Generalized Geometric Constraint Solving and Mechanism Output Parameterization for Computational Design

William C. Rodgers, III; Dr. Joran Booth

Abstract—**Computational design tools have traditionally existed to improve the efficiency of mid- to late-stage phases in the mechanical design process, such as dimensional drawing generation, preparation for manufacturing, and assembly bill-of-material finalization. This CAD (Computer-Aided Design) software has become irreplaceable in the engineering and creative industries for the development of technical documentation, and has even expanded into numerical analysis for optimization of designed structures, but is nonetheless absent from early concept generation. With developments in machine learning, the field of generative design has begun to expand into this gap, providing designers with enhanced tools for application in the earlier stages of mechanism creation. This project seeks to explore and contribute to these new advancements, most notably through the development of a generic three-dimensional geometric constraint solver and the tools for classification of mechanism outputs and motion.**

I. Introduction

"GENERATIVE" or "computational design" are terms used to describe the process through which a computer is made to iterate over and optimize some aspects of an engineering design. The most widespread example is static analysis of structures. Many major engineering and architecture CAD packages now include workflows for the automated optimization of structural member mass given a set of user-provided static loads and manufacturing constraints. [1]–[3] The use of these tools is often referred to as 'optioneering,' because they still rely on user selection of a final design based on experience and intuition. [4] That is, the tools require some seed, provided by the designer, from which to generate potential optimized outputs.

The expansion of generative design into the creation of dynamic systems, such as mechanisms, is ongoing. Foundational conceptual works are largely based on abstraction and representation of mechanism functions and features. The techniques involved are widely varied, but earlier works tend to focus on graph-based representations of kinematic or structural 'building blocks.' [6], [7] Equivalent representations using adjacency matrices are also common in literature and lend themselves well to discretized collections of input-output types. [8], [9] The use of these abstractions for design is often referred to as 'type synthesis' or 'type selection,' and was typically limited to simplified, abstract development of a kinematic chain. Notably, these works are typically sufficiently generic to be applicable for both planar and three-dimensional mechanisms, but are too generic for specific input of desired, dimensioned output characteristics. The major relevant takeaways for this project are that mechanism function and mechanism structure are best abstracted and modularized separately, and that graph theory, and the equivalent linear algebra computations, is applicable to both. [6], [10]

Fig. 1. An example of a topologically optimized mounting bracket. [5]

More modern developments have been in the fields of graphics, robotics, and creative mechanisms, such as toys or animatronics. These works typically pull from literature on type synthesis in their use of a discretized solution-space – that is, a collection kinematic building blocks – but often emphasize user interface over provably complete abstractions. [11]–[13] This research also suffers from limitations in that specified motions are generally planar, or the choice of available actuators is made intentionally limited. The work by Coros et al. is perhaps most comprehensive in its applicability and usability, given that the user provides only a desired motion path and receives a complete, parameterized mechanism, but is nonetheless limited to sliding and four-bar linkages with single-input, two-dimensional gear-trains and cyclic outputs. [13] The paper also does not consider velocity or timing in mechanism outputs, under the assumption that, for artistic purposes, non-circular gears would suffice for control. The goal of this project is thus to attempt to expand upon those aspects that limited the scope of this work while maintaining its versatility.

II. Geometric Constraint Solver

A. Background

The bulk of the work for the project was in the development of a geometric constraint solver (GCS) suitable for more generic, three-dimensional applications than those possible with that proposed by Coros et al. [13] This task is non-trivial. As alluded to in the type synthesis literature, because a GCS is effectively a structural decomposition, it is well-suited to graph-based formulations. This is supported by literature on the subject. Joan-Arinyo describes three methods of constraint solving: (1) graph-based, (2) logic-based, and (3) algebra-based, wherein the corresponding system is solved either symbolically or numerically. [14] In practice, in three dimensions, the first and third are often combined. Peng et al. describe one formulation of a combined graph-numerical approach to constraint solving. [15]

However, as evidenced by Coros et al. and Durand, numerical techniques alone are sufficient, while, in some cases, graph-based representations may fail to entirely decompose a constraint proposition geometrically and would thus occasionally rely on some form of simultaneous solver. [13], [14], [16] Thus, despite longer asymptotic running times and typical dependence on starting conditions, the developed GCS relies on numerical approximation. However, with the understanding that this is non-optimal, the system is designed such that mechanism representation is decoupled from solution methodology, ensuring later developments can easily include graph-based techniques.

