Spatial Grammar-Based Recurrent Neural Network for Design Form and Behavior Optimization

Gary M. Stump
Applied Research Laboratory,
Pennsylvania State University,
PO Box 30,
State College, PA 16804
e-mail: gms158@psu.edu

Simon W. Miller
Applied Research Laboratory,
Pennsylvania State University,
PO Box 30,
State College, PA 16804
e-mail: swm154@psu.edu

Michael A. Yukish
Applied Research Laboratory,
Pennsylvania State University,
PO Box 30,
State College, PA 16804
e-mail: may106@psu.edu

Timothy W. Simpson
Industrial and Manufacturing Engineering,
Penn State University,
University Park, PA 16802
e-mail: tws8@psu.edu

Conrad Tucker
Engineering Design and Industrial and Manufacturing Engineering,
Penn State University,
University Park, PA 16802
e-mail: ctucker4@psu.edu

A novel method has been developed to optimize both the form and behavior of complex systems. The method uses spatial grammars embodied in character-recurrent neural networks (char-RNNs) to define the system including actuator numbers and degrees of freedom, reinforcement learning to optimize actuator behavior, and physics-based simulation systems to determine performance and provide (re)training data for the char-RNN. Compared to parametric design optimization with fixed numbers of inputs, using grammars and char-RNNs allows for a more complex, combinatorial infinite design space. In the proposed method, the char-RNN is first trained to learn a spatial grammar that defines the assembly layout, component geometries, material properties, and arbitrary numbers and degrees of freedom of actuators. Next, generated designs are evaluated using a physics-based environment, with an inner optimization loop using reinforcement learning to determine the best control policy for the actuators. The resulting design is thus optimized for both form and behavior, generated by a char-RNN embodying a high-performing grammar. Two evaluative case studies are presented using the design of the modular sailing craft. The first case study optimizes the design without actuated surfaces, allowing the char-RNN to understand the semantics of high-performing designs. The second case study extends the first by incorporating controllable actuators requiring an inner loop behavioral optimization. The implications of the results are discussed along with the ongoing and future work.

[DOI: 10.1115/1.4044398]

Keywords: conceptual design, design optimization, shape grammars, simulation-based design

1 Introduction

In this technical brief, we introduce a novel method to optimize a design's form and behavior simultaneously using spatial grammars, reinforcement learning (RL), and a physics-based evaluation. The method includes using character-recurrent neural networks (char-RNNs) to embody spatial grammars covering the design space, using physics-based simulations to evaluate system performance (both form and behavior) and RL to identify optimal control policies for system behavior with actuators.

The first motivation for this research is to move beyond parametric design optimization to design spaces with arbitrary depth and combinatorial complexity. Spatial grammars offer the ability to explore design spaces that are capable of modeling systems of varying complexity (e.g., hydraulic systems on aircraft, robotic design, vehicle component design). Two key challenges with spatial grammars are to first ensure that only feasible designs result when exercising the rules of the grammar and, secondly, to encode complex constraints into the generating rules. Overcoming these two challenges is our second motivation. The solution approach adopted for this grammar implementation/maintenance problem is to repurpose the ability of RNNs to learn natural language grammars to the domain of spatial grammars. The RNN's ability to learn syntax from training data permits encoding maintainable sets of generative rules with constraints and having the RNN learn their intersection. The third motivation supports the ability to include both form and behavior into the optimization analysis. To address this need, we need to extend the spatial grammar to permit specifying arbitrary numbers of actuators and their degrees of freedom of movement, and use RL to perform an inner loop behavior optimization in the larger design evaluation process. This enables a proper comparison of design solutions.

The remainder of this technical brief consists first a review of relevant literature, highlighting the use of neural networks for generative design, the existing research using spatial grammars for design, and the capabilities embodied in physics-based game engines. Section 3 provides a formal description of our method. Section 4 introduces the grammar used for the two case studies. Section 5 shows how the grammar can be optimized to generate high-performing designs for two case studies. Section 6 concludes the brief with a discussion of limitations and future work.

2 Literature Review

Shape grammars and similar spatial grammar-based design approaches are well studied and have been applied to architecture [1,2], mechanical design [3–7], and aircraft design [8], with recent research using grammars as a design generation method linked with local neighborhood searches and simulated annealing optimization algorithms to update Pareto optimal sets. Previous work supports optimization by iteratively applying individual grammar rules to base designs and comparing the performance of newly sampled designs with existing Pareto optimal solutions. The authors are proposing methods to compliment this research using text-based deep learning approaches to support a broad exploration of the trade space, by learning the semantics of good feasible design, with the ability to change the assembly layout, components geometries, material properties, component scales, and actuated degrees of freedom simultaneously using string representations.

Deep learning is a tool that has been used for image, speech, and text classification [9]. Recent advancements have explored the

Contributed by the Design Automation Committee of ASME for publication in the Journal of Mechanical Design. Manuscript received March 1, 2019; final manuscript received July 22, 2019; published online August 1, 2019. Assoc. Editor: Ying Liu.

possibility of extending deep learning to generative applications including generative adversarial networks [10–12], variational autoencoders [13], and char-RNNs.[1] The generative approaches use existing data to train a deep neural network and use the network to generate similar yet new data once it is trained. Approaches to text generation include using char-RNNs that learn the underlying structure of a document (i.e., its grammar) by exposing it to repeated character-level language [14]. These generative networks have successfully been taught to generate complex computer source code and quotes from the literature. This ability of char-RNNs to learn and generate string samples of varying length is used to support optimization of assembly layout designs with varying complexities. Existing examples of varying complexity problems include robotic assembly and vehicle component design [15,16].

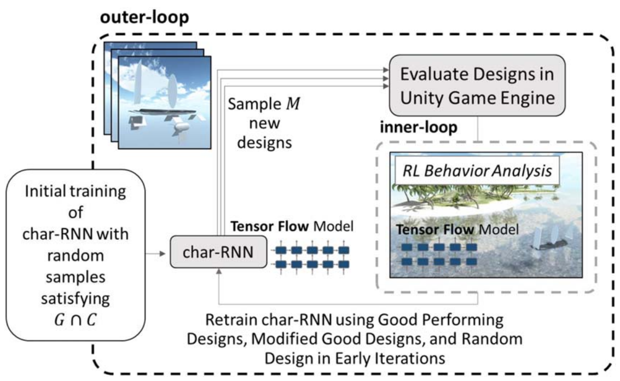
Evaluation of any design requires a model to assess its performance. To support low fidelity trade space analysis, the authors have actively been using game engines [17] (e.g., Unity3D[2]) to provide a variety of physics-based capabilities including rigid bodies with mass properties, colliders, joints, custom meshes, particle systems, and scripting for customization. Recent implementations of RL plugins have been added to video game engines [18], creating intelligent agents that are readily extensible for solving problem, behavior, and design problems simultaneously. The application of RL is unique in that agents directly learn within the environment and have been shown to learn how to act in complex games [19,20]. Our use of RL is unique in that it supports a design-learn-iterate approach to optimization; an outer loop optimizes the design assembly form, and an inner loop optimizes the behavior policy of a given design.

