Proceedings of the Design Society, Volume 6: DESIGN 2026
https://doi.org/10.1017/pds...

Generating vehicle designs using probabilistic programs and reinforcement learning

Metadata placeholder

1. Introduction

Artificial intelligence (AI) is reshaping many industries and is changing the way that we use tools to aid us in our everyday lives. Engineering design is no different, with new AI tools being used in many areas ranging from the conceptual design of buildings (Roy et al., 2022) to AI options in computer-aided design (CAD) tools (AutoDesk, 2025). In particular, the recent surge in the use of generative AI, has opened interesting avenues for generative modelling within the domain of design.

A key challenge of engineering design is data representation and generation. This challenge distinguishes it from other more prevalent applications of generative AI such as image and video generation, where an incorrectly generated pixel might have no perceivable impact on the final artifact, and large image and video datasets are widely available. In comparison, slight errors in design representations can render a design infeasible. Furthermore, large datasets of engineering designs are infrequent, and often depend on access to a scientific simulation engine. These two challenges are the motivation for FORGE (Framework for Optimization and Reinforcement-driven Generative Engineering), our proposed probabilistic programming approach to generative design.

FORGE is a framework that facilitates design generation using probabilistic programming, based on a declarative specification of the design domain. FORGE is a symbolic, declarative approach that is complementary to neural-based approaches. As such, we use FORGE to both generate large datasets of designs, and to refine design distributions using reinforcement learning. We show how we can handle a mix of discrete and continuous parameters, as well as open-ended (recursive) design, wherein a finite set of primitive components can be combined in an infinite number of ways.

In Section 2, we provide a background for the work described here. Section 3, we describe our approach in detail, in Section 4 we show results in several different domains, and in Section 5 we summarize our work and provide some concluding remarks.

2. Background

In previous works, a precursor to FORGE was used as a data generating module. This includes Anonymous (2023a), Anonymous (2023b), Anonymous (2025a). For these use-cases FORGE was a vital component to both defining the structure of the design language, as well as playing the role of the data generation module.

Anonymous (2023a) was the first instance where we initiated the development of both a design language, and a framework for sampling designs. The goal of this work was to develop a machine learning-friendly dataset of unmanned aerial vehicle designs. While the development of a such a framework was necessary to build such a dataset, the focus of the work was more on the structure of the final artifacts themselves, as well as the use of the data to benchmark ML models, such as transformers, for engineering design. Having developed this framework, we further extended it to the Webots simulator (Cyberbotics Ltd. (n.d.).) for the use in vehicle design. More recently, we extended FORGE to another flight dynamics simulator, where we are looking to design air taxis (Anonymous, 2025b). Throughout the development of this framework, we have seen FORGE as the vital "design backend", which has facilitated this line of work on generative design. This paper provides an opportunity to highlight the novelties and features that were not included in prior publications.

More generally, AI for design has seen a significant focus in recent years. These works vary from replacing individual design modules with machine learning models (e.g. computational fluid dynamics, (Shen & Alonso, 2025), (Bonnet et al, 2022)), all the way to a direct replacement of the entire pipeline (Elrefaie et al., 2025). While the use of AI in design has seen significant success, there is still the immediate need to be able to translate both inputs and outputs of AI tools back into a framework that can interface and interact with the "gold standard" simulator. As such, this is one of the purposes of introducing FORGE in this paper.

The other aspect of FORGE that we explore in this paper is in its use as a direct design module. This relates to the use of generative AI in engineering design. This both differs from the direct use of black-box optimization techniques such as Bayesian optimization (Balandat et al., 2020) and evolutionary algorithms (Pétrowski & Ben-Hamida, 2017), as well as the complete replacement of simulators with neural-based learned distributions (Cobb et al. 2024). Unlike these prior approaches, since FORGE is a parameterizable probabilistic program built from prior knowledge, it can take the form of both a generative model (unlike black-box optimization) and can be used as part of a design optimization pipeline via the use of reinforcement learning (RL), or maximum-likelihood estimation when good designs are already available. As such FORGE presents itself as a unique and useful tool in the use of AI for design.

3. Approach

We describe our approach from the point of view of generating designs for a given domain, starting from scratch. We break the process down into the following steps: Modelling (Section 3.1), Generation (Section 3.2), Evaluation (Section 3.3), and Training (Section 3.4). Note that, while we describe these aspects as different parts of an integrated system, they can also be used independently of one another. For example, Modelling-Generation-Evaluation can be used to generate training data for ML tasks; Evaluation alone can be used to evaluate manually generated designs; and Training can be coupled with a different generator (e.g., a neural network).

3.1. Modelling

The first step in our approach is to create a data model, or schema, for our designs. We use Python, and the dataclasses and typing packages as fundamental building blocks. We use UAV design as our running example.

