Rapid Path-Planning for Unstable Concentric Tube Robot Guidance

Konrad Leibrandt, Christos Bergeles, *Member, IEEE*, and Guang-Zhong Yang, *Fellow, IEEE*

***Abstract*— The complex, non-intuitive kinematics of concentric tube robots can make their telemanipulation challenging. Collaborative control schemes that guide the operating clinician via repulsive and attractive force feedback based on intraoperative path-planning can simplify this task. Computationally efficient algorithms, however, are required to perform rapid path-planning and to solve the inverse kinematics of the robot at interactive rates. Until now, ensuring stable and collision-free robot configurations required long periods of pre-computation to establish kinematic look-up tables. This paper presents a high-performance robot kinematics software architecture, which is used together with a multi-node computational framework to rapidly calculate dense path-plans for safe telemanipulation of unstable concentric tube robots. The proposed software architecture enables on-the-fly incremental inverse-kinematics estimation at interactive rates, and is tailored to modern computing architectures with efficient multi-core CPUs. The effectiveness of the architecture is quantified with computational-complexity metrics, and in a clinically demanding simulation inspired from neurosurgery for hydrocephalus treatment. By achieving real-time path-planning we can generate active constraints on-the-fly and support the operator in faster and more reliable execution of telemanipulation tasks.**

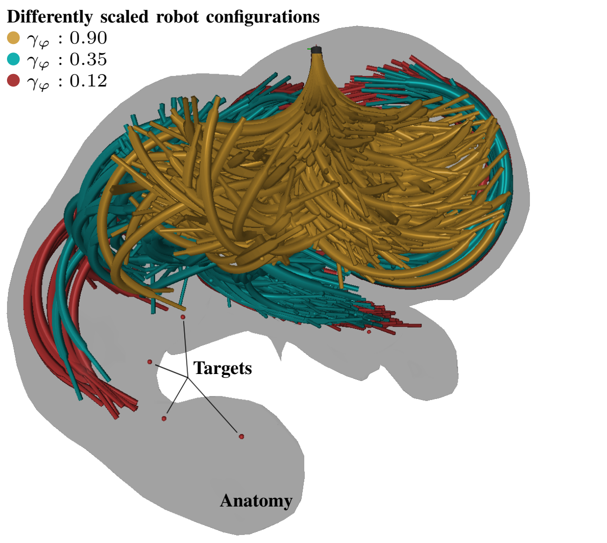
Differently scaled robot configurations
● γ_φ : 0.90
● γ_φ : 0.35
● γ_φ : 0.12
Targets
Anatomy

Fig. 1. Concentric tube robots with different translation scaling. Each colour consists of 1000 robot configurations with random joint values.

I. Introduction

Accessing pathologies through natural orifices or single ports for minimally invasive surgery (MIS) is a challenging task well suited for continuum robots as their shape flexibility allows them to controllably conform to the traversed anatomy [3]. Concentric tube robots (CTRs) represent an example of continuum robots, and entail the translation and rotation of pre-curved super-elastic tubes that are concentrically arranged [4]. The pre-curved tubes mutually interact to create a final robot shape, allowing CTR to exhibit full tip-pose control. As the number of tubes of the robot increases, its redundant configuration space can be exploited to simultaneously achieve shape control and tip manipulation. By controlling the number and mechanical properties of tubes, CTRs can be deployed for a variety of surgical applications, such as haemorrhage evacuation [5], transurethral prostatectomy [6], tissue approximation for heart surgery [7] and bronchoscopic biopsy [8]. Computational design of CTR can deliver optimal robot architectures for different surgical tasks [7], [9]. The computational requirements for safe inverse kinematics increase with the number of robot tubes that these tailored robots possess, supporting the need for quick and efficient solvers.

Detailed models based on tube mechanics have been developed for the forward and inverse kinematics of CTRs. The most advanced models account for the torsional wind-up along the length of each tube, as well as the instability of configurations (equilibrium conformations) arising from torsion [10], [11]. Energy accumulated through the torsional wind-up of the tubes can rapidly release in certain tube configurations and cause uncontrollable motion [11], [12]. Therefore, unstable configurations need to be actively avoided. However, their detection incurs additional computational cost and constraints on the inverse kinematics.

Real-time, safe and effective path-planning for custom-designed CTRs has several benefits. First, it empowers the operator with a reliable telemanipulation approach, ensuring avoidance of anatomical collisions and dangerous unstable configurations [1], [7], [13], [14], [15]. Second, if real-time performance is guaranteed, it is possible to custom-make concentric tubes for specific patients using rapid workspace optimisation with minimal time delay. Especially in critical cases such as ruptured aneurysms [5], a rapid plan-to-deployment period can significantly reduce intra-operative complication and potential morbidity. Finally, real-time path-

K. Leibrandt and G.-Z. Yang are with the Hamlyn Centre for Robotic Surgery, Imperial College London, United Kingdom. C. Bergeles is with the Translational Imaging Group, Centre for Medical Image Computing, University College London, United Kingdom. K. Leibrandt was supported by the President's PhD Scholarship of Imperial College London and EPSRC UK. C. Bergeles was supported by an EPSRC-funded UCL Future Leaders Grant. Corresponding author: `k.leibrandt12@imperial.ac.uk`.

A preliminary version of this paper was presented at IEEE/RSJ Int. Conf. Intelligent Robots and Systems 2015, 2016 [1] and [2]. The current manuscript expands our work, contains a new in-depth description of the software framework, and additional metrics. This research was demonstrated at the 2016 Surgical Robot Challenge, held during the 9th Hamlyn Symposium on Medical Robotics, London, UK.

planning schemes can be coupled with active constraints (AC)/virtual fixtures (VF) [16] to form cooperative robotic telemanipulation controllers wherein the operator is gently steered towards the estimated safe paths. In this shared-control approach, the complexity of inverse kinematics is invisible to the operator, allowing seamless avoidance of collisions and unstable configurations while retaining overall control over the approach path to the anatomical targets.

Fast solvers for robot inverse kinematics are a primary building block towards assisted telemanipulation of CTRs. To this end, researchers have achieved real-time kinematics performance by creating look-up tables and employing root-finding approaches to estimate the joint values for a desired pose of the tip [10]. Others have investigated Jacobian-based approximations [17] to speed-up computations. Pre-computation of dense path-plans within the robot workspace has been proposed by [18], while sparse path-plans and random trees have been proposed in [14], [19].

Most commonly, the solvers do not consider stability and prune the robot-design space to avoid unstable robots. This approach, however, limits the combinations of tubes that can be used, despite [7] having demonstrated that unstable CTRs may well be the only robots that can perform certain interventions. In addition, pre-computation of path-plans assumes a static anatomy and complicates the intraoperative adaptation of the said plans when there is anatomy motion. Finally, pre-computation for achieving interactive-rate kinematics solution adds many hours of overhead in computational requirements - even when instability is not considered. Only faster approaches to inverse kinematics estimation can deliver the fundamental building blocks for safe and effective CTR telemanipulation in dynamic environments. Our proposed approach is well suited to tackle these challenges.