TABLE I
Standard Constraints

Constraint	Input	Evaluation Function
Location	position vectors, p_1, p_2	$\lvert p_1 - p_2 \rvert^2$
Orientation	orientation quaternions, o_1, o_2	$4 \arccos^2 \left(\Re \left(o_1 o_2^{-1} \right) \right)$
Axis-Aligned	axis vectors, a_1, a_2	$\arccos^2 \left(\frac{a_1^T a_2}{\lvert a_1 \rvert \lvert a_2 \rvert} \right)$

B. Implementation

The overall role of the GCS is to group members and constraints into mechanisms, then based on member parameters, evaluate the constraints as a cost function. The smaller the magnitude of the sum of the evaluated constraint functions, the closer the mechanism is to a solved state.

1) Mechanism and Member Structure: As suggested by Gui and Mäntylä, the motion and structure of the mechanism are decoupled in the GCS. [10] This is achieved by abstracting the mechanism into a collection of members. Each member maintains position and orientation separate from its shape. The shape is itself an arbitrary collection of points describing the member's structure. More formally, a given mechanism, $\boldsymbol{A}$, is described as:

$$\begin{gathered} \boldsymbol{A} = \{M, C\} \\ M = \{m_1, m_2, ...\},\ C = \{c_1, c_2, ...\} \\ m_i = \{p_m, o_m, S\} \\ S = \{p_1, p_2, ...\} \\ p_m, p_i \in \mathbb{R}^3, o_m \in \mathbb{R}^4 \end{gathered}$$

where C is the set of constraints in $\boldsymbol{A}$. Note that $o_m \in \mathbb{R}^4$ because it is stored in quaternion representation. This ensures the solver will avoid gimbal lock. [15] Shape points are stored in member space - that is, relative to the members position and orientation. This allows for the same shape data to be used for multiple members, decreasing memory requirements for large, symmetric mechanisms.

2) Constraint Structure and Numerical Solver: The set of possible constraints is designed explicitly for extensibility. The current implementation defines a collection of 'standard constraints,' wherein each can be either 'fixed' or 'relative.' Inputs and the associated evaluation functions are shown for each constraint in Table I. Fixed constraints define geometric relations between a single member and world space, while relative constraints define relations between two members. For fixed constraints, only the first input is transformed from member space

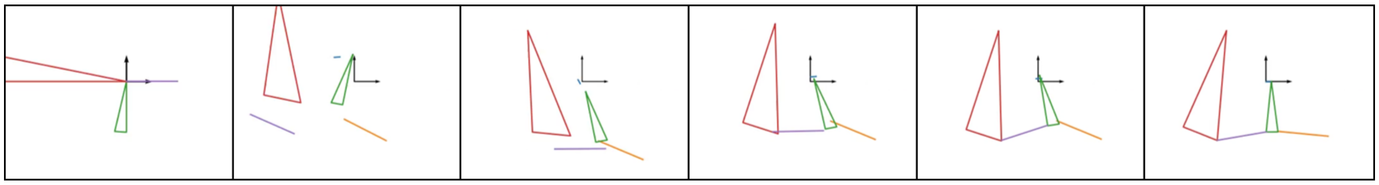

Fig. 2. From left to right, the starting state, then SLSQP iterations 5, 10, 15, 30, and 43 (final) for a planar six-bar mechanism.

to world space - by rotation and translation for positions and rotation only for axes and orientations. For relative orientations, both inputs are transformed prior to comparison.

If a constraint is not 'standard,' it is either a group constraint - a collection of other constraints which evaluates as the sum of its sub-constraints - or a custom constraint, defined by the user as an arbitrary function that takes a set of members' positions and orientations and evaluates to an associated cost.

The currently implemented Solver takes the Mechanism state - that is, its members and constraints - and performs Sequential Least Squares Programming (SLSQP) optimization on the constraint cost function by altering member locations and orientations. At each iteration, the Jacobian of the cost function is approximated with a two-point finite difference. An example convergence for a planar six-bar mechanism is shown in Fig. 2. This numerical approximation is implemented using the open-source `scipy` scientific computing library. As previously noted, the mechanism data- and functional-structures are intentionally abstracted from the nature of the solver to facilitate future development of more advanced solving techniques.

Solved mechanism states are also cached for each time step, such that, should the user wish to walk through the steps, they need not be recalculated.