The data model contains one top-level class (e.g., UAV) and usually contains many other classes to represent components (e.g., Propeller, Wing). It should be constructed such that any conceivable design is a valid instance of the top-level class. It is important to not inject too much designer bias and try to limit the data model to only "good" designs at this stage.

```python
@dataclass
class Wing(ABC):
    origin_x_rel: float = field(metadata={'range': (0., 1.)})
    origin_z_rel: float = field(metadata={'range': (-.5, .5)})

@dataclass
class MainWing(Wing):
    span_proj: float = field(metadata={'range': (10., 40.)})
    chord_root: float = field(metadata={'range': (.5, 5.)})
    root_cross_section: RootMainWingCrossSection
    inner_cross_sections: List[InnerMainWingCrossSection] = field(
        metadata={'length': (0,3)})
    tip_cross_section: TipMainWingCrossSection
    prop_arms: List[PropArm] = field(metadata={'length': (0,3)})

@dataclass
class VerticalTail(Wing):
    area_reference: float = field(metadata={'range': (.5, 5.)})
    taper: float = field(metadata={'range': (.1, 1.)})
    aspect_ratio: float = field(metadata={'range': (1., 5.)})
    sweep_quarter_chord_deg: float = field(metadata={'range': (0., 30.)})
    thickness_to_chord: float = field(metadata={'range': (.06, .24)})

@dataclass
class HorizontalTail(Wing):
    area_reference: float = field(metadata={'range': (0., 30.)})
    taper: float = field(metadata={'range': (.1, 1.)})
    aspect_ratio: float = field(metadata={'range': (3., 15.)})
    sweep_quarter_chord_deg: float = field(metadata={'range': (-30., 30.)})
    thickness_to_chord: float = field(metadata={'range': (.06, .24)})
    dihedral_deg: float = field(metadata={'range': (-5., 5.)})

@dataclass
class Battery:
    battery_type: str = field(metadata={'choices_key': 'battery_types'})
    voltage: float = field(metadata={'range': (300., 800.)})
    mass: float = field(metadata={'range': (300., 2000.)})

@dataclass
class Aircraft:
    main_wings: List[MainWing] = field(metadata={'length': (1,2)})
    horizontal_tails: List[HorizontalTail] = field(metadata={'length': (0,1
    vertical_tails: List[VerticalTail] = field(metadata={'length': (0,1)})
    fuselage: Fuselage
    battery: Battery
    forward_prop: RotorTemplate
    lift_prop: RotorTemplate
```

Figure 1. Figure 1. A partial data model for Aircraft designs.

```python
class MainSegment(ABC):
    pass

@dataclass
class RotatedArmSegment(MainSegment):
    armLength: float = field(metadata={'range': (50.,100.)})
    rot: float       = field(metadata={'range': (0.,270.)})
    mainSegment: MainSegment    # Recursive!

@dataclass
class CrossSegment(MainSegment):
    armLength: float = field(metadata={'range': (50.,500.)})
    sideSegment: MainSegment    # Recursive!
    centerSegment: MainSegment  # Recursive!

@dataclass
class PropArm(MainSegment):
    armLength: float = field(metadata={'range': (50.,500.)})
    motor: Motor
    prop: Prop
    flange: FlangeWithSide

@dataclass
class WingArm(MainSegment):
    armLength: float = field(metadata={'range': (50.,500.)})
    wing: Wing
    servo: Servo

class ConnectedHub(ABC):
    pass

@dataclass
class ConnectedHub3_Sym(ConnectedHub):
    mainSegment: MainSegment
    angle: float = field(default=120.)

@dataclass
class UAV:
    generator_version: str
    name: str
    hub: ConnectedHub
    fuselageWithComponents: FuselageWithComponents
```

Figure 2. Figure 2. Part of a data model for UAVs. (Recursive)

The following features are implemented by our framework: a) Primitive types: float, for continuous parameters such as chord_root; int, for discrete values or counts; str, used for enumerated types, such as battery_type (a fixed list of allowed values must be provided); and bool, to indicate presence or absence of some feature, b) Structured types: List (with a type parameter), usually to indicate an unknown number of a sub-component, e.g., prop_arms; and dataclasses, with fields containing values of the previously mentioned types, or other dataclasses. We also allow for abstract classes and subclasses, which can be used to indicate a choice of wholly different types of sub-structures (e.g., VerticalTail vs. HorizontalTail). Many other types (e.g., Union, Tuple, Optional) could easily be added to the framework if needed. Figure 1 shows an example of (part of) a data model for UAVs. The top-level class Aircraft may contain different types of wings (MainWing, VerticalTail, HorizontalTail), all of which inherit from the abstract baseclass Wing.