3 Method for Design Optimization Using a Char-RNN

The proposed method optimizes a design by iteratively retraining a char-RNN to learn the semantics of good performing designs based on results calculated using physics-based simulations. As shown in Fig. 1, the char-RNN is trained first using randomly generated strings from a grammar. Once trained, the char-RNN can generate similar as well as unique string samples. In this work, strings are representations of designs sampled from a grammar that encodes the assembly layout, component geometry, density, and actuator information describing a modular sailing craft (detailed in the next section). Next, an outer loop optimization routine is performed wherein M new samples are generated from the char-RNN that are then evaluated using a physics-based simulation. For sampled designs with actuated surfaces, reinforcement learning analysis is used in an inner loop to calculate a control policy that maximizes a performance reward in the same simulation environment.

After evaluating the generated designs, the algorithm sorts the resulting scores using higher performing configurations as a basis for creating a new training document. The training document is additionally augmented with random designs based on a diminishing schedule to promote diversity and exploration of the trade space. The char-RNN is then retrained using the updated training document, thereby exposing it to more high-performing designs, which biases future sampled designs. The algorithm repeats until a stopping criterion is met. This method applies to any design problem that can be defined using a grammar-based approach and evaluated with physics-based simulation.

This method is unique in that text-based deep learning methods are used to support trade space exploration and optimization of design assemblies with varying complexities. Once trained, the char-RNN has the ability to inexpensively generate a large variety of feasible samples for each new generation with high probability, thereby reducing the cost of sampling invalid designs from the grammars. The case study to demonstrate the method is introduced next.



outer-loop
Evaluate Designs in Unity Game Engine
Sample M new designs
inner-loop
RL Behavior Analysis
Tensor Flow Model
Initial training of char-RNN with random samples satisfying $G \cap C$
char-RNN
Tensor Flow Model
Retrain char-RNN using Good Performing Designs, Modified Good Designs, and Random Design in Early Iterations

Fig. 1 Method to design optimization using char-RNNs, grammars, reinforcement learning, and a simulation environment

4 Experimental Setup: Defining a Generative Grammar *G*

A formal context-free grammar G is used to define a set of assembly configurations using the Backus–Naur Form [21]. A sailboat design problem is used for demonstration as it includes multiple component types having unique geometry, material properties, and scale. Sailboats also have control surfaces (e.g., sails, rudders) with behaviors that can be optimized using RL. The appendix includes the grammar G, and Fig. 2 displays a feasible sailboat design based on a valid string sampled from the grammar G.

As shown in Fig. 3, the boat assembly consists of one mandatory hull, up to three sails placed on the top side of the hull, and a modular arrangement of foils, rods, floaters, and sinkers that are positioned below the hull. The hull has three sets of slots (forward, mid, and aft), which can be filled with no connections (empty), foils, or strut assemblies. If a strut assembly is added, then a connection (foil or rod) is placed into the slot and a component (floater or sinker) is attached to the end of the connection. Each component includes one set of slots positioned on the bottom of the component, where additional connections and foils can be added. This assembly structure extends recursively, resulting in complex assemblies using a small set of production rules; as a result, there are an infinite number of assembly combinations, with no fixed set of input variables to optimize.

The geometries of the sails, floaters, sinkers, and the hull are represented using underlying 3D model files and geometry construction rules assigned to characters within the grammar. The base hull 3D model database consists of 10 geometries downloaded from an open source 3D model repository,[3] as shown in Fig. 4. The 10 geometries were selected based on a combination of their variety in shape and tagged information (e.g., hull, boat, ship). Additionally, four floaters/sinkers and sail geometries are included in the database, based on tagged data and custom shape representations. To generate a larger database of similar geometries, linear particle interpolation, particle joining, and surface reconstruction algorithms are used to generate manifold objects for hulls, floater/sinkers, and sails [22]. The following examples demonstrate how geometries are defined, using a component type and database identifier.

- h1: hull (*h*) with a geometry of id 1
- s2: sail (*s*) with a geometry of id 2
- lc2: light component (*lc*) or floater with a geometry of id 2
- hc2: heavy component (*hc*) or sinker with a geometry of id 2
- h0_1_2: particle interpolation using hulls (*h*) 0 and 1 with 20% influence (Fig. 4)
- h1_4: particle union using hulls (*h*) 1 and 4 (Fig. 4)

[1]http://karpathy.github.io/2015/05/21/rnn-effectiveness/
[2]http://unity3d.com

[3]https://www.thingiverse.com/

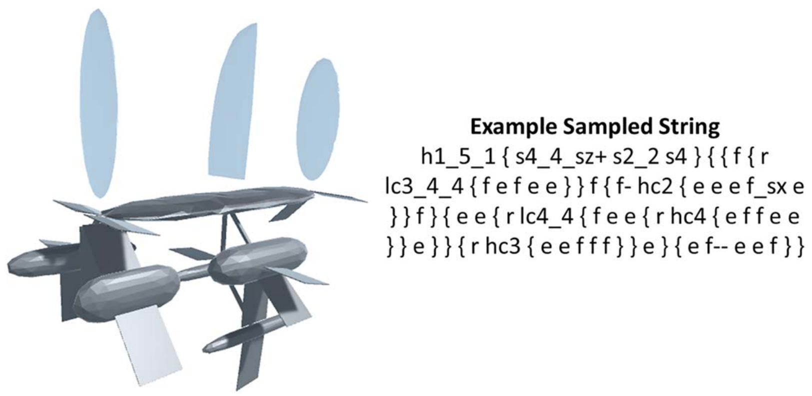

Example Sampled String
h1_5_1 { s4_4_sz+ s2_2 s4 } { { f { r lc3_4_4 { f e f e e } } f { f- hc2 { e e e f_sx e } } f } { e e { r lc4_4 { f e e { r hc4 { e f f e e } } e } } { r hc3 { e e f f f } } e } { e f-- e e f } }

Fig. 2 Valid string sampled from the grammar that represents a feasible sailboat design