Our work expands on the algorithms of [1], [2]. The proposed software architecture takes advantage of distributed computing resources and multi-core CPUs to provide globally dense workspaces (robot task space) within minutes. By creating memory-optimal data structures to represent CTRs and the inverse kinematics problems, the developed infrastructure can, in real-time, leverage the dense workspaces for on-demand path-planning with on-line local inverse kinematics to guide the user along safe trajectories. Further, our research contributes to Implicit Active Constraints (IAC) for CTRs by proposing constraint generation based on the generated path-plans. Contrary to ACs, which are manually defined, IACs are algorithmically generated and tailored to the respective task based on limited user input [20]. As such, IACs are both more complex to define and to estimate intraoperatively, further increasing the benefit of a software architecture that enables real-time safe and effective robot guidance. Further expanding upon the work in the literature, the developed software requires minimal pre-computation, and is quick and reliable for incorporation in telemanipulation control schemes of CTRs. The computational benefits are supported by extensive quantitative evaluation. User-based evaluation of the guidance scheme is carried out through a study simulating a challenging clinical scenario from neurosurgery.

II. Robot Kinematics And Constraints

The sections of CTRs can be of variable curvature (VC) or fixed curvature (FC) [10]. A VC section consists of two tubes, which rotate individually but translate in tandem. Therefore, a VC section has 3 degrees-of-freedom (DoF). An FC section is a single tube and has 2 DoF. The i^{th} tube's translation is denoted as ${}_{i}\varphi$, and its base rotation as ${}_{i}\alpha^{\mathrm{B}}$. The joint-space $\boldsymbol{q}$ of a CTR is therefore described by the set of $\left\{{}_{i}\varphi,{}_{i}\,\alpha^{\mathrm{B}}\right\}\ i \in [1, N_{\mathrm{t}}]$, where N_{t} is the number of tubes.

This work estimates the shape of the CTR based on the unloaded torsionally compliant kinematics model [4], [10]. Estimating the robot shape relies on computationally demanding iterative solving of a boundary value problem. Beginning at a torsion-free tip angle ${}_{i}\alpha^{\mathrm{T}}$, differential equations are iteratively solved via Euler approximation (discretisation step ϵ_{arc}) to obtain the base angle ${}_{i}\alpha^{\mathrm{B}}$ for each tube. Subsequently, the shape of the robot is calculated using matrix exponentials and the curvature along the robot's centre line.

A. Anatomical Constraints

CTRs are often deployed within delicate anatomy, making it important to avoid contact with the anatomy (denoted as Γ). Rasterized polygons, extracted from pre-operatively acquired images, provide the representation of Γ as a set of 3D points. Rasterisation occurs with a maximal lattice of ϵ_{lat} and the points are stored in a k-d tree (3-d tree). This k-d tree allows for rapid distance queries between the robot shape and Γ. Each point along the robot centreline is associated with the maximum tube diameter ${}_{i_{\mathrm{cl}}}r^{t}$ at that robot location (the external surface of the robot). The k-d tree is queried to determine the nearest-neighbour ${}_{i_{\mathrm{cl}}}\boldsymbol{d}^{\mathrm{nn}}$ of every robot point to Γ. Given the tube radii, ${}_{i_{\mathrm{cl}}}r^{t}$, the discretisation step, ϵ_{arc}, and the mesh lattice, ϵ_{lat}, the distance to the anatomy, d_{ana}, is:

$$d_{\mathrm{ana}} = \min_{i_{\mathrm{cl}}}\left({}_{i_{\mathrm{cl}}}\boldsymbol{d}^{\mathrm{nn}} - \left\|\frac{1}{2}[\epsilon_{\mathrm{arc}},\epsilon_{\mathrm{lat}}]^{T}\right\|_{2} - {}_{i_{\mathrm{cl}}}r^{t}\right) \tag{1}$$

A robot is colliding with the anatomy *iff* $d_{\mathrm{ana}} < 0$.

B. Stability Constraints

Anatomical constraints are complemented by the mechanics-based risk of instability. Here, a quantitative measure of stability is used, first proposed in our work in [2]. In [10], it was demonstrated that instability manifests as an S-shaped curve relating the tube rotation at the base, α^{B}, to tube rotation at the tip, α^{T}. Observing Fig. 2, stable CTRs do not exhibit negative slopes on this S-curve [14]. This notion gives rise to the quantitative measure of stability, d_{sta}:

$$d_{\mathrm{sta}} = \frac{\pi}{2} - \operatorname{atan2}\left(1,\sigma^{q}\right) \tag{2}$$

where σ^{q} is the minimum inverse slope along the entire S-curve. The angle difference between the most critical slope

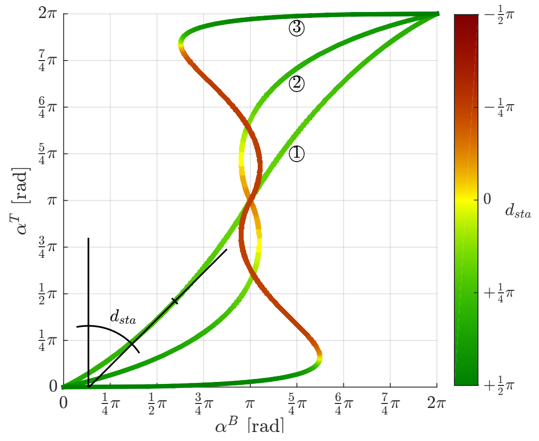

Fig. 2. S-curve: Relation of $\alpha^B \rightarrow \alpha^T$. The colour of the curve is a representation of d_{sta}, defined in (2). Curve ① is stable, curve ②, ③ are unstable.

of the S-curve [$\measuredangle = \operatorname{atan2}(1, \sigma^q)$] and the vertical axis ($\measuredangle = \frac{\pi}{2}$) represents the distance to an unstable configuration. The configuration is unstable *iff* $d_{\text{sta}} < 0$.

The advantage of quantitative measurements for the anatomical constraints (distance to anatomy) and stability (distance to unstable configuration) is the ease of generation of i. differentiable cost-functions for inclusion in optimisation-based inverse kinematics solvers, and ii. thresholds for accepting/rejecting robot configurations under examination.

III. Rapid Forward and Inverse Kinematics

This section introduces and describes our high-performance software architecture to iteratively solve the boundary value problem defining the kinematics, while respecting stability and anatomical constraints. The developed software is based on the C++14 standard. It achieves a compromise between the multitude of criteria requiring consideration in high-performance robotics applications, *e.g.*:

1) memory allocation;
2) memory alignment and random memory access;
3) cache misses;
4) cache coherency;
5) branching, conditionals;
6) dynamic dispatch (virtual functions);
7) vector instructions.

Criteria 1-4 relate to memory access bottlenecks. Dynamically allocated memory is reserved on the heap, which is time consuming and should be avoided. Furthermore, all memory-access timings depend on where the data is stored, *e.g.* main memory, processor cache. Accessing the cache is order of magnitudes faster. Therefore, avoiding cache misses and increasing cache coherency helps increase execution speed.