3) Mechanism Inputs and Outputs: Mechanism inputs are represented as interpolated constraints. That is, a given input holds a constraint type, a time range, and a set of parameters - vectors or quaternions - through which to interpolate. At solve-time, the input constructs the constraint by interpolating the given parameters and constructing the corresponding constraint based on the current time step and its associated time range. If the current time step is outside the input's time range, it creates no constraint. For vector parameters, the interpolation is linear. For quaternion parameters, the interpolation is spherical linear.

Mechanism outputs are represented as a set of tracked points relative to associated members. At each solved time step, the mechanism transforms these tracked points from member space to world space and stores their positions with the current time. Once the mechanism motion has completed solving, the user can request the path which the point took throughout the motion.

Upon request, the velocities at each time step are approximated. If the initial and final points are within some small threshold t, the mechanism is assumed to by cyclic. For cyclic mechanisms, the velocities are calculated as the derivative of the parabolic interpolation through each point and its two neighboring points:

$$\begin{aligned} v(t_i) \approx & \frac{t_i - t_{i+1}}{(t_{i-1} - t_i)(t_{i-1} - t_{i+1})} p_{i-1} \\ & + \frac{(t_i - t_{i+1})(t_i - t_{i-1})}{(t_i - t_{i+1})(t_i - t_{i-1})} p_i \\ & + \frac{t_i - t_{i-1}}{(t_{i+1} - t_{i-1})(t_{i+1} - t_i)} p_{i+1} \end{aligned}$$

For non-cyclic mechanisms, the first and final velocities are approximated using single-point forward and backward differences, respectively, while all others use parabolic interpolation.

Note that, because solved mechanism states are cached, the addition of a tracked point to the mechanism provides immediate curve output if the motion has already been calculated. Because track points are arbitrary in their position relative to a member, this permits the user to gain significant detail regarding mechanism characteristics from only a single set of solution iterations.

Fig. 3 shows the solution for a simple planar crank-rocker mechanism. The blue link is an input which rotates 360° throughout the motion. Three points are tracked on the green coupler, and the corresponding output curves are shown with their velocity fields as they are progressively calculated. Fig. 4 shows the solution for a similar mechanism with an additional relative input on the red coupler

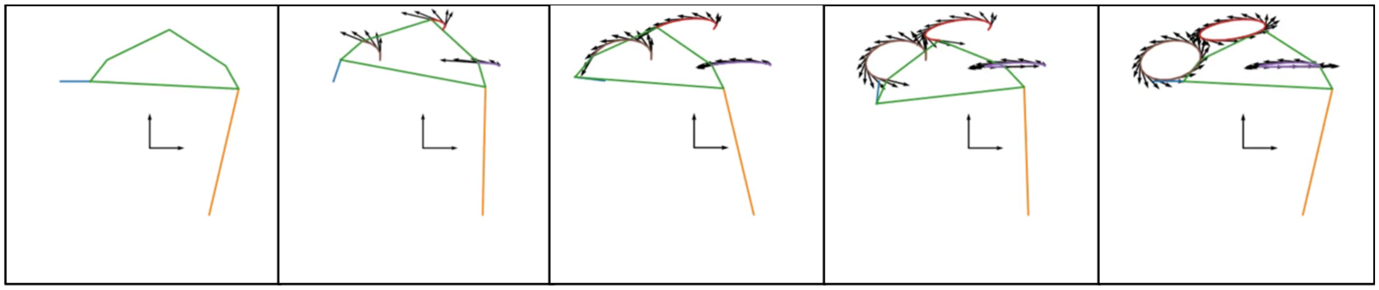
Fig. 3. From left to right, frames 0, 5, 10, 15, and 20 of a 20 frame animation captured from a planar crank-rocker mechanism motion.