Finally, once the model has been created, we annotate the fields with custom metadata annotations (using the metadata attribute on dataclasses.fields). These convey information to the generative framework (described below) about the desired range of values. For float and int, we provide min and max values; for List, the min and max length of the list; for str, the set of allowed values. The ranges should not be set too restrictively; going an order of magnitude outside of "known good" value ranges

is encouraged, to allow for surprising and novel designs. A key point to keep in mind when writing the data model is that it maps out the space of design choices, and therefore the space of possible designs.

3.1.1. Recursive Data Models

The data model can be recursive, and this opens the possibility of a very open-ended kind of novelty and creativity. To create a recursive model, we must use an abstract class that has both non-recursive "leaf node" subclasses, and recursive subclasses that refer back up to the abstract class. As an example of how this can be used, consider a UAV that has a body that is built out of tubular segments, which can be connected with different kinds of joints (2-way, 3-way, etc). The end of a tube can also be connected to a component like a propeller or a wing. This can be modelled using a structure like in Figure 2. Here, we have a class ConnectedHub3_Sym (which represents the central hub of the UAV; this variant has 3 symmetric segments going out from it). This class has a field mainSegment of type MainSegment. The latter is an abstract class. It has non-recursive subclasses like PropArm and WingArm, and recursive subclasses like CrossSegment and RotatedArmSegment. These, in turn have fields with type MainSegment. This recursive structure allows us to build arbitrarily deep tree structures (of course, at some point, we must pick non-recursive classes to finish our design). Figure 3 shows some examples of the range of designs that can be generated with this approach.

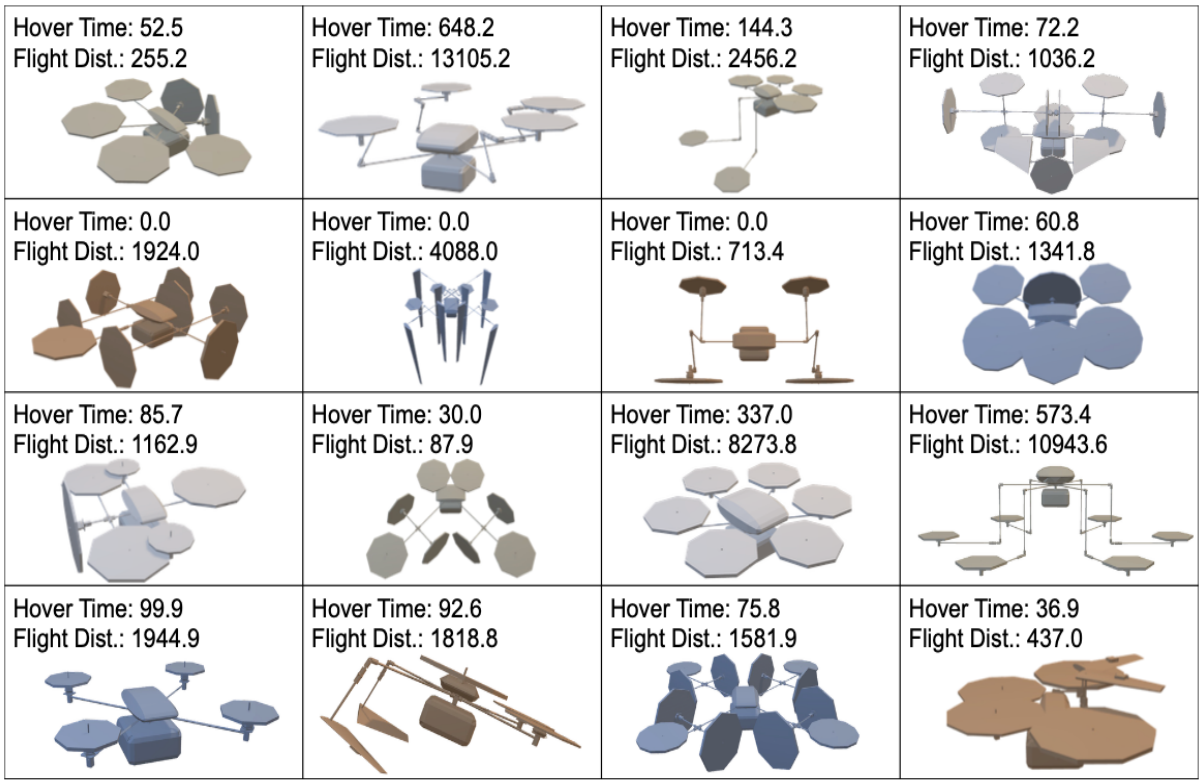

Figure 3. Creative designs, made possible using a recursive data model (Picture adapted from Anonymous (2023a).