Criteria 5-7 relate to bottlenecks regarding instructions. Branching means that the sequence of processor instructions depends on specific conditions that usually need to be evaluated at run-time. Delays are caused if the processor mispredicts the branch of instructions it needs to flush and therefore reloads the instruction pipeline. Dynamic dispatch (*i.e.* virtual functions) means that the instructions for a specific function call are determined at run-time instead of compile-time, which introduces an overhead for each function call. Finally, vector-instructions operate on an array of data simultaneously [single instruction multiple data (SIMD)] rather than on a single datum. Using vector-instructions in conjunction with expensive operations, such as trigonometric operations, can substantially speed-up algorithm execution.

Memory usage in CTRs mainly depends on the number of sample points along the robot centre line as well as the robot architecture (*i.e.* type of tubes and number of tubes). Fixing the number of sample points allows the preallocation of the maximally needed memory, and thus no allocation is necessary during the forward kinematic calculations. Further, the memory is constrained by memory-alignment requirements to enable vector instructions (*e.g.* 64-byte for the Xeon Phi™architecture). Respecting these requirements allows memory allocation as a single chunk, which simplifies and expedites memory copying and network transmission of robot objects. Further, optimal memory allocation and alignment helps to fully utilize cache lines and hence optimises overall processor cache utilization, which reduces cache misses and increases its coherency, reducing potential bottlenecks caused by memory access times.

Branching caused by conditional expressions is, when possible, reduced by substituting loops and conditionals with equivalent constructs that are resolved at compile-time. The next section describes our implementation for compile-time unrolled `for`-loops and `if-else`-statements, based on `template` meta-programming.

The advantage of compile-time resolved loops and conditionals is not only the avoidance of branching, but also the further optimisation (*e.g.* auto vectorization) of code instructions by the compiler, since most instructions can be fully determined at compile-time.

Dynamic dispatch in C++ is the usage of virtual function calls. The overhead of these calls can accumulate significantly in particular if the function's complexity is minimal. We tackle this by avoiding some layers of polymorphism using variadic `templates` and precompiling many different concentric tube instances to avoid virtual function calls. Furthermore, vector instructions for costly functions (*e.g.*

```
#define CTR_ALIGNMENT   64
// Base class for the Concentric Tube Robot
template<class Real>
class alignas(CTR_ALIGNMENT) RobotBase
{
 public:
  typedef RobotBase<Real>  base_t;
  /* Append FC section and return the pointer to the new created robot */
  virtual base_t* appS(const FC<Real> & _sec) const noexcept = 0;
  /* Append VC section and return the pointer to the new created robot */
  virtual base_t* appS(const VC<Real> & _sec) const noexcept = 0;

  virtual Tr calcKinematic(const VecJ& _JVal,
                           const Real& _ArcLengthStep) noexcept = 0;
};
```

Code 1: Abstract `class` representing the CTR architecture.

```
 1 /* Helper struct to recursively append a section to the
 2    variadic template class (at compile-time) */
 3 #define CTR_MAXSECTION   8
 4 template<class Real, class ...Args> class Robot;
 5 namespace details {
 6  template<size_t secs_s>
 7  struct appH
 8  {
 9   typedef RobotBase<Real>  base_t;
10   template<class Real, class SecT, class ...Args>
11   static base_t* appS(const Robot<Real,Args...>& _rob_prev,
12                       const SecT& _sec) noexcept
13   { return new Robot<Real,Args...,SecT>(_rob_prev,_sec); }
14  };
15  template<> // Stop criteria for recursive appending
16  struct appH<CTR_MAXSECTION>
17  {
18   typedef RobotBase<Real>  base_t;
19   template<class Real, class SecT, class ...Args>
20   static base_t* appS(const Robot<Real,Args...>&,
21                       const SecT&) noexcept
22   { return nullptr; }
23  };
24 }
```

Code 2: Helper `struct` to append FC/VC section to the CTR.

trigonometric functions or square-root) can be used since the structural description of the robot is known.

A. *Robot architecture using `template classes`*

Fully defining a robot description at compile time reduces adaptability for the software user, and a compromise between compile- and run-time definitions is required.

Using `template classes` as described below provides a fully defined robot kinematic architecture at compile time. Although compilation might require some extra time, the gain of shifting run-time computations to compile-time is significant. Furthermore, code inlining and compiled code optimisation is improved, since the sequence of instructions is transparent at the time of compilation.

We create an abstract base `class` (see Code 1) from which a variadic `template class` derives (see Code 3). The abstract `class` has only a limited interface of elevator functions so that the computational overhead resulting from dynamic dispatch is marginal. The `Robot-class` encapsulates the generating components of a CTR, *i.e.* its sections and section types (FC, VC) in a tuple (Code 3, Line 29). Tuples provide an optimal memory footprint while allowing to store the sections without abstraction.

Many different `classes` derived from the templated `Robot class` will get compiled, covering several expected robot architectures[1]. For example if the maximum number of sections is set to 2, 6 different `Robot classes` will be automatically generated, corresponding to the 6 possible robot architectures maximally having 2 sections: {{FC},{VC},{FC,FC},{FC,VC},{VC,FC},{VC,VC}}. The number of generated `classes` is $n_{\texttt{class}} = 2^{(n_{\max,\text{sec}}+1)} - 2$. For $n_{\max,\text{sec}} = 8$ the compiler generates 510 `classes` in less than a minute.

[1]Compilation of multiple `Robot classes` is only necessary if the used robot architecture is not known at compile time. In case the software is tailored to a specific robot architecture abstraction or compilation of a set of differently `templated classes` is not necessary.

```
 1 template<class Real, class ...Args>
 2 class alignas(CTR_ALIGNMENT) Robot : public RobotBase<Real>
 3 {
 4  // Base class type
 5  typedef RobotBase<Real>      base_t;
 6  // Tuple type holding the different sections.
 7  typedef std::tuple<Args... > secs_t;
 8  // Tuple size, number of sections
 9  static constexpr size_t secs_s = std::tuple_size<secs_t>::value;
10  public:
11   /* Constructor to append a section.
12       I. arg: robot with one section less.
13      II. arg: section to append.  */
14   template<class R, class SecT>
15   explicit Robot(R&& _rob_prev, SecT&& _sec) noexcept
16           : m_Secs(std::tuple_cat(_rob_prev.m_Secs,
17                    std::forward_as_tuple(_sec))){}
18   virtual base_t* appS(const FC<Real> & _sec) const noexcept
19   { return details::appH<secs_s>::template
20     appS<Real,FC<Real>,Args... >(*this,_sec); }
21   virtual base_t* appS(const VC<Real> & _sec) const noexcept
22   { return details::appH<secs_s>::template
23     appS<Real,VC<Real>,Args... >(*this,_sec); }
24   virtual Tr calcKinematic(const VecJ& _JVal,
25                            const Real& _ArcLengthStep) noexcept
26   { /* Implementation of the forward kinematics */ }
27  protected:
28   // Different sections stored in tuple
29   alignas(CTR_ALIGNMENT) secs_t m_Secs;
30 };
```

Code 3: CTR variadic `template class`. The `template` arguments determine the section types, order and number.