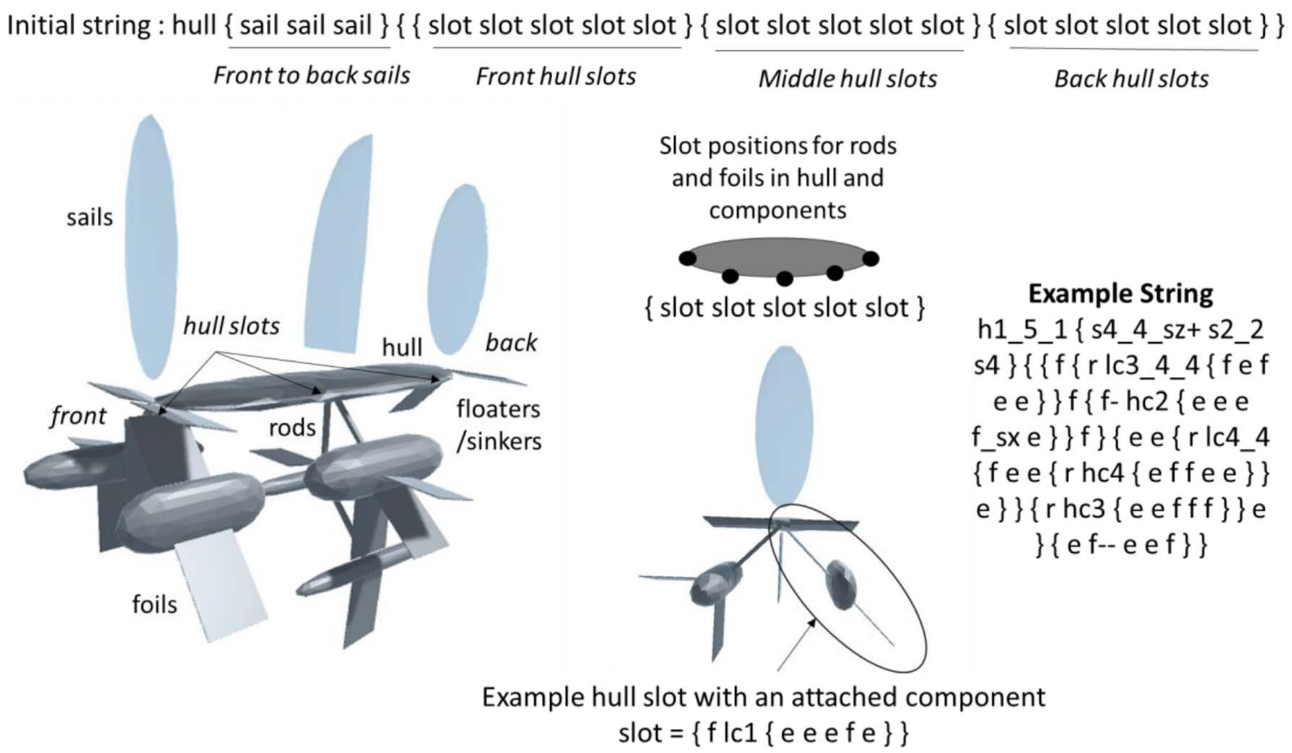

Initial string : hull { sail sail sail } { { slot slot slot slot slot } { slot slot slot slot slot } { slot slot slot slot slot } }
Front to back sails
Front hull slots
Middle hull slots
Back hull slots
sails
hull slots
hull
back
front
rods
floaters /sinkers
foils
Slot positions for rods and foils in hull and components
{ slot slot slot slot slot }
Example String
h1_5_1 { s4_4_sz+ s2_2 s4 } { { f { r lc3_4_4 { f e f e e } } f { f- hc2 { e e e f_sx e } } f } { e e { r lc4_4 { f e e { r hc4 { e f f e e } } e } } { r hc3 { e e f f f } } e } { e f-- e e f } }
Example hull slot with an attached component
slot = { f lc1 { e e e f e } }

Fig. 3 Assembly layout rules and example generated design using the string shown

Rods, foils, and empty slots are represented using *r*, *f*, and *e*, respectively. The scale of the hull, sails, foils, floaters, and sinkers are increased or decreased using + or − characters, respectively, where each character represents a 10% increment (++++ scales the geometry by +40%).

Actuators for the sailboat, which include rotatable foils and sails, are defined using behavior modifiers that identify the feasible local axis-of-rotation of each actuator surface. The feasible set of spin axes include the body-fixed *x*, *y*, and *z* axes, represented in the grammar as *_sx*, *_sy*, and *_sz*, respectively. Figure 5 displays example actuator controls in the local coordinate system of the component. Optimization of boat designs using this grammar and the proposed method is discussed next.

The grammar *G* is partially feasible in that exercising the grammar does not guarantee geometrically feasible designs that satisfy geometry constraints *C* (e.g., overlapping geometries) as shown in Fig 6. As an experiment, a random generation of 10,000 strings are sampled from the grammar *G* satisfied *C* with a 62.1% success rate. These feasible strings are then used to train a char-RNN (RNN size = 128, learning rate = 0.002, sequence length = 128, number of layers = 2, batch size = 50, number of epochs = 50, decay rate = 0.97, and model type = LSTM), and the char-RNN is sampled 10,000 times after training. Results show that 9158 of the sampled strings satisfy *G* (91.6%) and, of the 10,000 samples, 8978 strings satisfy *G* and *C* (89.8%). An important question is whether the char-RNN is learning or simply memorizing the training data. To assess this, we evaluated all (8978) the sampled strings that satisfy *G* and *C*, and determined that every generated string was unique and not found in the training data. The key results from this analysis shows that the char-RNN learns a representation that biases designs that adhere to a string-based grammar and then have the ability to sample new designs that satisfy design constraints.

5 Case Studies: Design Optimization Using Char-RNNs

5.1 Design Optimization Using Char-RNN Retraining Without RL (No Actuated Surfaces). A physics-based simulation

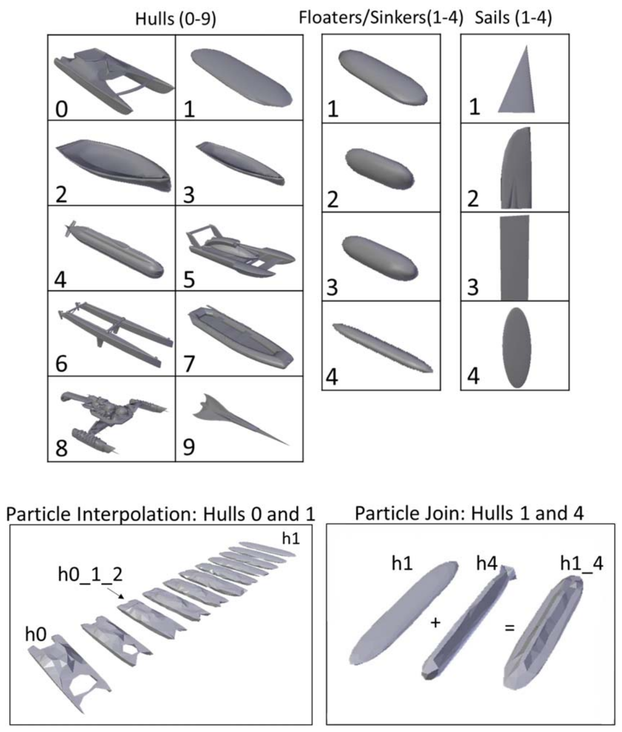

Hulls (0-9)
Floaters/Sinkers(1-4)
Sails (1-4)
Particle Interpolation: Hulls 0 and 1
h0
h0_1_2
h1
Particle Join: Hulls 1 and 4
h1
h4
h1_4