3.1.2. Probability Distributions

The next step is to assign a probability distribution to each choice point, and to give it initial parameter values (prior). The distribution types must be chosen to match the type of choice it represents (discrete vs continuous, infinite vs limited range). We use PyTorch's built-in distribution classes, and we use PyTorch's Parameter class for the parameters of the distributions (e.g., the mean and variance of a Gaussian distribution). We discuss learning these parameters in the Section 3.4. The initialization of these parameters should not be too restrictive as we do not want to bias the generator initially

To specify the distributions and their initial values, we create a subclass of an abstract class ProbTreeModel that is built into FORGE, and implement several abstract methods, whose role is to

initialize the distribution parameters, and to return the full distribution objects, for each choice point in our data model. See Figure 4 for an example. As shown in the figure, there are methods like init_continuous_params() and get_continuous_dist() (and similarly for discrete choice, and other types of distributions). These methods take a class and field name as parameters. We create one distribution for each class-field combination, as well as for the choice of subclass of an abstract class. For many domains, we often use the same type of distribution for every case of a given type such as continuous parameters (e.g., use Gaussian for all of them), but it is possible to introduce special cases by examining the class and field parameters in these methods.

```
class AircraftGeneratorModel(ProbTreeModel):

    def init_continuous_params(self, cls: Type, field: Field):
        basename = f'{cls.__name__}_{field.name}'
        alpha = nn.Parameter(torch.tensor(0.0))
        beta = nn.Parameter(torch.tensor(0.0))
        self.set_param(f'{self.model_prefix}{basename}_alpha', alpha)
        self.set_param(f'{self.model_prefix}{basename}_beta', beta)

    def init_boolean_params(self, cls: Type, field: Field):
        param = nn.Parameter(torch.tensor(0.))
        self.set_param(f'{self.model_prefix}{cls.__name__}_{field.name}_p', param)

    def init_list_length_params(self, cls: Type, field: Field):
        min_len, max_len = get_field_metadata(field, 'length')
        avg_length = (max_len - min_len) / 2.
        param = nn.Parameter(torch.log(torch.tensor(avg_length)))
        self.set_param(f'{self.model_prefix}{cls.__name__}_{field.name}_lambda', param)

    def get_continuous_dist(self, cls: Type, field: Field) -> Distribution:
        basename = f'{cls.__name__}_{field.name}'
        alpha = self.get_param(f'{self.model_prefix}{basename}_alpha')
        beta = self.get_param(f'{self.model_prefix}{basename}_beta')
        return Beta(torch.exp(alpha), torch.exp(beta))  # make sure the values are positive

    def get_boolean_dist(self, cls: Type, field: Field) -> Distribution:
        param = self.get_param(f'{self.model_prefix}{cls.__name__}_{field.name}_p')
        return Bernoulli(torch.sigmoid(param))   # sigmoid ensures values are in [0,1]

    def get_list_length_dist(self, cls: Type, field: Field) -> Distribution:
        param = self.get_param(f'{self.model_prefix}{cls.__name__}_{field.name}_lambda')
        return Poisson(torch.exp(param))
```

Figure 4. Part of a probabilistic model for aircraft generation. We associate parametrized probability distributions and initial parameter values for each type of distribution in the data model.

The subclass of ProbTreeModel that we create is also (indirectly) a subclass of PyTorch's Module class. The use of PyTorch parameter and module classes ensures FORGE can utilize the automatic differentiation backend to enable training, as we will discuss in Section 3.4.

3.2. Generation

The next step is to generate designs from our data model and the associated probability distributions. Generation works in a top-down manner, starting from the root class (e.g., UAV), and at each choice point, sampling from the associated distribution. The code that executes this process is essentially a meta-program, or interpreter, which processes the data model using Python's reflection capabilities. The algorithm is shown in Figure 5. Along the way, we also calculate the log-likelihood of each design (this is simply the sum of the log-likelihoods of each individual sampling operation, as these are assumed to all be independent). The log-likelihoods are used by the training algorithm described in Section 3.4.