The recursive generation of new `classes` stops when the upper limit of total section number is reached. Using partial-specialization for the `template` arguments (see Code 2, Line 16-24) the recursive instantiation process is halted. The number of sections and tubes for each generated `class` is known at compile time. Iterating over the tuple storing the section is also possible at compile time.

B. *Unrolling loops at compile time*

Iterating over tuples of sections requires the availability, at compile time, of the indices of the elements to be accessed, *i.e.* the index is a `template`-parameter. Therefore, a standard `for`-loop with dynamic loop count cannot be used; every

```
template <int Iter, int Last, int Inc = 1>
struct static_for
{
   template <typename Fn>
   inline void operator()(Fn&& fn) const noexcept
   {  // Call function
      fn( std::integral_constant<int,Iter>() );
      // Recursive call
      static_for<Iter+Inc,
                 // Last iteration if positive increments
                 (Inc>0)*(Iter+Inc*(1+((Last-Iter-1)/ Inc)))+
                 // Last iteration if negative increments
                 (Inc<0)*(Iter+Inc*(1+((Iter-Last-1)/-Inc))),
                 Inc>()(std::forward<Fn>(fn));
   }
};
template <int Last, int Inc>
struct static_for<Last,Last,Inc>
{  // Termination criteria
   template <typename Fn>
   inline void operator()(Fn&&) const noexcept{ }
};
```

Code 4: `for`-loop resolved at compile time, using `template`-ing and partial specialization.

```
1 auto t = std::make_tuple(unsigned(0.0),int(1.1),float(2.2),double(3.3));
2 static_for<0,std::tuple_size<decltype(t)>::value,1>()
3 ([&](auto ie)
4 {
5   static constexpr auto i =  decltype(ie)::value;
6   auto& ti = std::get<i>(t);
7   std::cout<<ti<<typeid(ti).name()<<", "<<std::flush;
8 });
```

```
Output: 0j,1i,2.2f,3.3d,
```

Code 5: Usage of `static_for`, using generic lambdas to provide the iteration index as `template`-argument.

loop needs to have iteration bounds known at compile time. A standard approach for this is to use recursive `template` calls, which act with each level of recursion on the next element in the tuple. This is addressed with a developed dedicated convenience `struct`: `static_for` (see Code 4). Passing a function-pointer to `static_for` results in a call to the pointed function with the current iteration index, and a recursive call to `static_for` with an incremented (`Inc` $>$ 0) or decremented (`Inc` $<$ 0) `Iter`-value. This approach unrolls the `for`-loop at compile-time, resulting in a speed-up of computations at runtime as the instructions can be vectorized. The lengthy calculations for the `Last-template` parameter are necessary to ensure that the stop criterion is met (*i.e.* if ($|$`Last-First`$| \bmod |$`Inc`$| \neq 0$)). To use the function parameter as a `template` parameter, we package the iteration index into an `integral_constant-class` (see Code 4, Line 7).

Using generic lambdas (see Code 5, Line 3) we can retrieve the loop index in the function definition, from the type [`decltype(ie)`] of the function parameter as `constexpr`, since we encoded it in the parameter type, instead of the parameter value. Code 5 shows the usage and the output using the `static_for` construct. The output, listed below the code segment, shows that each element type is different.

C. Resolving conditionals at compile-time

Runtime performance is commonly decreased by the abundance of machine-level jumps created by `if`/`else` statements. To improve performance by limiting the number of such jumps at compile time, an approach similar to the `static_for`-loop is pursued. When different code sequences need to be executed depending on the CTR section type, a dedicated construct, `static_if_else`, is used. Two function-pointers are passed to the `static_if_else struct` (Code 6). The first represents the `if`- instructions, while the second represents the `else`-instructions. Depending on the `template`-parameter of the `static_if_else`, the correct branch is already chosen at compile-time.

```
template <bool valid> struct static_if_else
{ // Call function: Call if-function
 template <typename FnIf,typename FnElse, typename... Args>
 auto operator()(FnIf&& fnif, FnElse&&, Args&&... args) const
  -> decltype(fnif(std::forward<Args>(args)...))
 { return fnif(std::forward<Args>(args)...); }
};
template <> struct static_if_else<false>
{ // Partial specialisation: Call else-function
 template <typename FnIf,typename FnElse, typename... Args>
 auto operator()(FnIf&&, FnElse&& fnelse, Args&&... args) const
  -> decltype(fnelse(std::forward<Args>(args)...))
 { return fnelse(std::forward<Args>(args)...); }
};
```

Code 6: `if`/`else` resolve at compile time depending on `template`-parameter, `FnIf` or `FnElse` is called.

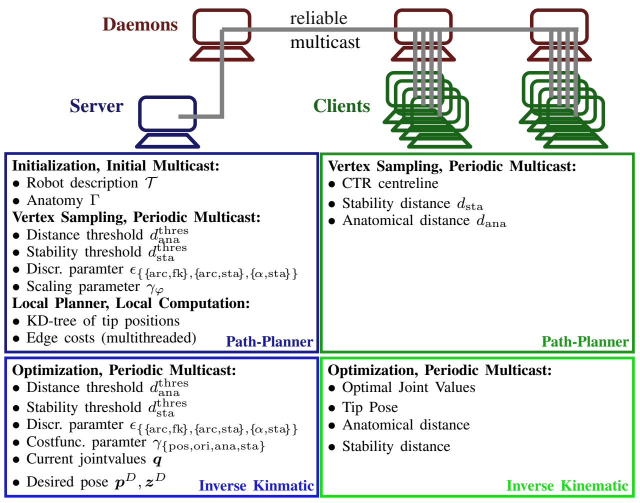
Daemons
reliable
multicast
Server
Clients
Initialization, Initial Multicast:
• Robot description $\mathcal{T}$
• Anatomy Γ
Vertex Sampling, Periodic Multicast:
• Distance threshold $d_{\text{ana}}^{\text{thres}}$
• Stability threshold $d_{\text{sta}}^{\text{thres}}$
• Discr. paramter $\epsilon_{\{\{\text{arc,fk}\},\{\text{arc,sta}\},\{\alpha,\text{sta}\}\}}$
• Scaling parameter γ_{φ}
Local Planner, Local Computation:
• KD-tree of tip positions
• Edge costs (multithreaded)
Path-Planner
Vertex Sampling, Periodic Multicast:
• CTR centreline
• Stability distance d_{sta}
• Anatomical distance d_{ana}
Path-Planner
Optimization, Periodic Multicast:
• Distance threshold $d_{\text{ana}}^{\text{thres}}$
• Stability threshold $d_{\text{sta}}^{\text{thres}}$
• Discr. paramter $\epsilon_{\{\{\text{arc,fk}\},\{\text{arc,sta}\},\{\alpha,\text{sta}\}\}}$
• Costfunc. paramter $\gamma_{\{\text{pos,ori,ana,sta}\}}$
• Current jointvalues $\boldsymbol{q}$
• Desired pose $\boldsymbol{p}^D, \boldsymbol{z}^D$
Inverse Kinmatic
Optimization, Periodic Multicast:
• Optimal Joint Values
• Tip Pose
• Anatomical distance
• Stability distance
Inverse Kinematic

Fig. 3. Blocks for calculating valid CTR configuration samples for the i. path-planning, ii. and for solving the inverse kinematic. Computational tasks performed by server: blue, by client: green. Server-client communication using reliable multicast via a daemon network: red.