Fig. 4 Hull, component, and sail base 3D model files and example of particle interpolation and particle join between two geometries, respectively

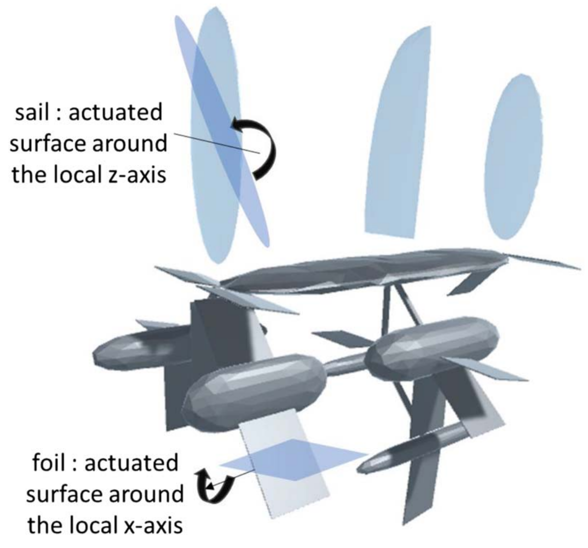

sail : actuated surface around the local z-axis
foil : actuated surface around the local x-axis

Fig. 5 Actuated sailboat surfaces (foils and sails)

environment was implemented in the Unity Game Engine to evaluate the performance of a sailboat design. The 3D model extracts energy from the physical environment (e.g., wind energy) to perform a task satisfying a performance metric. In this case study, the task is maximizing the velocity of the sailboat toward the target in a crosswind (see Fig. 7). The sailboat surface mesh is exposed to aerodynamic forces above the water plane and hydrodynamic forces below the water plane. The simulation ends when the boat reaches the target, capsizes, sinks, or reaches a time duration limit. If a boat capsizes or sinks, then it receives a minimum performance score, and these designs are removed from the training document. The simulation settings are 1000 kg/m^3 for water density, 400 kg/m^3 for the hull density, 200 kg/m^3 for the floater and sail

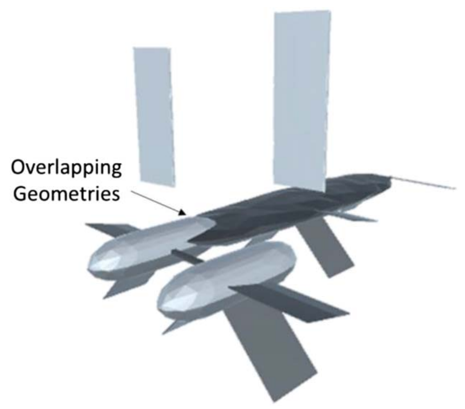


Fig. 6 Example of an infeasible assembly layout with overlapping geometries

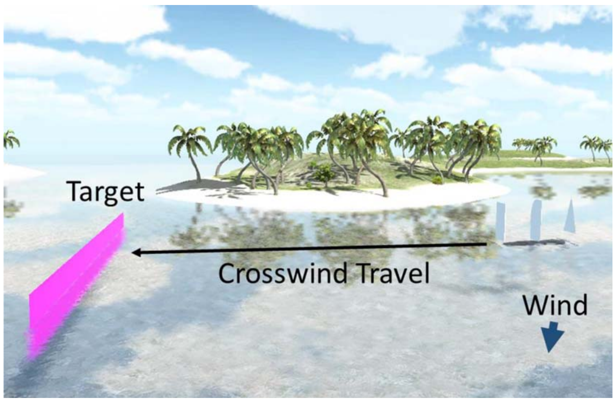


Fig. 7 Unity sailboat scene and example sailboat design evaluation

density, 1800 kg/m^3 for the sinker density, and 10 m/s for the wind velocity, and the water surface has no waves.

Based on the method shown in Fig. 1 (without the inner loop RL analysis), a char-RNN optimization analysis is performed with $N =$ 20 generations and $M = 1000$ generations. The process of retraining of the char-RNN (model = LSTM, rnn_size = 128, num_layers = 2, seq_length = 128, batch_size = 50, num_epochs = 50, learning_rate = 0.002, decay_rate = 0.97, output_keep_prob = 1.0, and input_ keep_prob = 1.0) is the key feature being evaluated in this first case study. Figure 8 displays the pseudocode for the char-RNN optimization algorithm, where the random probability (*rp*) is set to 0.5 and the mutated probability (*mp*) is set to 0.1.

Figure 9 displays tabulated and graphed data of the mean and best velocity for each generation of the optimization analysis along with the average number of designs that float with a forward velocity. The mean velocity for each generation increases during the analysis, and the best velocity shows an increasing trend with large increments in Generations 7, 8, and 16. The number of floating sailboats with forward velocity increases from Generations 1 to 12 and then reaches a steady-state behavior.

Figure 10 displays the progression of the best scoring designs during the optimization analysis for each generation. The shape of the hull changes from a double hull to a single hull in the seventh generation, where the side floater is removed. In general, the size of the hull decreases throughout the generations, thereby reducing overall drag on the sailboat with a resultant increase in speed. In generations 10 through 20, a high proportion of configuration modifications occur with sizes and placements of foils and sails, with fine-tuning of these components in the last five generations of the analysis.

char-RNN retraining optimization algorithm

```
init score_map[]
for i ← 1 to N
    init s[]
    for j ← 1 to M
        ss ← sample Char-RNN
        s.append(ss)
    for i ← 1 to length(s)
        score ← evaluateUsingSimulation(s)
        score_map[s] ← score(s)
    sort score_map by max score
    trim score_map (M/2)
    for j ← 1 to M/2
        write score_map[j]
    while training document string count < 10000
        write mutated design using score_map[j]
        if rand() < rp*(N - i)/N
            write random design
    retrain Char-RNN with training document
```

Mutated design algorithm

```
foreach w ← word in s
    if rand() < mp
        if w is foil or w is empty
            reset to emply slot
        else if w is sail
            reset to empty sail
        else if w is hull
            reset to empty hull
randomly set empty components using grammar rules
check for valid geometry
```

Fig. 8 Pseudocode of char-RNN optimization with reinforcement learning

Figure 11 shows the influence of a hull type by generation. All hull types have a similar percentage influence (10%) for the first generation. As the optimization progresses, certain hull types (1, 6, and 8) have higher probabilities as the char-RNN is biasing its sampling to these hulls, which perform better with respect to maximizing velocity. Figure 12 displays a plot showing decreasing trends in the average number of components of designs by generation, illustrating that char-RNN has learned that fewer components with a low drag hull yield better performing designs.