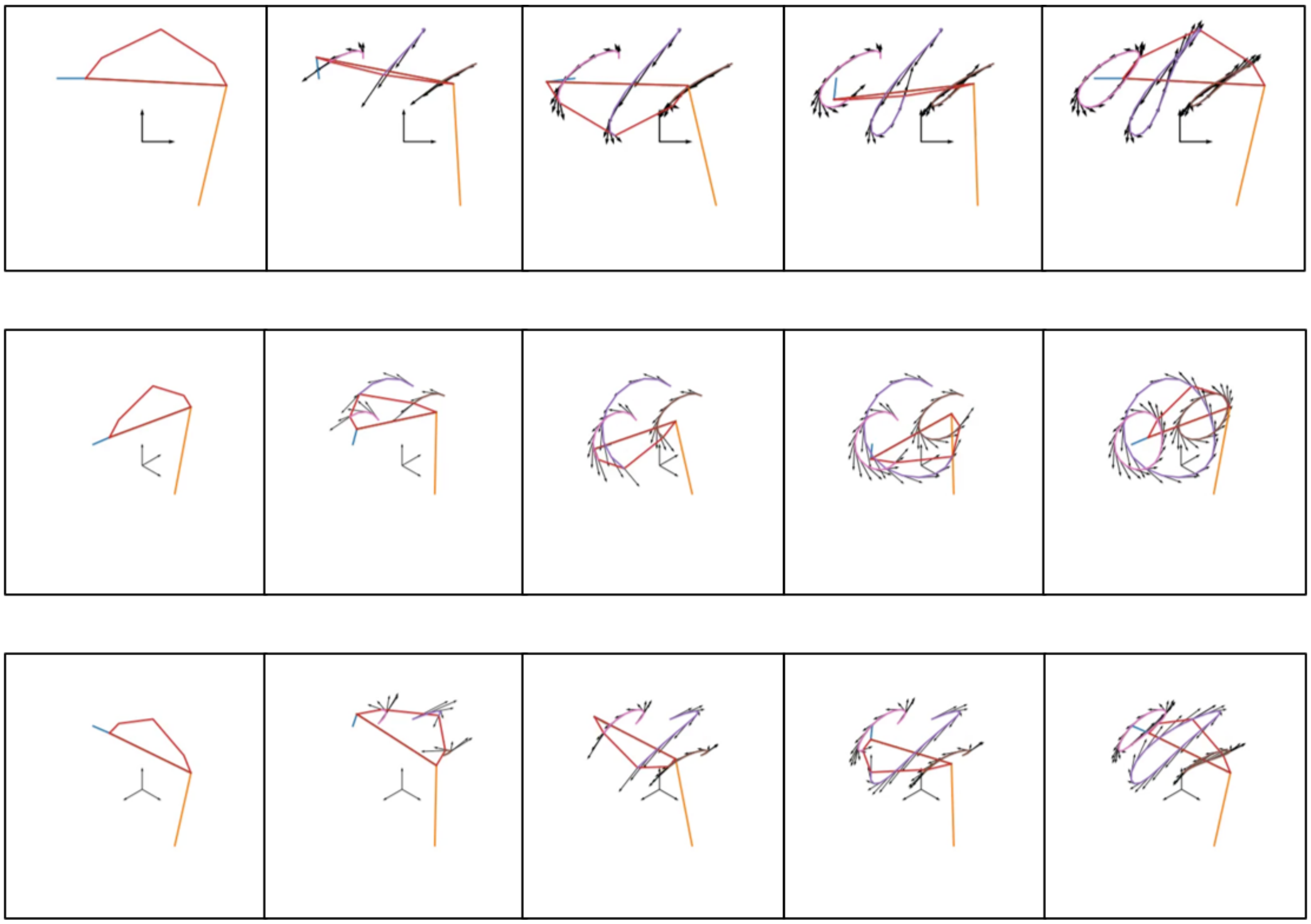
Fig. 4. Equivalent to Fig. 3 for a compound crank-rocker, wherein the coupler also rotates around its local x-axis. Top is an X-Y projection, middle is an isometric projection from $\langle -1, 1, 1\rangle$, bottom is an isometric projection from $\langle 1, 1, 1\rangle$.

member, causing it to rotate about its local x-axis. The solution is shown from three directions to emphasize the complex three-dimensionality of the output curves. Fig 5 shows a representative detail frame. The full animations are available on the project git repository.

4) Programming Interface: Because the GCS is intended to integrate with curve parameterization and generative design tools, significant thought has been put into the programming interface. A typical mechanism initialization has the following steps.