Using `static_for` and `static_if_else` to compile CTR `classes` for all expected combinations of sections, every single `class` can be individually optimized by the compiler, even if from a developer's perspective only a single `class` needs to be written. The majority of machine-level code optimisations is achieved by the compiler, while the complexity of the implementation is hidden from the library user.

The next section leverages the speed-up achieved by the described implementation, detailed by benchmarks in Sec. VI-A, towards on-line path-planning and inverse kinematics for assisted telemanipulation.

IV. Guidance via Implicit Active Constraints

Based on our high-performance software architecture, we generate IACs [20]. In our work, guiding paths (*i.e.* path plans) are generated based on pre-computed road-maps and navigation goal position as defined by the operating clinician. Due to the computational efficiency of the software framework pre-computation occurs in matter of minutes, rather than hours. In the pre-computation step, random configurations of the robot are generated using a parallel computing approach. A schematic representation of the framework is depicted in Fig. 3. Our path-planner's efficiency heavily relies on the developed multi-node framework. A central computer (server) controls computing clients that generate random robot-configuration samples and solve the inverse kinematics using different optimization techniques (see Sec V).

Please refer to [2] for a detailed description of our path-planning framework.

A. Probabilistic Roadmap

The constraint-generation framework is based on an undirected graph G, with vertices $v_i \in G$. The vertices of

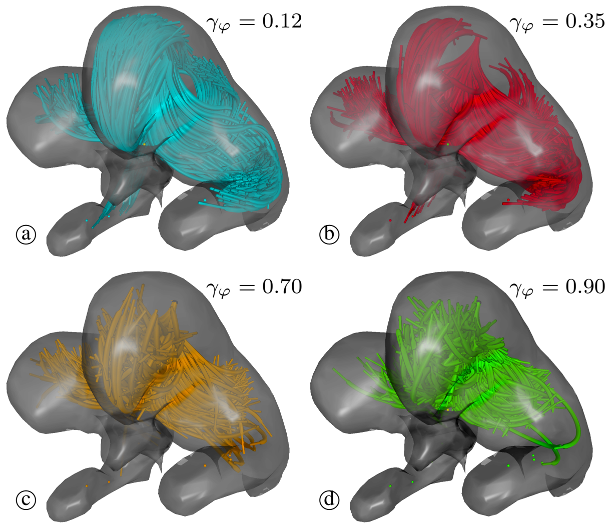

Fig. 4. Random concentric tube configurations with different scaling of the translational component, 1000 configurations each. ⓐ: $\gamma_\varphi = 0.12$, ⓑ: $\gamma_\varphi = 0.35$. ⓒ: $\gamma_\varphi = 0.70$, ⓓ: $\gamma_\varphi = 0.90$.

the graph represent random, stable, and collision-free CTR configurations. Edges $e_{i,j}$ between vertices v_i, v_j represent possible transitions among the configurations. The graph is queried using the $A^\star$ graph-search algorithm to efficiently extract the shortest path between the current robot configuration and the configuration corresponding to a desired tip pose. The Euclidean norm between two vertex positions is the admissible $A^\star$ heuristic.

The extracted series of robot configurations generates a guidance path along which the operator is guided.

B. Precomputation of Graph - Generation of Roadmap

To obtain the vertices of the graph safe robot configurations have to be found. During that process the server controls parameters defining configuration sampling (density) (γ_φ, $\boldsymbol{q}$) and configuration acceptance ($d_{\text{ana}}^{\text{thres}}$, $d_{\text{sta}}^{\text{thres}}$). Since the goal is to obtain uniformly distributed configurations in task-space, the joint-space sampling has to be non-linear. Extended robot-configurations in most scenarios will be rejected as they will collide with the anatomy. This leads to a bias towards shorter robots, which needs to be accounted for. Therefore, using a random uniformly distributed number $q_{\text{r,u}} \in [0,1]$ for a tube with a maximum extension of ${}_i\varphi^{\max}$ we scale the translation/extension joint values with:

$$q_{\text{r}} = {}_i\varphi^{\max} \, {q_{\text{r,u}}}^{\gamma_\varphi}, \; \gamma_\varphi \in [0,1] \tag{3}$$

The extent of sampling elongated versus retracted robots is governed by γ_φ, as depicted in Fig. 4.

Each client calculates the forward kinematics for the given joint value, determines d_{sta} (2) and d_{ana} (1), and sends the configuration to the server *iff*

$$d_{\text{col}}\left(\boldsymbol{q}_r\right) \leq d_{\text{col}}^{\text{thres}} \;\wedge\; d_{\text{sta}}\left(\boldsymbol{q}_r\right) \leq d_{\text{sta}}^{\text{thres}}. \tag{4}$$

Results on the time required to compute the road-maps and the effect of the scaling factor are reported in Sec. VI-B.

When the server has received a minimum number of configurations, a local planner calculates edges $e_{i,j}$ and the corresponding cost for transitioning from one robot configuration represented by the vertex v_i to another vertex v_j. Edge generation is performed in two stages: first the cost for a potential edge $e_{i,j}$ has to be below a certain threshold, and second, only the edges with the $N_s^{\max}$ smallest costs are introduced in the graph. The edge $\omega(v_j, v_k)$ cost depends on

$$f_{\text{ee}}(v_j, v_k) = \|\boldsymbol{x}_{ee}(v_j) - \boldsymbol{x}_{ee}(v_k)\| \tag{5.1}$$

$$\boldsymbol{g}_{\text{jv}}(v_j, v_k) = \text{abs}(\boldsymbol{q}\{v_j\} - \boldsymbol{q}\{v_k\}) \tag{5.2}$$

$$h_{\text{cl}}\left(v_j, v_k\right) = \sqrt{\frac{\sum_{i=1}^{N_{cl}} \|\boldsymbol{x}_{cl}\{i, v_j\} - \boldsymbol{x}_{cl}\{i, v_k\}\|^2}{N_{cl}}} \tag{5.3}$$

where $\boldsymbol{x}_{ee}$ provides the end-effector position of a vertex, $\boldsymbol{q}$ returns the joint-value vector of vertex, abs provides the element-wise absolute values of a vector, N_{cl} is the larger of the centre line point counts of v_j, v_k, and $\boldsymbol{x}_{cl}\{i, v_j\}$ provides the position of i-th centre line point of vertex v_j if i is smaller than the number of centre line points, and otherwise provides the position of the final centre line point.