The key results from this analysis show that a char-RNN can be retrained to learn the semantics of boat performance that satisfy geometry constraints, based on the results shown in Fig. 9. Results depicted in Figs. 10–12 show that the proposed algorithm has a broader exploration phase in the early generations based on hull-type selection percentages and the number of components, and then exploits and refines its search around good performing designs in the later generations. This example shows that the proposed approach learns the semantics of good design based on form, to bias the assembly layout, component geometries, material properties, and scale of components to minimize drag imposed on the sailboat, thereby maximizing its velocity performance. The next case study tests the proposed method using stochastic reinforcement learning to develop a control policy while optimizing the boat design.

5.2 Design Optimization Using Char-RNN Retraining With RL (With Actuated Surfaces). In this second case study, the sailboat simulation is extended to use Unity's ML Agent Toolkit [23], allowing agents to manipulate control surfaces and learn behavior using RL. The toolkit's RL algorithm uses states, actions, and rewards to identify an optimal control policy using proximal policy optimization [18]. The RL setup for the sailboat is displayed in Fig. 13, where an actuator's reward is to have the sailboat travel crosswind until a target x-position is reached before turning and traveling downwind. Each actuator has a unique "brain" and collects state information in the Unity scene (e.g., boat orientation, boat velocity, angles of other actuated surfaces, target reached), with an action state of positively or negatively incrementing rotation about its local axis.

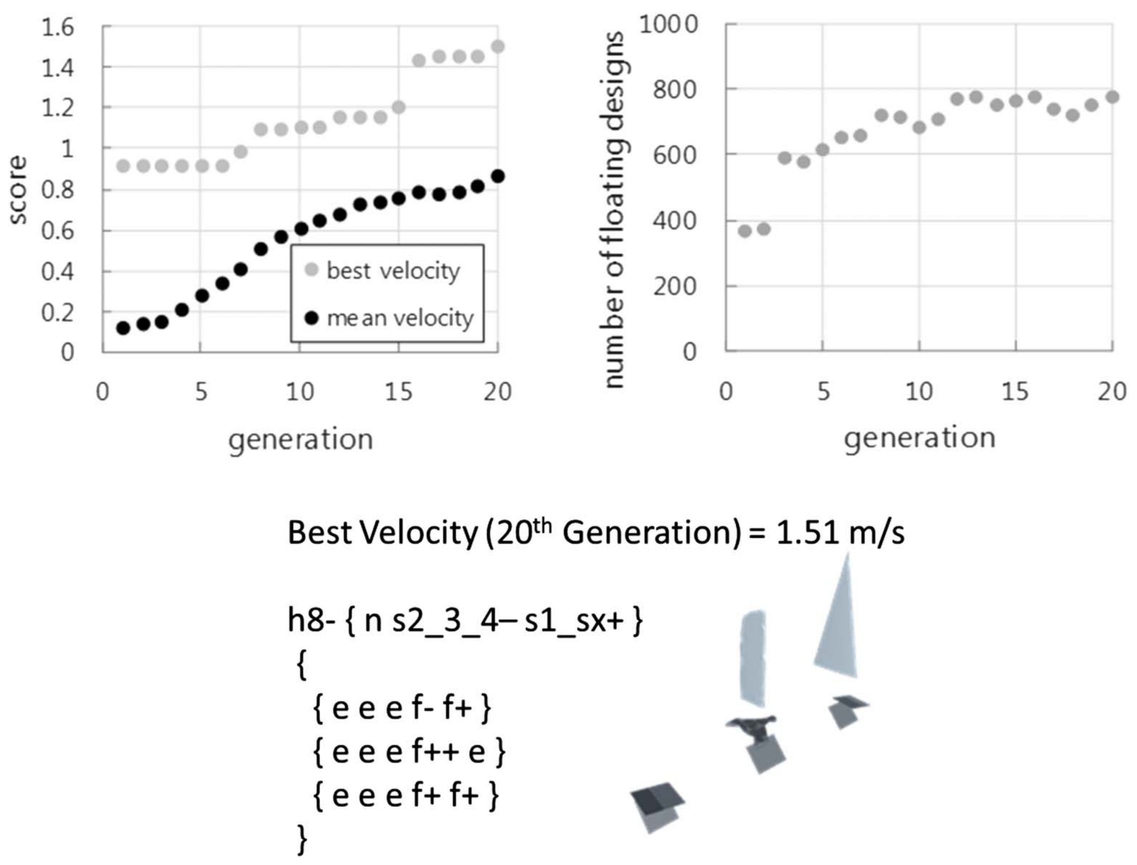


Fig. 9 Optimization analysis results for crosswind no actuator configurations

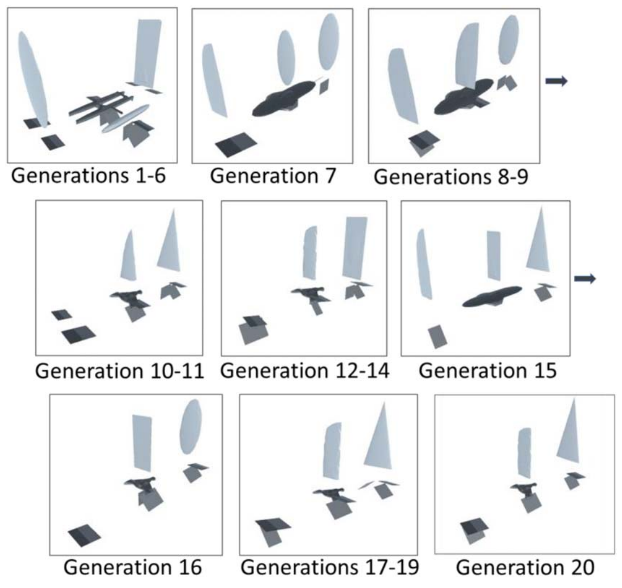


Fig. 10 Examples of the best scoring designs from the initial to the final generations

The process of retraining the char-RNN with the optimal control policy is the key feature evaluated in this second case study. The optimization analysis was performed for $N = 10$ generations, where each generation includes $M = 100$ configurations. The best design, based on form and behavior, is found in the last generation as shown in Fig. 14. The mean score for each generation increases, and the best score shows an increasing trend, with large increments in Generations 3, 7, 9, and 10.