1) Initialize Mechanism Object
2) Initialize Member Objects with Associated Shape Objects
3) Add Member Objects to Mechanism
4) Initialize and Add Constraint Objects to Mechanism
5) Determine, Initialize, and Add Necessary In-

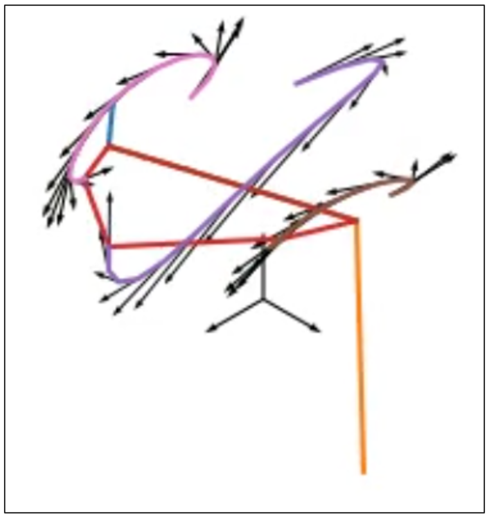

Fig. 5. A representative detail from from Fig. 4

put Constraints to Generate Desired Motion
6) Select Time Step Size and Solve

Examples of these steps can be found in `main.py` in the project git repository.

Notably, while the instantiation processes require knowledge of the internal mechanism representation, analyzing mechanism outputs does not. Mechanism members and output curves are exported as generic Shape and Curve objects, which encode position and velocity information as necessary, but contain no specifics regarding the internal mechanism data-structure or solver. Thus, these outputs could, in practice, be analyzed by and used for an entirely separate CAD or generative design task.

C. Future Developments

At this stage of development, the GCS is capable of convergence to solutions for generic, well-defined, three-dimensional mechanisms. It can solve under-constrained, over-constrained, and well-constrained geometric constraint problems. That being said, it does not run in real-time, with the six-bar mechanism shown in Fig. 2 requiring slightly more than one second to converge to a solution. Analytical derivation of the Jacobian of the constraint cost functions would forgo the need for finite difference approximation, drastically decreasing the necessary number of cost function evaluations. It is thus likely that this would greatly improve run-time. It can also be expected that implementation of the combined graph-based and Newton-Raphson to Homotopy Continuation solver described by Peng. et. al. would improve convergence times. [15] This will require degree-of-freedom analysis of each constraint currently implemented. Alternatively, because the current standard constraints all have degree three, they could be broken into sub-constraints, and these simpler options analyzed instead.

The GCS would also likely benefit from a more generalized 3D Shape representation, perhaps based on existing standard filetypes, such as `.obj`. This would greatly improve visualization without impacting solution convergence times.

III. Curve Parameterization

The curve parameterization system implemented for this project is a proof-of-concept subset of the features described by Coros et. al. [13] The major structure of the algorithm is the same. The curves are resampled in equidistant segments and then Principal Component Analysis, implemented using the open-source `scikit-learn` library, is used to normalize, orient, and scale each curve to a normalized space. Fig. 6 shows this process.

At this point, the curve features are calculated and can be compared to others. Notably, because the output curves obtained from the GCS are three-dimensional, the ellipticity feature is expanded to three features: each being the ellipticity of the curve projected onto its local x-y, y-z, and x-z planes. In addition, curve area is not calculated, nor are curve intersections, nor per-point curve distances and angles. Extension of these features to three dimensions is not entirely intuitive, and further experimentation is required to develop a comprehensive feature-set for comparison and generation.

With the currently implemented feature set, f_{curve}, the curves shown as outputs in Fig. 4 give $|f_{middle} - f_{left}| = 2.18$, $|f_{middle} - f_{right}| = 3.11$, and $|f_{left} - f_{right}| = 4.08$. That is, the middle and left curves are most similar, and left and right curves are most different. This aligns with expectations upon viewing the normalized curve comparisons in Fig. 7.

IV. Graphical User Interface

The current graphical user interface is based on the open-source `matplotlib` plotting library. However, its public programming interface is left intentionally abstracted from the details of both `matplotlib` and of the GCS, relying instead on