The edge cost is calculated as:

$$\begin{aligned} &\omega(v_j, v_k) = \omega(e_{j,k}) \\ &= f_{ee}(v_j, v_k) + \boldsymbol{\omega}_{jv}^T \, \boldsymbol{g}_{\text{jv}}(v_j, v_k) + \omega_{cl} h_{\text{cl}}\left(v_j, v_k\right) \end{aligned} \tag{6}$$

where $\boldsymbol{\omega}_{jv}^T$ is the vector of weights for the joint value differences, and ω_{cl} is the weight for the root-mean-square error of the centre-line differences. The graph generation process is detailed in [2].

C. Guidance Generation - Query of Roadmap

The guidance constraint is calculated from the implicit constraint defined by the desired user end-effector position and the roadmaps, utilizing a shortest-path search. The source vertex is the vertex of graph G that is closest to the current configuration based on the cost-function described in (5.1)-(5.3). The goal vertex is determined by the user-defined target position. It is selected from a set of vertices that are ranked based on the distance between graph G vertices and the target position. Since the computational cost of finding the shortest

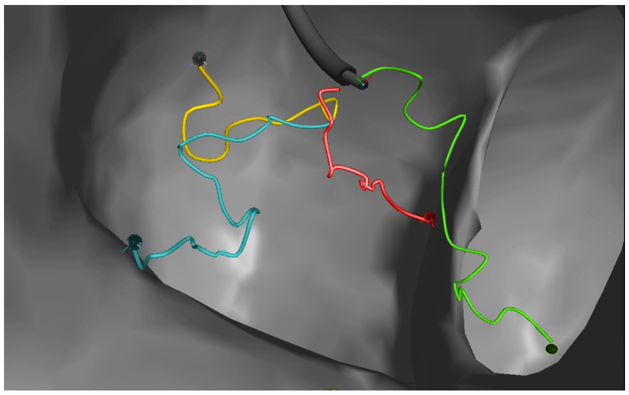

Fig. 5. Implicit Active Constraints: Guidance paths to navigation goals.

path from the source vertex to all vertices in the goal set can be very high, we chose a subset of the potential vertices. During parallel computation, calculating paths to very similar goal configurations is avoided by maximising the cost, *i.e.* the robot configuration discrepancy, between the potential goal vertices. This increases the chances of finding a low-cost path from the source vertex. In a last step, multiple shortest path searches to the set of goal vertices are performed using the $A^\star$ algorithm and the resulting lowest-cost path ($\mathcal{P}_s$) is chosen for user guidance. An example of shortest paths to various targets is depicted in Fig. 5.

V. Inverse Kinematics

The shortest-path search between the current robot configuration and the desired navigation target provides visual and haptic guidance to the user. Nevertheless, we ensure that the operator is in full control of the robot. Towards this end, we use the on-line inverse kinematics algorithm presented in [1] to allow user-intended deviation from the given paths. Multiple optimisation algorithms, namely:

- Controlled Random Search with Local Mutation;
- Multi-Level Single-Linkage with Low-Discrepancy;
- Bound Optimisation by Quadratic Approximation;
- Principal Axis;
- Software for unconstrained minimisation w/o derivative;
- Nelder-Mead simplex algorithm;
- Damped-least squares Jacobian;

run in parallel on disparate computing devices to minimise the cost function:

$$\begin{aligned} \mathcal{C}\left(\boldsymbol{q},\mathcal{T},\Gamma,\boldsymbol{p}^D,\boldsymbol{z}^D\right) &= \gamma_{\text{pos}}\ \left|\boldsymbol{p}^D-\boldsymbol{p}^O\right|_2 \\ &+ \gamma_{\text{ori}}\ \operatorname{asin}\left(\left|\boldsymbol{z}^D-\boldsymbol{z}^O\right|_2\right) \\ &+ \gamma_{\text{ana}}\ \mathcal{C}_{\text{ana}}\left(\boldsymbol{q},\mathcal{T},\Gamma\right) \\ &+ \gamma_{\text{sta}}\ \mathcal{C}_{\text{sta}}\left(\boldsymbol{q},\mathcal{T}\right) \end{aligned} \tag{7}$$

This cost function depends on the current robot end-effector pose ($\boldsymbol{p}^O,\boldsymbol{z}^O$), anatomy ($\Gamma$), robot kinematics ($\mathcal{T}$), and desired end-effector pose ($\boldsymbol{p}^D,\boldsymbol{z}^D$). Further, there is dependence on cost functions that encode risk of collision with the anatomy ($\mathcal{C}_{\text{ana}}$) and configuration instability ($\mathcal{C}_{\text{sta}}$).

The inverse kinematics solver is initialized based on (5.1)-(5.3) with the joint values of the vertex $v_i \in \mathcal{P}_s$ closest to the current robot configuration. Therefore, finding a locally best configuration for the current desired position (telemanipulation position) is based on the globally optimized configuration to reach the goal robot configuration.

Haptic guidance is based on an elasto-plastic friction model, which redirects kinetic energy from user input towards the guidance path [16]. The parameters governing the extent and feel of haptic guidance can be found in Table I and follow the formulas introduced in [1].

TABLE I
Friction Constraint Parameters

Parameter (GVF)		Value	Parameter (GVF)		Value
f_C	[N]	1.75	σ_0	[N/mm]	0.5
Θ_{appex}	[°]	15	σ_1	[N.s/mm]	0.0003
z_{css}	[mm]	3.5	σ_2	[N.s/mm]	0.0012

TABLE II
Profiler results for 1000 random configurations

Profiler metric	variadic template	simple template	completely dynamic
Instruction refs.	1124E5	1303E5	19 296E5
Data refs.	526E5	674E5	7824E5
Last-Level cache	0.31E5	0.55E5	2.75E5
Branches	28E5	40E5	2949E5
Misprediction	0.21E5	0.25E5	5.16E5
Misprediction rate	0.73%	0.63%	1.75%

VI. Experimental Results

A. Computational Benchmark

This section compares three kinematic implementations written in C++ to demonstrate the computational efficiency gained by the novelties of the work described herein.

The first implementation is «completely dynamic» and was used in [14]. Each robot consists of a set of sections, and each section consists of a number of tubes. Limited compile-time optimisation is possible with this approach, and, hence, the computational efficiency is low.

The second implementation, «simple template», encodes the robot as a `template class` with two parameters: i. number of sections, and ii. number of tubes. Although, this approach allows to fully unroll loops and to resolve conditional expressions at compile-time, the `section-class` had to be implemented using `virtual`-functions, which resulted in important run-time overheads.

The final approach, «variadic template», is the contribution of this paper and follows the new design patterns.

All benchmarks were performed on an Intel® Core™ i7-3770 CPU. Table II show profiler results from the Cachegrind-software package, when compiled with the Intel® C++ Compiler (ICC) 2016.3. It shows improvements of more than an order of magnitude between the `template`-code and the «completely dynamic» code. The improvements between the «variadic template» code and the «simple template» code are also significant, ranging from 13.7% (instruction references) to 42.4% (last-level cache references). Therefore, it is expected that computation time will significantly improve for the «variadic template» implementation.

