftarrow \text{minDistance}(objects, q_{collConstr_m})$
10:         **if** $\mathscr{R} > \mathscr{L}_d$ **and** $alpha! = 1$ **then**
11:             $linErrorActive \leftarrow True$
12:             A risk of collision due to linearization error exists !
13:             Re-linearize f_m around $q_{collConstr_m}$
14:             $J_{f_m}(x_{i+1})(x_{i+1}) = -f_m(x_{i+1})$
15:             Replace J_{f_m} in J_f and f_m in b
16:     **end for**
17:     **if** $linErrorActive$ **and** $isValid(x1)$ **then**
18:         svd(J_f, b)
19:         (J_f is further used to compute $\mathbf{p}$)
20:         $addConstraints \leftarrow False$
21: **end procedure**

Notes about the algorithm:

- (cf. test line 10): if x_{i+1} (applying collision constraints) is computed with $\alpha = 1$ and is not collision free, we choose to not waste time re-linearizing using x_{i+1}, because this path is 'far' from the path that will be computed on the next step with α_{init} (when the $minQP$ is not collision-free, otherwise the algorithm would be over whenever a need of compensation would have occured).

- (cf line 20): it is difficult to modify J_f at the same time with collision constraints and with re-linearisation. So whenever x_{i+1} is in collision, and as soon as it was not computed with $\alpha = 1$, linear compensation can be applicated **and** we will ignore x_{i+1}'s collision (because they could be caused by the linearization error itself!).

  [Possible improvement] Test if the part k of the path which is in collision is the same as the part on which compensation(s) occur(s). If not, we can re-linearize and add a new collision-constraint during the same path-iteration.

- $addConstraint$ is a boolean tested when a collision is found on x_{i+1} to determine if the associated constraint can be added. In the global algorithm figure **??**, we can notice that $\alpha = 1$ is also preventing from adding a new collision-constraint.

5 Possible future work

5.1 Constraints relaxation

In this part, some ideas for relaxing constraints are proposed.
In the case constraints are not directly canceled, but a compromise is found:

- Allow moving waypoints 'connected' to some constraints to *move* into the constraint direction ?
- Add waypoints near a constraint to improve path-mobility ?

In the case we allow to cancel 'old' constraints First.

- Define iteration threshold i_{reset} from which constraints will be canceled, based on an event such as:
  - when improvement is not efficient anymore (`norm(s)` $< \epsilon$), which constraints are been canceled ? All ?
  - when algo tries to add existing constraints: on same localPath (`colRank`), same objects are in collision (with similar t_{coll_k} ?) Accepting how many constr before saying 'no it is too similar' ? In this case, delete the old similar constraints
- Or when the optimum under constraints has been reached, can try once more clearing all constraints (as when we relaunch manually). i_{max} becomes the new criterion: number of tries to optimize path before give up
- Use distances between bodies (`get_dist`) ? May not be relevant, 'cf potential' example: an annoying constraint is added at the bottom, but near an obstacle; so if we use a criterion 'not cancel a constraint **near** an obstacle', this constraint will not be removed...

A completeness study should be provided when choosing beteween these ideas.

5.2 Add problem constraints

Initially, the planner provides a path containing waypoints and constraints. (Between two constrained waypoints, the interpolation does not garantee the constraint application for the whole path)

Basic sketch work:

- Remove initial path constraints and test if collision free. Depending on the answer, we will have to optimize keeping the constraints with the path or not.

- Applies a local linearisation of the PB constraints on all waypoints: filling diagonally the Jacobian constraints J_- with other problem-constraints Jacobians J_{PB}.

$$J_- = \begin{pmatrix} J_{PB} & & & 0 \\ & J_{PB} & & \\ & & \ddots & \\ 0 & & & J_{PB} \end{pmatrix}$$

Collision constraints J_f will be simply added *under* this matrix $\begin{pmatrix} J_- \\ J_f \end{pmatrix}$

$\rightarrow$ `implemented in 'GradientBased::getProblemConstraints'`

- Project obtained path (waypoints) on PB constraints tangent space to correct waypoints (`ConfigProjector->projectOnKernel ()`). And then compute collisions...

- If the PB constraints have been removed from the path in the first step, add them, project and test path. If in collision, stuck.

5.3 Optimization on $SO(3)$ Lie group

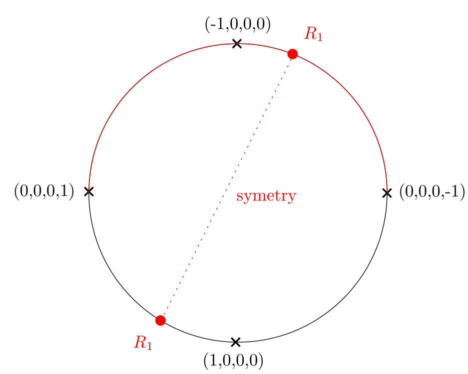

Figure 6: Representation of a subpart of $SO(3)$ when only ω_z is varying.

QP resolution falling into local minimum of quaternion sphere (negative real part which corresponds to the same rotation).
See Fig. **??**

Scalar product between $\mathbf{p}\nabla c(x)$ may be < 0?

The gradient vanishes on a maximum (of length) instead a min?

5.4 Waypoints prunning ?

Kineo-like prunning method. Before calling optimizer ? But reduces optimization quality (less waypoints = less DoF for our problem, even if less time consuming)