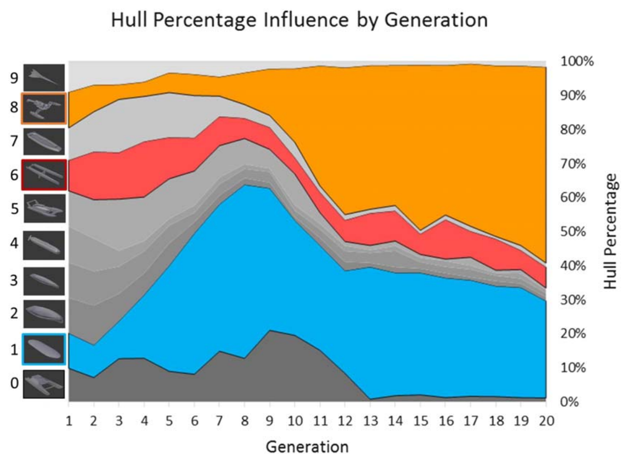


Fig. 11 Plot showing the percentage of hull-type usage by generation during the optimization analysis

Figure 15 displays the paths of each generation's best performing sailboat, identified by its generation number. All paths start at the origin and travel left in the crosswind direction until reaching a target x-position, after which the sailboat's reward changes to the downwind direction. For the best scoring sailboats in generations 1–4, the sailboats display good crosswind behavior without a turn to travel downwind. The best design in generations 5–6 results in a turn based on its form with no actuators. The resulting turn improves for generations 7–9, where a middle sail rotates horizontally about its x-axis. The best performing configuration in generation 10 includes one actuator sail, which rotates around its vertical axis.

For the best performing design, Fig. 16(a) shows the comparison of its travel path with and without the actuator behavior

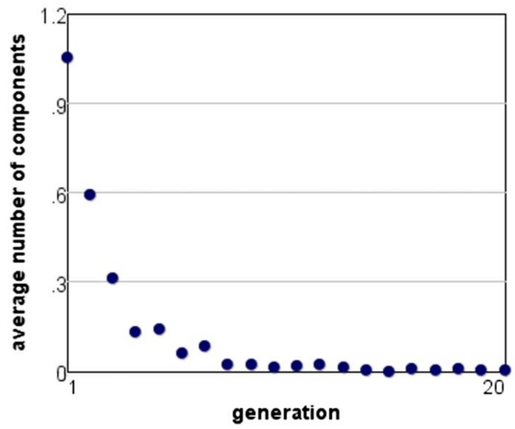
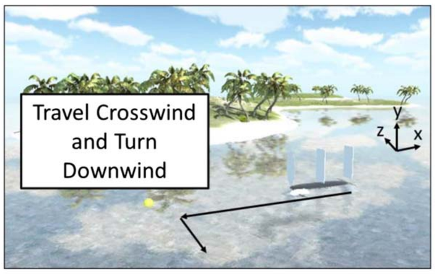


Fig. 12 Average number of components by generation



State
- Angles of all components with behavior, (angle_deg/90 : [-1, 1])
- Forward x component of the hull, [-1,1]
- Forward z component of the hull, [-1,1]
- Upward y component of the hull, [-1,1]
- Velocity x magnitude of the hull
- Velocity z magnitude of the hull
- 0 or 1 flag if hull reached turn location

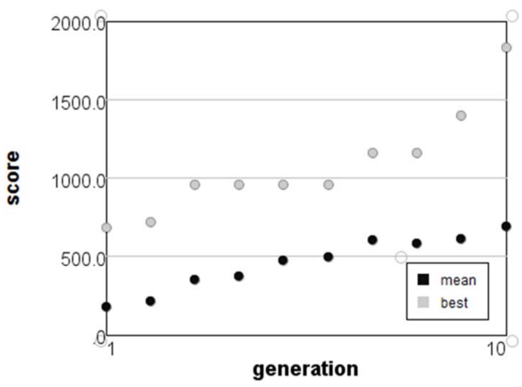
Reward
reward = 0.75 * (pref_dir.x * velocity.x + pref_dir.z * velocity.z) + 0.25 * hull.up.y;

pref_dir: Cross-wind velocity towards [-1, 0, 0] until turn location (x=-18.0) and then down-wind velocity towards [0, 0, -1]

Action
Rotation about control axis for all components with behavior
- 0 : no rotation
- 1 : negative 1^o rotation
- 2 : positive 1^o rotation

PPO Hyper parameters
```
trainer: ppo
batch_size: 64
beta: 1.0e-3
buffer_size: 4096
epsilon: 0.2
gamma: 0.9
hidden_units: 64
lambd: 0.92
learning_rate: 1.0e-4
max_steps: 5.0e4
memory_size: 256
normalize: false
num_epoch: 5
num_layers: 2
time_horizon: 32
sequence_length: 64
summary_freq: 10000
use_recurrent: false
```

Fig. 13 Reinforcement learning setup of the Unity sailboat scene to maximize a reward



Best Score (10th Generation) = 1829.9

```
h9_8_2---{ s1_sy-- s2+++ s2+++++ }
{
  { f- f– f- f- e }
  { f++ f---- f+++++ e { r lc1_2_4+ { e e e e e } } }
  { f- f+ f+ e e }
}
```

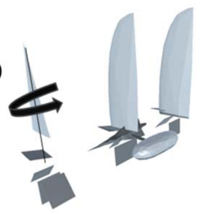


Fig. 14 Optimization analysis including reinforcement learning to calculate actuator control policies

policy. By removing the actuator policy, the travel path results in a turn, which shows that the base form of the sailboat naturally turns. In this case, the sail actuator keeps the sailboat traveling in the downwind direction, resulting in a better turn performance; the policy of the actuators is displayed in the 2D scatter plot as the angle about its local axis. The example sailboat shown in Fig. 16(*b*) displays a behavior where actuators turn the sailboat. By removing the actuator policy, the sailboat travels in the crosswind direction without a turn.

The key results from this analysis show that a char-RNN can be retrained to learn the semantics of good performing designs based on both form and behavior, since the average and best running score show an increasing trend in Fig. 14. For this case study, the form of the boat is the key driver for the best performing designs, with the actuated surfaces used to slightly modify the boat's travel path to achieve a good design. This example shows that the proposed optimization approach, having an inner loop wherein each design learns a control policy for its actuated surfaces, can learn the semantics of good design based on both form and behavior even when the performance of a design is partially affected by the results of a stochastic analysis due to calculation of the control policy.