The computation times for the three implementations, listed in Table III, validate this expectation. The computational-time reduction between the «completely dynamic» implementation and the `template`-code is more than an order of magnitude. Depending on the compiler choice, «variadic template» code ran up to 15.8 times faster. In comparison with the «simple template» code, an improvement of

TABLE III
Computational time for 100000 random configurations

Implementation	Time	Ratio*	Time	Ratio*	Time	Ratio*
	ICC 2016.3		GCC 6.1		Clang 3.8	
variadic template	1.75 s	100 %	2.94 s	100 %	2.80 s	100 %
simple template	1.88 s	107 %	3.25 s	111 %	2.93 s	105 %
completely dynamic	27.68 s	1580 %	32.28 s	1100 %	31.58 s	1129 %

*Ratio of the time compared to the respective fastest algorithm.

TABLE IV
DURATION TO SAMPLE WORKSPACE
TIME TO COMPLETE SAMPLING [S] VS. ACCEPTANCE RATE [%]

Scaling (γ_φ)	**configurations 250,000**	**configurations 1,000,000**	**Parametri-zation:**
0.1	107.9 s	550.8 s	$\epsilon_{\text{arc,fk}}$
	0.044 %	0.046 %	1 mm
0.4	21.5 s	81.1 s	$\epsilon_{\text{arc,sta}}$
	2.38 %	2.55 %	2 mm
0.7	16.9 s	70.4 s	$\epsilon_{\alpha,sta}$
	10.6 %	10.7 %	$1\,^\circ$
1.0	17.4 s	60.7 s	$d_{\text{col}}^{\text{thres}}$ / $d_{\text{sta}}^{\text{thres}}$
(uniform)	20.9 %	20.4 %	1 mm / $5\,^\circ$

7% is measured using the ICC compiler. This evolutionary improvement justifies an increased initial implementation complexity because usability, and computational precision are identical and the source-code maintenance became simpler using the «variadic template». The compiler flags were tuned individually for all examined compilers to guarantee that each one performs at its best. The binaries produced by GCC 6.1 and Clang 3.8 had a similar runtime, whereas the ICC 2016.3 compiler produced a significantly faster code[2].

The relative improvements from ICC to GCC or Clang were less for the «completely dynamic» implementation.

B. Creation of Road-map

Durations for computing safe configuration samples for the road maps are listed in Table IV. The results show timings and acceptance rates for the respective translational scaling factor (γ_φ). Obtaining a graph with 1,048,576 vertices, and 209,715 vertices per $\gamma_\varphi \in [0.12, 0.25, 0.35, 0.70, 0.90]$ took 186.8 s with 149.4 s for the sampling and 37.4 s for defining the edges of the graph in the local planning step.

C. User Experiment

This section describes the benefits of the developed framework from the operator's perspective, evaluating the value of interactive-rate inverse kinematics and implicit active constraints. The simulated clinically scenario is based on an intervention that involves cauterisation of the choroid plexus in hydrocephalic ventricles. In this procedure, an elongated CTR needs to access the base of the ventricles and cauterise them to limit the production of cerebrospinal fluid as a means to indirectly reduce ventricle pressure. The procedure is envisioned as an alternative to endoscopic third ventriculostomy [7], and requires highly curved CTR that are prone both to instabilities and to collisions with the anatomy (see robot parameters in Table V).

In the experiment 4 novice female and 10 novice male participants, age 25-35, had to manoeuvre the CTR towards 13 cauterization points using a haptic device (Geomagic Touch). The task was performed in two different modes: «guidance», «free». The order of the modes was randomised for every participant. The experimental protocol recorded metrics on a seven-level Likert scale, such as execution time, frustration, ease of telemanipulation etc. Each participant

[2]Linking the GCC and Clang executable to the Intel® Math Kernel Library (MKL) increases performance such that the advantage of ICC reduces to $\approx 10\%$.

TABLE V
ROBOT PARAMETERS (3 SECTIONS, 4 TUBES)

Robot Section Type	**Curvature** [mm^{-1}]	**Straight/Curved Length [mm]**	**Stiffness Ratio**	**Tube Index**
variable curvature	0.0522	0.00/66.47	5:1	1, 2
fixed curvature	0.05263	66.47/66.70	1:1	3
fixed curvature	0.0	113.17/12.65	0.05:1	4

could activate the IAC on-demand in the «guidance» mode to evaluate the effectiveness of the proposed cooperative guidance framework which provides: visual guidance, haptic guidance, and path-planning aided inverse kinematics, compared to the «free» mode, where no path guidance was provided but only collision avoidance guidance and the inverse kinematics was solely based on on-line optimization. Users could choose to follow closely or approximatively the guidance path to command the robot, allowing easy manipulation even when the shape of the path appeared complex.

The list of quantitative performance metrics used includes:

- Duration [min]: Duration of task completion;
- P_{ee} TD [m]: Travel distance of CTR tip;
- P_{sp} TD [m]: Travel distance of CTR desired position;
- $\Delta P_{\text{ee}|\text{sp}}$ o.T. [m min]: Integral of $\|P_{\text{ee}} - P_{\text{sp}}\|_2$;
- ΔP_{IK} [mm]: Accuracy of inverse kinematics;
- ΔT_{opt} [ms]: Mean time of IK updates.

The results of the experiment are listed in Table VI. They show statistical significance based on the Wilcoxon signed-rank test in metrics 1-4. This indicates an overall better task performance when using the proposed path-planning and guidance framework. Fig. 6 shows two plots of the distance between the operator's desired position and the CTR's tip position indicating that the path-planning improves the convergence of the inverse-kinematics; the reason for this is that without path-planning the local inverse kinematics solver cannot escape a local minimum, which originates from the stability constraints.

In a qualitative assessment, the participants were asked to compare the modes, on a scale from -3 to $+3$ (-3: negative evaluation, $+3$: positive evaluation). The results are listed in Table VII. The majority of the users stated to prefer the «guidance» mode over the «free» mode. The users felt more in-control with guidance enabled, giving them a safer feeling. Unsurprisingly, the haptic feedback arising from the attractive and repulsive elasto-plastic friction model was evaluated as useful by the users.

TABLE VI
QUANTITATIVE RESULTS FROM USER EXPERIMENT.

Metric	**Free Mode**†	**Guidance Mode**†	**p-value**
Duration [min]	12.9 [10.5,15.9]	8.6 [6.0,10.5]	0.9%
P_{ee} TD [m]	2.0 [1.8, 3.5]	1.1 [1.0, 1.3]	1.6%
P_{sp} TD [m]	3.2 [2.7, 4.7]	1.4 [0.9, 1.6]	0.0%
$\Delta P_{\text{ee}|\text{sp}}$ o.T. [m min]	7.4 [2.9, 8.7]	0.6 [0.5, 1.4]	1.6%
ΔP_{IK} [mm]	0.7 [0.4, 1.7]	0.7 [0.2, 0.8]	11.3%
ΔT_{opt} [ms]	2.0 [2.0, 2.0]	2.0 [2.0, 2.0]	20.6%

†Value represents the median, in parenthesis 25th and 75th percentile.