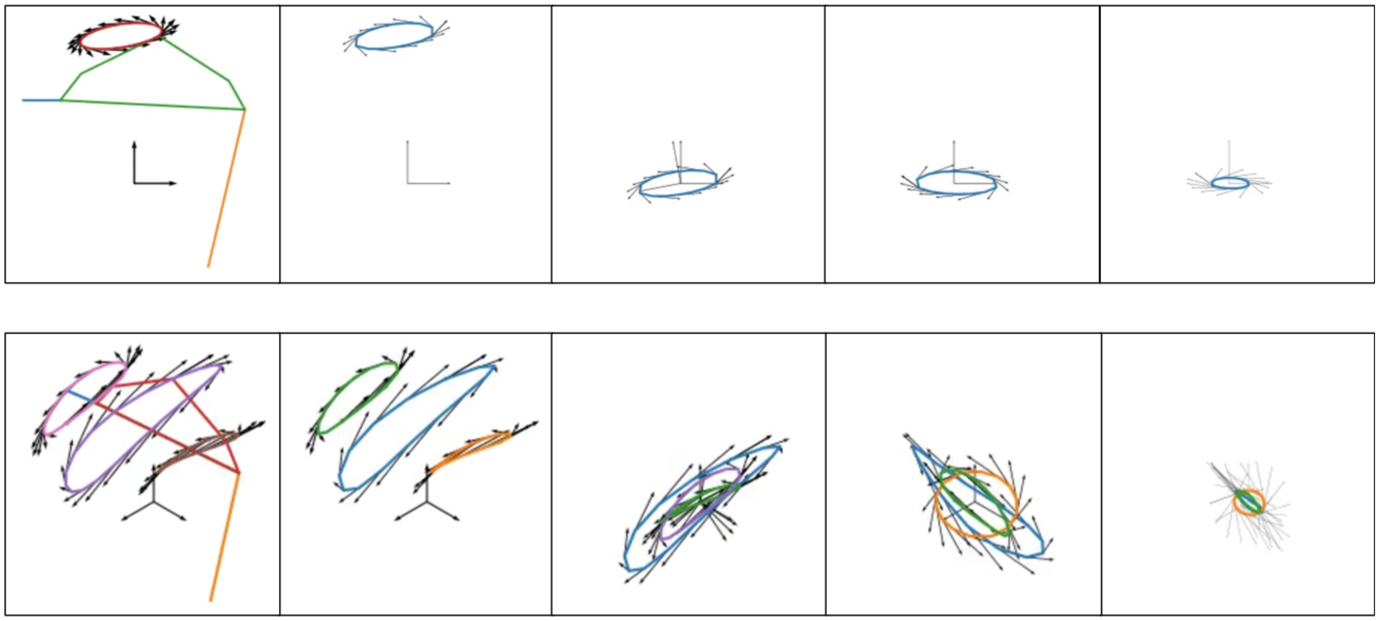

Fig. 6. The process of Principle Component Analysis and Curve Normalization. From left to right, (1) the original output curves, (2) the equidistant resampled curves, (3) the averaged curves with principle axes shown, (4) the curves rotated to align their principle axes with the global coordinate system, and (5) the curves normalized by their lengths along their major principle axes. The top is the middle output from Fig. 3; the bottom is an overlay of all outputs from Fig. 4.

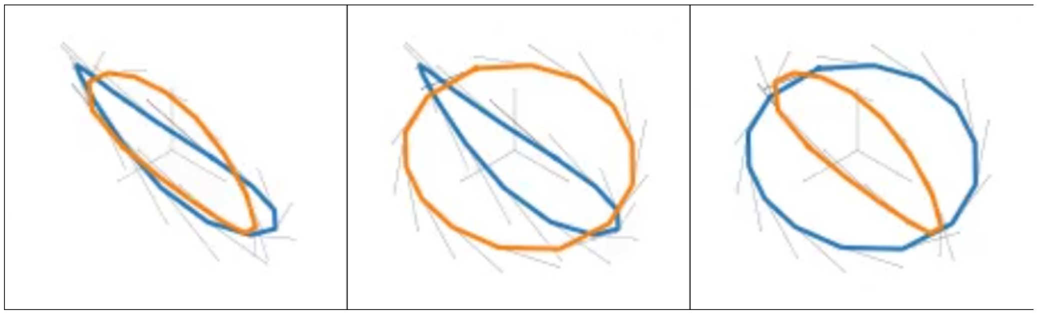

Fig. 7. Normalized curve comparisons for the outputs of Fig. 4. From left to right, (1) comparison of middle (blue) and left (orange) outputs, (2) comparison of middle (blue) and right (orange) outputs, and (3) comparison of right (blue) and left (orange) outputs. Note that the first comparison is the closest, qualtiatively aligning with the results of feature comparison.

generic Shape and Curve data-structures. As such, the system is designed for extsibility and ease of future development.