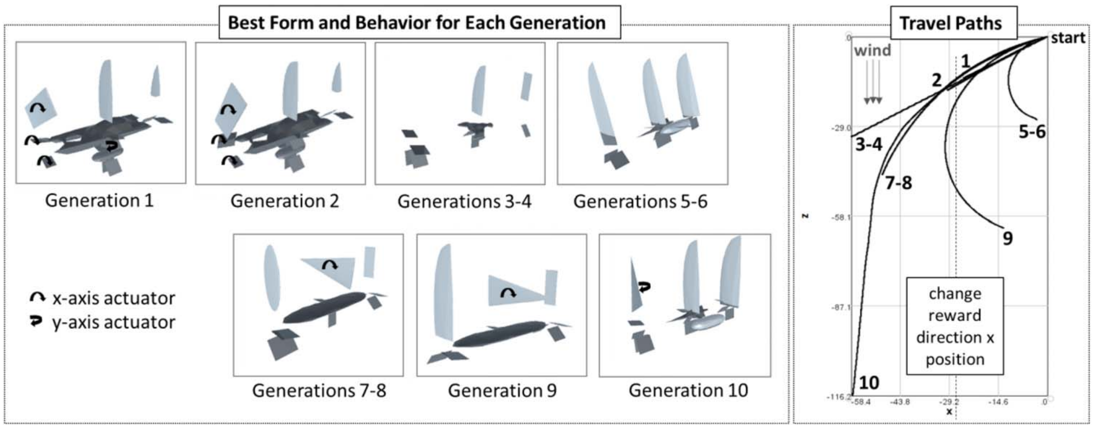
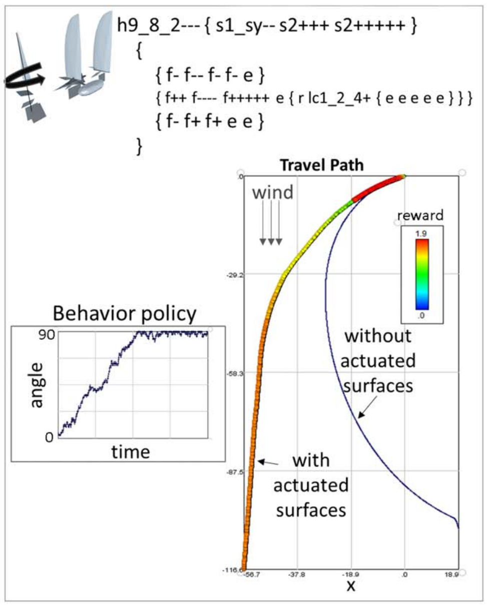


Fig. 15 Travel paths of the best performing designs in each generation

(*a*)



(*b*)

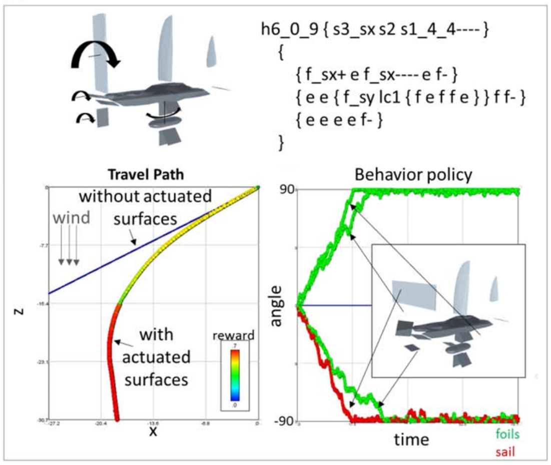


Fig. 16 Detailed behavior analysis of two example configurations: (*a*) form naturally turns the sailboat, score = 1829.9 and (*b*) actuators turns the sailboat, score = 839.2

6 Discussion and Conclusion

The grammar-based method for design optimization presented in this technical brief directly addresses issues of a restrictive design space, ease in composing performance simulations, and including behavior into the design optimization loop. By repurposing RNNs from the natural language domain, the design space can be expanded from fixed sets of parameters to combinatorically infinite sets and includes arbitrarily assigning actuators with varying degrees of freedom of motion in the design space. Analysis composition has been accelerated using physics-based game engine technology, and reinforcement learning has been used within the game engine to incorporate behavior optimization into the design evaluation process. The approach is readily extendable beyond the modular sailboat design to any complex configuration and the behavior optimization problem that can be cast as a spatial grammar.

The reinforcement learning analysis in this case study includes a small number of total population designs (1000), and we expect better performing designs with more simulation runs and more generations. The usage of parallel computing resources and Unity scene optimization will improve the ability to generate larger population sizes. The proposed method includes a large number of algorithmic tuning parameters and an approach to update the training document, and future research will focus on performing an analysis of tuning parameters and testing different approaches of updating training documents. For good performing tuning parameter sets, the performance of the proposed approach will be compared with others in previous research.

Funding Data

- DARPA (Grant No. HROO1 11820008; Funder ID: 10.13039/100000185).