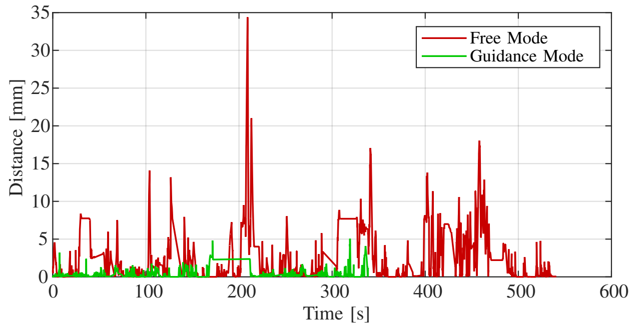

Fig. 6. Distance between robot end-effector P_{ee} and user set-point P_{sp}.

VII. Discussion and Conclusions

The paper presents a novel software design and computing architecture to rapidly calculate dense path-plans to safely guide the operator during telemanipulation of a CTR. The emphasis of this paper is on the high-level code optimisation using a `variadic template` software-approach and C++14 features. The paper makes the case for the requirement of tailored and optimised software libraries as a solution to computationally challenging problems in robotics. Existing approaches in the literature highlight the inverse kinematics of CTR as a highly resource-intensive problem, which our software architecture makes tractable and solvable in real-time. The performance and effectiveness of the presented approach is demonstrated by a variety of metrics established in software engineering, as well as via a multi-user experiment that focused on human factors and useability.

Our approach is still limited by the fact that a reduced-time pre-computation is required, and by the time required to calculate the path plan intraoperatively. Even though this amounts only to 3-5 min, and 1-10 s, respectively, we aim to reduce it further by investigating approaches that warp the workspace based on real-time collision avoidance to enable dynamic active constraints, providing operator guidance in cases where the anatomy undergoes motion and manipulation. Without frameworks such as the proposed one, advances in complex-robot telemanipulation will lag.

The proposed approach of completely `templated classes` can also be used for other variations of continuum robots where the kinematics variables and robot shape are coupled, leading to computationally intensive kinematics. Kinematic chains of standard serial-link robots could also be an area of application, with each tuple element representing a different type of joint (*e.g.* rotational, prismatic, universal), albeit with limited benefit due to their significantly simpler kinematics.

TABLE VII
Qualitative Average Results of User Experiment [−3,+3]

Metric	Free Mode	Guidance Mode
Performance efficiency	-1.54	1.54
Performance safety	-1.54	2.15
Frustration	-1.46	1.23
Overall performance	-0.69	1.62
Guidance Forces	1.85	

VIII. Acknowledgement

The authors gratefully acknowledge Prof. Pierre Dupont from Harvard Medical School for providing the anatomy used in this paper.

References

[1] K. Leibrandt, C. Bergeles, and G.-Z. Yang, "On-line collision-free inverse kinematics with frictional active constraints for effective control of unstable concentric tube robots," *IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, pp. 3797–3804, 2015.

[2] K. Leibrandt, C. Bergeles, and G.-Z. Yang, "Implicit active constraints for safe and effective guidance of unstable concentric tube robots," *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pp. 1157–1163, 2016.

[3] J. Burgner-Kahrs, D. C. Rucker, and H. Choset, "Continuum robots for medical applications: a survey," *IEEE Trans. Robotics*, vol. 31, no. 6, pp. 1261–1280, 2015.

[4] R. J. Webster and B. A. Jones, "Design and kinematic modeling of constant curvature continuum robots: a review," *Int. J. Robotics Research*, vol. 29, no. 13, pp. 1661–1683, 2010.

[5] J. Burgner *et al.*, "A telerobotic system for transnasal surgery," *IEEE/ASME Trans. Mechatronics*, vol. 19, no. 3, pp. 996–1006, 2014.

[6] R. J. Hendrick, C. R. Mitchell, S. D. Herrell, and R. J. Webster, "Hand-held transendoscopic robotic manipulators: A transurethral laser prostate surgery case study," *Int. J. Robotics Research*, vol. 34, no. 13, pp. 1559–1572, 2015.

[7] C. Bergeles *et al.*, "Concentric tube robot design and optimization based on task and anatomical constraints," *IEEE Trans. Robotics*, vol. 31, no. 1, pp. 67–84, 2015.

[8] L. G. Torres, R. J. Webster, and R. Alterovitz, "Task-oriented design of concentric tube robots using mechanics-based models," *IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, pp. 4449–4455, 2012.

[9] J. Burgner, H. B. Gilbert, and R. J. Webster, "On the computational design of concentric tube robots: Incorporating volume-based objectives," *IEEE Int. Conf. on Robotics and Automation*, pp. 1193–1198, 2013.

[10] P. E. Dupont, J. Lock, B. Itkowitz, and E. Butler, "Design and control of concentric tube robots," *IEEE Trans. Robotics*, vol. 26, no. 2, pp. 209–225, 2010.

[11] H. B. Gilbert, R. J. Hendrick, and R. J. Webster, "Elastic stability of concentric tube robots: a stability measure and design test," *IEEE Trans. Robotics*, vol. 32, no. 1, pp. 20–35, 2016.

[12] R. Xu, S. F. Atashzar, and R. V. Patel, "Kinematic instability in concentric-tube robots: modeling and analysis," *IEEE RAS/EMBS Int. Conf. Biomedical Robotics and Biomechatronics*, pp. 163–168, 2014.

[13] J. Ha, F. C. Park, and P. E. Dupont, "Elastic stability of concentric tube robots subject to external loads," *IEEE Trans. Biomedical Engineering*, vol. 63, no. 6, pp. 1116–1128, 2016.

[14] C. Bergeles and P. E. Dupont, "Planning stable paths for concentric tube robots," *IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, pp. 3077–3082, 2013.

[15] R. J. Hendrick, H. B. Gilbert, and R. J. Webster, "Designing snap-free concentric tube robots: A local bifurcation approach," *IEEE Int. Conf. Robotics and Automation*, pp. 2256–2263, 2015.

[16] S. Bowyer and F. Rodriguez y Baena, "Dynamic frictional constraints in translation and rotation," *IEEE Int. Conf. Robotics and Automation*, pp. 2685–2692, 2014.

[17] D. C. Rucker and R. J. Webster, "Computing jacobians and compliance matrices for externally loaded continuum robots," *IEEE Int. Conf. Robotics and Automation*, pp. 945–950, 2011.

[18] L. G. Torres *et al.*, "A motion planning approach to automatic obstacle avoidance during concentric tube robot teleoperation," *IEEE Int. Conf. Robotics and Automation*, pp. 2361–2367, 2015.

[19] A. Kuntz *et al.*, "Motion planning for a three-stage multilumen transoral lung access system," *IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, pp. 3255–3261, 2015.

[20] K. Leibrandt, H. Marcus, K.-W. Kwok, and G.-Z. Yang, "Implicit active constraints for a compliant surgical manipulator," *IEEE Int. Conf. Robotics and Automation*, pp. 276–283, 2014.