The domain-specific generator can be considered a probabilistic program, but no actual programming is needed from the designer. The meta-program of FORGE takes care of interpreting the model and generating designs. By this stage, FORGE can now generate random designs (using the forward() method of our model class) and get log-likelihoods for each generated design for the given domain.

```python
class ProbTreeGenerator():

    def _generate_dataclass_instance(self, cls: Type, prefix: str,
                                     obs: Optional[object]=None) -> Tuple[object,torch.Tensor
        self._debug(f'generate_dataclass_instance {cls}')
        new_prefix = f'{prefix}=>{cls.__name__}'
        sum_log_prob = 0.
        parts = {}
        for field in dataclasses.fields(cls):
            if meta.is_model_field(field):
                value,log_prob = self._generate_field_value(cls, field, new_prefix, obs)
                parts[field.name] = value
                sum_log_prob += log_prob
        return cls(**parts), sum_log_prob  # generate the instance

    def _generate_field_value(self, cls: Type, field: Field, prefix: str,
                              obs: Optional[Any]=None) -> Tuple[Any,torch.Tensor]:
        fname = field.name
        ftype = field.type
        self._debug(f'generate_field_value {cls}:{fname}:{ftype}')
        new_prefix = f'{prefix}=>{fname}'
        fvalue = meta.get_field_value(obs,fname)
        if meta.is_primitive_type(field.type):
            if meta.is_discrete_choice(field):
                return self._generate_discrete_value(cls, field, new_prefix, fvalue)
            elif ftype == bool:
                return self._generate_boolean_value(cls, field, new_prefix, fvalue)
            elif ftype == float:
                return self._generate_continuous_value(cls, field, new_prefix, fvalue)
            elif ftype == int:
                return self._generate_int_value(cls, field, new_prefix, fvalue)
            else:
                raise TypeError(f'Primitive type {field.type} not supported!')
        elif meta.is_parameterized_list(ftype):
            return self._generate_list_value(cls, field, new_prefix, fvalue)
        else:    # class instance
            return self.generate_instance(ftype, new_prefix, fvalue)

    def _generate_continuous_value(self, cls: Type, field: Field,
                                   prefix: str, obs: Optional[Any]=None) -> Tuple[Any,torch.T
        self._debug(f'generate_continuous_value {cls}:{field.name}:{field.type}')
        if obs is not None:
            obs_norm = torch.tensor(meta.normalize_float(field,obs))
        else:
            obs_norm = None
        # Sample continuous parameter value (e.g. arm length, wing span)
        val, log_prob = self.sampler.sample_continuous_param(prefix, cls, field, obs_norm)
        return meta.denormalize_float(field,val.item()), log_prob
```

Figure 5. Part of the probabilistic meta-program. This program traverses the data model using reflection.

```python
def ppo_loss(new_probs, old_probs, rewards, eps=0.2):
    ratio = torch.exp(new_probs - old_probs)
    clip_adv = torch.clamp(ratio, 1 - eps, 1 + eps) * rewar
    loss = torch.min(ratio * rewards, clip_adv)
    return -loss

def train(model: nn.Module, optimizer: optim.Optimizer,
          env: Callable[[List[Any],str],torch.Tensor],
          batch_size=16, episodes=1000, k_epochs=4):

    # Run PPO training loop
    for i_episode in range(episodes):
        # 1. Generate a batch of designs and their probs
        with torch.no_grad():
            designs, old_probs = model(num_samples=batch_s

        # 2. Run the designs through the environment and ge
        #    the rewards
        rew = env(designs)

        # 3. Optimization step.
        # Generate new probs -> Compute loss -> Backprop lo
        # -> repeat
        for i in range(1,k_epochs+1):
            optimizer.zero_grad()
            new_probs = model.log_prob(designs)
            loss = ppo_loss(new_probs, old_probs, rew)
            lm = loss.mean()
            lm.backward()
            optimizer.step()
```

Figure 6. Training algorithm. We learn parameters for the probabilistic model using a multi-armed bandit version of PPO.

3.3. Evaluation

The next step is to be able to evaluate our generated designs. We assume that we already have the necessary simulator(s), such as a flight dynamics simulator, to evaluate designs in some input format (but usually not directly in the data model we have been using). We also assume that our simulators return some output data that we can use to score our designs. Thus, the next steps are a) to write a translator from the high-level data model, to the low-level format that is consumed by the simulator(s), b) to write the code that invokes the simulator(s) from our Python framework, and c) process the results into a usable form (such as a pandas DataFrame), and write a reward function over the result objects. The reward function should incorporate all the requirements on design and give higher scores to better (simulated) performance, according to the relevant design metrics.

3.4. Training

While we can stop here and use our derived data model to generate a large dataset for machine learning tasks, we can also directly learn a more suitable generator by optimizing the distributional parameters. The generator produces random designs and their log-likelihoods, and a way to compute rewards for designs. Now, we can finally connect the pieces and introduce RL (Reinforcement Learning). The final output of this step is simply a better tuned generator (i.e., new values for the parameters in the generator). Subsequently, we can generate as many designs as we want from the improved generator.

At a high level, the training algorithm proceeds as follows. We generate a batch of random designs and evaluate them in our black-box simulator. The evaluation gives a reward to each design, according to our expert-designed reward function. Then, we compute a loss and update the probabilistic parameters of the generator using gradient descent, such that higher reward designs become more likely. This process repeats over many episodes, until we have learned a generator for designs that fit the intended

purpose. See Figure 6 for more details. Note that our specific use of RL is what is known as a "multi-armed bandit" problem. In other words, generating a complete design is done as one atomic action, and each episode consists of exactly one such action. Specifically, we use a bandit version of PPO (Schulman et al., 2017).

Our framework can also complete partial designs, which can be of great use for system designers, when part of the design is constrained in some way. This completion of partial designs allows design requirements to be strictly enforced, while maintaining the ability of the generator to sample from, and hence explore, the remainder of the design space. The generator then samples only the missing parameters, and the training algorithm trains those parameters, while holding the rest of the parameters fixed.