Appendix

start: main mainconnections
main: hull open sail sail sail close
mainconnections: open radialconnections radialconnections radialconnections close
radialconnections: open slot slot slot slot slot close
hull: “h” SIGNED_NUMBER
| “h” SIGNED_NUMBER “_” SIGNED_NUMBER
| “h” SIGNED_NUMBER “_” SIGNED_NUMBER “_” SIGNED_NUMBER
| “h” SIGNED_NUMBER scale
| “h” SIGNED_NUMBER “_” SIGNED_NUMBER scale
| “h” SIGNED_NUMBER “_” SIGNED_NUMBER “_” SIGNED_NUMBER scale
sail: “s” SIGNED_NUMBER
| “n”
| “s” SIGNED_NUMBER behavior
| “s” SIGNED_NUMBER “_” SIGNED_NUMBER
| “s” SIGNED_NUMBER “_” SIGNED_NUMBER behavior
| “s” SIGNED_NUMBER “_” SIGNED_NUMBER “_” SIGNED_NUMBER
| “s” SIGNED_NUMBER “_” SIGNED_NUMBER “_” SIGNED_NUMBER behavior
| “s” SIGNED_NUMBER scale
| “s” SIGNED_NUMBER behavior scale
| “s” SIGNED_NUMBER “_” SIGNED_NUMBER scale
| “s” SIGNED_NUMBER “_” SIGNED_NUMBER behavior scale
| “s” SIGNED_NUMBER “_” SIGNED_NUMBER “_” SIGNED_NUMBER scale
| “s” SIGNED_NUMBER “_” SIGNED_NUMBER “_” SIGNED_NUMBER behavior scale
behavior: “_sx” | “_sy” | “_sz”
scale: “+++++” | “++++” | “+++” | “++” | “+” | “−” | “−−” | “−−−” | “−−−−” | “−−−−−”
slot: “f” | “f” scale | “e” | “f” behavior | “f” behavior scale | sa
sa: open strut nodal close
nodal: node radialconnections
node: “lc” SIGNED_NUMBER
| “lc” SIGNED_NUMBER “_” SIGNED_NUMBER
| “lc” SIGNED_NUMBER “_” SIGNED_NUMBER “_” SIGNED_NUMBER
| “mc” SIGNED_NUMBER
| “hc” SIGNED_NUMBER
| “hc” SIGNED_NUMBER “_” SIGNED_NUMBER
| “hc” SIGNED_NUMBER “_” SIGNED_NUMBER “_” SIGNED_NUMBER
| “lc” SIGNED_NUMBER scale
| “lc” SIGNED_NUMBER “_” SIGNED_NUMBER scale
| “lc” SIGNED_NUMBER “_” SIGNED_NUMBER “_” SIGNED_NUMBER scale
| “hc” SIGNED_NUMBER scale
| “hc” SIGNED_NUMBER “_” SIGNED_NUMBER scale
| “hc” SIGNED_NUMBER “_” SIGNED_NUMBER “_” SIGNED_NUMBER scale
strut: “f” | “f” scale | “r” | “f” behavior | “f” behavior scale
open: “{“
close : “}”

References

[1] Tepavcevic, B., and Stojaković, V., 2012, "Shape Grammar in Contemporary Architectural Theory and Design," Facta Univ., Ser.: Archit. Civ. Eng., **10**(2), pp. 169–178.
[2] Ruiz-Montiel, M., Boned, J., Gavilanes, J., Jiménez, E., Mandow, L., and PéRez-De-La-Cruz, J.-L., 2013, "Design With Shape Grammars and Reinforcement Learning," Adv. Eng. Inf., **27**(2), pp. 230–245.
[3] Stöckli, F., and Shea, K., 2017, "Automated Synthesis of Passive Dynamic Brachiating Robot Using a Simulation-Driven Graph Grammar Method," ASME J. Mech. Des., **139**(9), p. 092301.
[4] Helms, B., and Shea, K., 2012, "Computational Synthesis of Product Architectures Based on Object-Oriented Graph Grammars," ASME J. Mech. Des., **134**(2), p. 021008.
[5] Königseder, C., and Shea, K., 2016, "Visualizing Relations Between Grammar Rules, Objectives, and Search Space Exploration in Grammar-Based Computational Design Synthesis," ASME J. Mech. Des., **138**(10), p. 101101.
[6] Chakrabarti, A., Shea, K., Stone, R., Cagan, J., Campbell, M., Hernandez, N. V., Wood, K. L., 2011, "Computer-Based Design Synthesis Research: An Overview," ASME J. Comput. Inf. Sci. Eng., **11**(2), p. 021003.
[7] Schmidt, L. C., and Cagan, J., 1995, "Recursive Annealing: A Computational Model for Machine Design," Res. Eng. Des., **7**(2), pp. 102–125.
[8] Oberhauser, M., Sartorius, S., Gmeiner, T., and Shea, K., 2015, "Computational Design Synthesis of Aircraft Configurations With Shape Grammars," *Design Computing and Cognition '14*, J. Gero and S. Hanna, eds, Springer, Cham, pp. 21–39.
[9] Goodfellow, I., Bengio, Y., and Courville, A., 2016, *Deep Learning*, The MIT Press, Cambridge, MA.
[10] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y., 2014, "Generative Adversarial Nets," Advances in Neural Information Processing Systems NIPS(2014), Vol. 27.
[11] Wu, J., Zhang, C., Xue, T., Freeman, W. T., and Tenenbaum, J. B., 2016, "Learning a Probabilistic Latent Space of Object Shapes Via 3D Generative-Adversarial Modeling," NIPS'16 Proceedings of the 30th International Conference on Neural Information Processing Systems, Barcelona, Spain, Dec. 5–10, pp. 82–90.
[12] Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., and Bharath, A. A., 2018, "Generative Adversarial Networks," IEEE Signal Process. Mag., **35**(1), pp. 53–65.
[13] Kingma, D. P., and Welling, M., 2013, "Auto-Encoding Variational Bayes," e-print arXiv: 1312.6114, pp. 1–14.
[14] Graves, A., 2014, "Generating Sequences With Recurrent Neural Networks," e-print arXiv: 1308.0850, pp. 1–43.
[15] Diepen, M. V., and Shea, K., 2018, "A Spatial Grammar Method for the Computational Design Synthesis of Virtual Soft Robots," Proceedings of the ASME 2018 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference IDETC/CIE 2018, Quebec City, Canada, Aug. 26–29, p. V02AT03A012.
[16] Hoisl, F., and Shea, K., 2011, "An Interactive, Visual Approach to Developing and Applying Parametric Three-Dimensional Spatial Grammars," Artif. Intell. Eng. Des. Anal. Manuf., **25**(4), pp. 333–356.
[17] Dering, M., Cunningham, J., Desai, R., Yukish, M. A., Simpson, T. W., and Tucker, C. S., 2018, "A Physics-Based Virtual Environment for Enhancing the Quality of Deep Generative Designs," ASME 2018 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, Quebec City, Canada, Aug. 26–29, p. V02AT03A015.
[18] Juliani, A., Berges, V.-P., Vckay, E., Gao, Y., Henry, H., Mattar, M., and Lange, D., 2018, "Unity: A General Platform for Intelligent Agents," e-print arXiv:1809.02627.
[19] Mnih, V., Koray, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M., 2013. "Playing Atari With Deep Reinforcement Learning," e-print arXiv:1312.5602.
[20] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepe l, T., and Hassabis, D., 2017, "Mastering the Game of Go Without Human Knowledge," Nature, **550**(1), pp. 354–359.
[21] Backus, J., Bauer, F., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J., van Wijngaarden, A., and Woodger, M., 1963, "Revised Report on the Algorithm Language ALGOL 60," Comput. J., **5**(4), pp. 349–367.
[22] Bernardini, F., Mittleman, J., Rushmeier, H., Silva, C., and Taubin, G., 1999, "The Ball-Pivoting Algorithm for Surface Reconstruction," IEEE Trans. Visualization Comput. Graphics, **5**(4), pp. 349–359.
[23] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O., 2017, "Proximal Policy Optimization Algorithms," e-print arXiv:1707.06347.