V. Conclusion

This project thus involved the development of a generic three-dimensional geometric constrain solver for mechanism creation and output analysis. This work is a direct improvement on the constraint solver developed by Coros et al. [13] This solver, in combination with the representative curve parameterization scheme could be used with the parameter space exploration and continuous optimization techniques from [13] to increase their generative possibilities. They thus provide the tools for automated generation of three-dimensional, generic mechanisms.

Future work includes actual implementation of the machine-learning processes described by Coros et. al. and completion of the curve comparison algorithm. With this, the system would be a complete, self-contained generative design software package.

References

[1] "Generative Design," *Autodesk*. [Online]. Available: https://www.autodesk.com/solutions/generative-design. [Accessed: 30-May-2021].

[2] "Enhance Your Design Workflow with Generative Design," *Solidworks*. [Online]. Available: https://www.solidworks.com/media/enhance-your-design-workflow-generative-design. [Accessed: 30-May-2021].

[3] "Refinery Toolkit," *Dynamo*. [Online]. Available: https://dynamobim.org/refinery-toolkit/. [Accessed: 30-May-2021].

[4] N. Peters, "Enabling Alternative Architectures : Collaborative Frameworks for Participatory Design," M.Des. thesis, Grad. School of Design, Harvard Univ., Cambridge, MA, 2018.

[5] "Generative Design," *Siemens*. [Online]. Available: https://www.plm.automation.siemens.com/global/en/our-story/glossary/generative-design/27063. [Accessed: 30-May-2021].

[6] D. G. Olson, A. G. Erdman, D. R. Riley, "A Systematic Procedure for Type Synthesis of Mechanisms with Literature Review," *Mechanism and Machine Theory*, vol. 20, no. 4, pp. 285-295, 1985, doi:10.1016/0094-114X(85)90033-3.

[7] N. J. Mitra, Y-L. Yang, D-M. Yan, W. Li, M. Agrawala, "Illustrating How Mechanical Assemblies Work," *ACM Trans. on Graph.*, vol. 29, no. 4, Jul 2010, Art. no. 58, dio:10.1145/1778765.1778795.

[8] S. Kota, S-J. Chiou, "Conceptual Design of Mechanisms Based on Computational Synthesis and Simulation of Kinematic Building Blocks," *Res. in Eng. Design*, vol. 4, pp. 75-87, Jun 1992, dio:10.1007/BF01580146.

[9] S-J. Chiou, S. Kota, "Automated Conceptual Design of Mechanisms," *Mechanisn and Machine Theory*, vol. 34, no. 3, pp. 467-495, Apr 1999, dio:10.1016/S0094-114X(98)00037-8.

[10] J-K. Gui, M. Mäntylä, "Functional Understanding of Assembly Modelling," *Comput. Aided Design*, vol. 26, no. 6, pp. 435-451, Jun 1994, dio:10.1016/0010-4485(94)90066-3.

[11] L. Zhu, W. Xu, J. Snyder, G. Wang, B. Guo, "Motion-Guided Mechanical Toy Modeling," *ACM Trans. on Graph.*, vol. 31, no. 6, Nov 2012, Art. no. 127, dio:10.1145/2366145.2366146.

[12] V. Megaro, B. Thomaszewski, M. Nitti, O. Hilliges, M. Gross, S. Coros, "Interactive Design of 3D-Printable Robotic Creatures," *ACM Trans. on Graph.*, vol. 34, no. 6, Oct 2015, Art. no. 216, dio:10.1145/2816795.2818137.

[13] S. Coros, B. Thomaszewski, G. Noris, S. Sueda, M. Forberg, R. W. Sumner, W. Matusik, B. Bickel, "Computational Design of Mechanical Characters," *ACM Trans. on Graph.*, vol. 32, no. 4, July 2013, Art. no. 83, dio:10.1145/2461912.2461953.

[14] R. Joan-Arinyo, "Basics on Geometric Constraint Solving," in *XIII Encuentros de Geometria Computacional*, Jan 2009.

[15] X. Peng, K. Lee, L. Chen, "A Geometric Constraint Solver for 3-D Assembly Modeling," *Int. J. of Adv. Manuf. Technol.*, vol. 28, pp. 561-570, 2006, dio:10.1007/s00170-004-2391-1.

[16] C. B. Durand, "Symblic and Numerical Techniques for Constraint Solving," Ph.D. dissertation, Comput. Sci., Purdue Univ., West Lafayette, IN, 1998.