4. Results

Here, we show examples of designs generated using our approach, in several domains. Our first domain uses Webots (Michel, O., 2004), an open-source mobile robot simulation software developed by Cyberbotics Ltd. Webots is a relatively lightweight system, suitable for experimentation and algorithm development. We consider a domain of highly unconventional land vehicles, and construct an obstacle course consisting of a racetrack with multiple obstacles in it of varying height and size. The goal is to design a vehicle that can finish this track successfully and quickly. So, we set the reward function to be the sum of rewards obtained by crossing each of the obstacles discounted by the total time taken to complete the course. The data model provides several avenues for novelty in the design of vehicle, including: (a) allowing chassis to be non-rigid and comprising several components connected via flexible joints, wheels that can be of different sizes and shapes such as cylindrical, spherical or square, and varying offsets of wheels. Figure 7 shows the environment and an example of a generated vehicle design, as well as the loss and reward curves for this example, over 500 episodes of RL training.

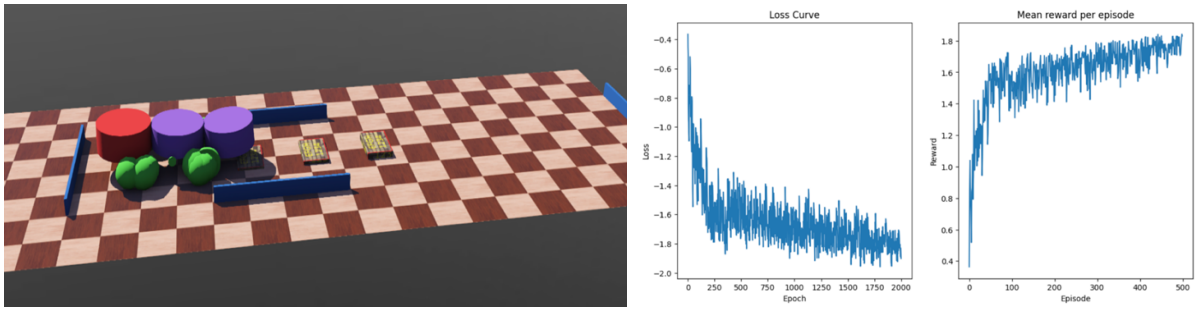

Figure 7. The Webots test environment, and an example of a generated vehicle design. Loss and reward curves across 500 episodes of RL training.

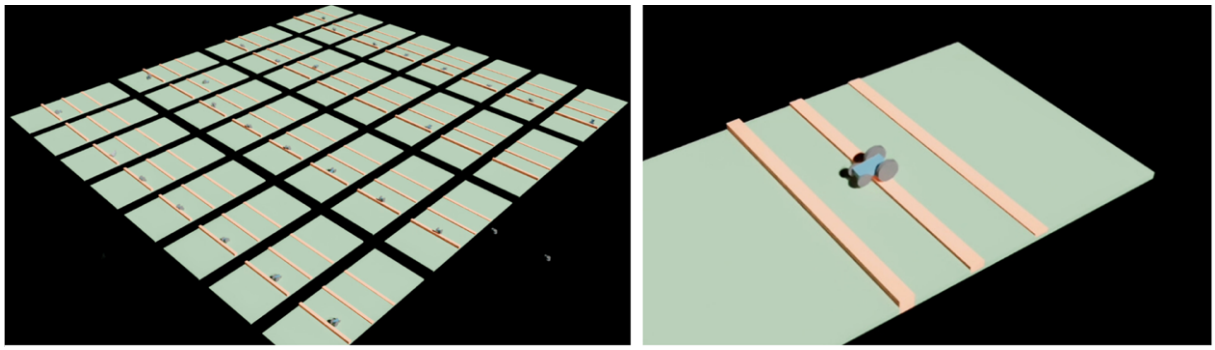

Figure 8. Parallel batch evaluation in IsaacSim.

Subsequently, we adapted the same data model to use IsaacSim (NVIDIA), a high-fidelity 3D modeling environment and physics simulator. This has the advantage that we can run an entire batch (e.g., 32 designs) simultaneously, with GPU-acceleration (see Figure 8). This alleviates the CPU bottleneck of running multiple parallel complete instances of a simulator like Webots.

We also looked at the domain of rocket design. Here, we made use of the RocketPy simulator (Ceotto et al., 2021). We ran different experiments, with reward functions to optimize for max apogee, precision landing, and target altitude. Figure 9 shows an example of generated rocket design.

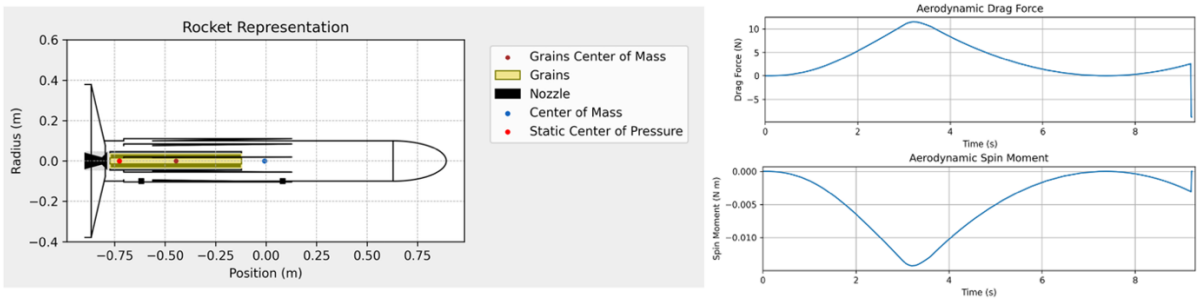

Figure 9. A rocket design from RocketPy (left), and sample output data from the simulator (right).

Figure 10. Examples from generated air taxis using FORGE for machine learning training data.

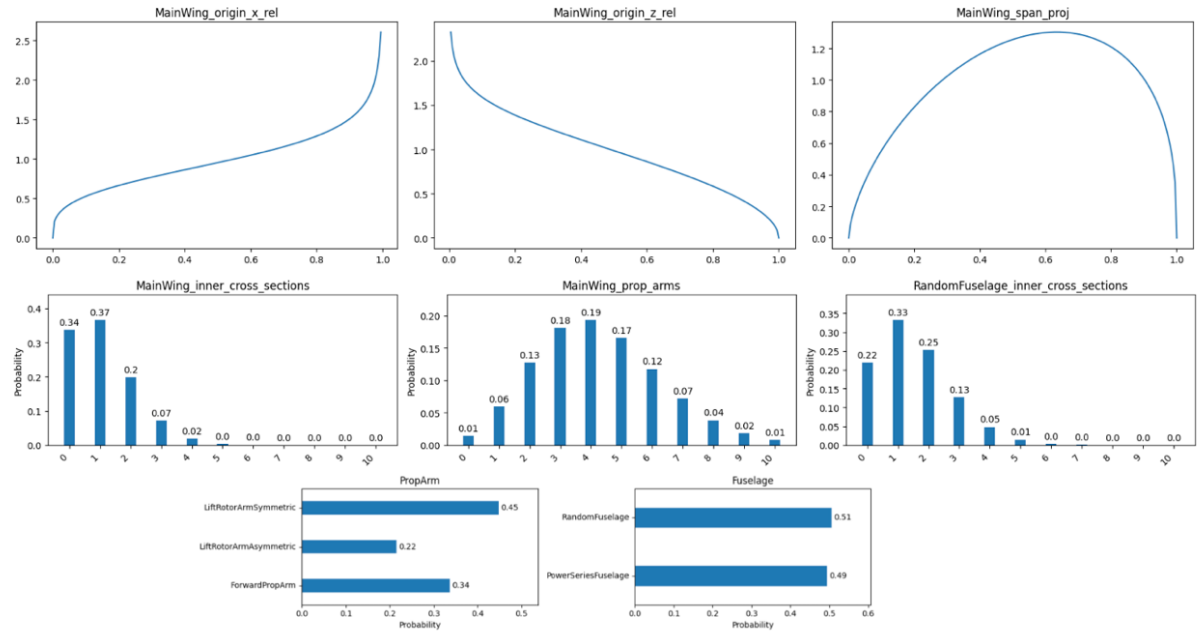

Figure 11. Examples of learned probability distributions for continuous, list-length, and subclass-choice parameters, in the UAV domain.

In our most recent work (Anonymous, 2025b), we connected FORGE to an air taxi flight dynamics simulation pipeline based on Suave (MacDonald et al., 2017). Figure 10 shows some of the designs produced by FORGE. These designs show the variations in both the individual components of the aircraft, such as wing dimensions, and the global properties of the design, such as the number of wings and the configuration of propulsion systems.

We also applied our RL pipeline to this domain, where we optimized for lift over drag (our best design achieved a very competitive L/D ratio of 14.7). In Figure 11, we show some of the probability distributions learned through this method. On the first row, we see examples of continuous distributions (note that the x axis is normalized to map the full [min,max] range of each field to [0,1]). In the second row of the figure, we see examples of list-length distributions. The third row shows some of the subclass-

choice distributions. The learned distributions describe several interesting trends. Firstly, the distribution of the wing x position is biased towards the upper end, indicating that the wings should be on average towards the back of the input range for aircraft stability. Secondly, the optimal span is found to be around 60% of the maximum span. Increasing the span decreases the drag induced by the lift of the aircraft but increases the drag due to air resistance (known as profile drag). At lower speeds, induced drag dominates, so to maximize the lift and minimize the drag, the algorithm is incentivized to increase the span until the skin friction drag or structural constraints dominate. Finally, the number of inner wing cross-sections is aligned with the numbers of panels on most aircraft, with most general aviation aircraft being one panel (0 inner cross sections), and commercial aircraft being two or three panel wings (1 or 2 inner cross sections). The full model includes about 100 distributions (and each distribution has either one or two parameters). Note that the entire learned model is fully interpretable (unlike neural models), and we can immediately see the meaning of each parameter and distribution.

5. Conclusions

In this paper we have introduced FORGE, a probabilistic framework for engineering designs. We have focused on the illustrative examples of vehicle designs, whereby we have shown how one adapts FORGE to varying design domains. We then demonstrated how the framework works, in the setting up of the data model, and the ability to generate a large variety of designs thereafter. At this point, one can use these generated designs along with the domain-specific simulator to build a large dataset for machine learning tasks as we have done in our prior works. Additionally, we have shown how we can directly optimize over the data model's probabilistic distributional parameters to achieve highly performant designs. This optimized design distribution is a more interpretable generative model than neural-based generative distributions. In future work, we would like to combine refined probabilistic models from our FORGE framework with deep generative models for novel design domains.

Acknowledgements

This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-23-C-0519 and HR0011-24-9-0424, and the U.S. Army Research Laboratory under Cooperative Research Agreement W911NF-17-2-0196. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force, DARPA, the U.S. Army Research Laboratory, or the United States Government.

References

Anonymous (2023a). *Details omitted for double-blind review.*

Anonymous (2025a). *Details omitted for double-blind review.*

Anonymous (2025b). *Details omitted for double-blind review.*

Autodesk. (2025). Generative design: Streamlining iteration and innovation. Retrieved from https://www.autodesk.com/solutions/generative-design

Balandat, M., Karrer, B., Jiang, D. R., Daulton, S., Letham, B., Wilson, A. G., & Bakshy, E. (2020). BoTorch: A framework for efficient Monte-Carlo Bayesian optimization. In Advances in Neural Information Processing Systems, 33 (NeurIPS 2020). Retrieved from http://arxiv.org/abs/1910.06403

Bonnet, F., Mazari, J., Cinnella, P., & Gallinari, P. (2022). Airfrans: High fidelity computational fluid dynamics dataset for approximating reynolds-averaged navier–stokes solutions. Advances in Neural Information Processing Systems, 35:23463–23478.

Ceotto, G.H., Schmitt, R.N., Alves, G.F., Pezente, L.A., & Carmo, B.S. (2021). RocketPy: Six Degree-of-Freedom Rocket Trajectory Simulator. Journal of Aerospace Engineering, 34(6). https://doi.org/10.1061/(ASCE)AS.1943-5525.0001331

Cobb, A. D., Matejek, B., Elenius, D., Roy, A., & Jha, S. (2024, March). Direct amortized likelihood ratio estimation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(18).

Cyberbotics Ltd. (n.d.). Webots: Open-source mobile robot simulation software. Retrieved from http://www.cyberbotics.com

Elrefaie, M., Qian, J., Wu, R., Chen, Q., Dai, A., & Ahmed, F. (2025). AI Agents in Engineering Design: A Multi-Agent Framework for Aesthetic and Aerodynamic Car Design. *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference.* Issue 89237.

MacDonald, T., Clarke, M., Botero, E.M., Vegh, J.M., & Alonso, J.J. (2017). SUAVE: An Open-Source Environment Enabling Multi-Fidelity Vehicle Optimization. *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference.* doi: 10.2514/6.2017-4437

Michel, O. (2004) Webots: Professional Mobile Robot Simulation. Journal of Advanced Robotics Systems. 1(1), 39-42. http://www.ars-journal.com/International-Journal-of-Advanced-Robotic-Systems/Volume-1/39-42.pdf

NVIDIA. Isaac Sim (Version 5.1.0) [Computer software]. https://github.com/isaac-sim/IsaacSim

Pétrowski, A., & Ben-Hamida, S. (2017). *Evolutionary algorithms* (Vol. 9). John Wiley & Sons.

Roy, A., Kim, S., Yin, M., Yeh, E., Nakabayashi, T., Campbell, M., Keough, I., & Tsuji, Y., 2022, May. A learning-based framework for generating 3D building models from 2D images. 2022 IEEE Workshop on Design Automation for CPS and IoT (DESTION).

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv: https://doi.org/10.48550/arXiv.1707.06347

Shen, Y., & Alonso, J.J. (2025). Performance evaluation of a graph neural network-augmented multi-fidelity workflow for predicting aerodynamic coefficients on delta wings at low speed. *AIAA SciTech 2025 Forum*, page 2360.