Learning-Aided Design with Structured Generative Modeling

Ari Seff

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Ryan Adams

September 2021

Abstract

Humans exhibit a remarkable ability to design new tools and systems for solving problems. At the heart of this capability lies a rich suite of evolved pattern recognition modules, enabling the reuse and modification of previously existing solutions. Machine learning offers the intriguing possibility of automatically capturing the patterns that arise in various design domains, in particular via generative modeling. By leveraging the general tool of distribution approximation over data, we may hope to computationally encode common design patterns, leading to applications that enable more efficient workflows in engineering, design, and even scientific discovery.

But this promising route faces challenges. Integral to the above fields are discrete domains, such as molecules and graphs. In contrast to the Euclidean spaces where generative modeling has recently thrived (e.g., images), discrete domains often exhibit irregular structure, strict validity constraints, and non-canonical representations, presenting unique challenges to modeling. In addition, machine learning as applied to engineering still lacks maturity, and as a result there is a need for curated datasets, benchmarks, and shared pipelines.

First, we discuss a novel generative model for discrete domains, known as reversible inductive construction. Building off of generative interpretations of denoising autoencoders, the model employs a Markov chain where transitions are restricted to a set of local operations that both preserve validity and avoid marginalization over a potentially large space of construction histories. Next, we introduce SketchGraphs, a dataset and processing pipeline that contains millions of real-world computer-aided design (CAD) sketches paired with their original geometric constraint graphs. Unlike previously available CAD data, explicit supervision is provided regarding the designer-imposed geometric relationships between primitives, making the typically latent construction operations directly accessible. Lastly, we present Vitruvion, a model that is trained to autoregressively generate CAD sketches as a sequence of

primitives and constraints. Generations from the model may be directly imported, solved, and edited in standard CAD software according to downstream design tasks. In addition, we condition the model on various contexts, including partial sketches (primers) and images of hand-drawn sketches. Evaluation demonstrates the potential for this approach to aid the mechanical design workflow.

Acknowledgements

This thesis is the work of many people—those who contributed to the research directly as well as those who have supported my arbitrary pursuits over the years.

First, I would like to thank my advisor, Ryan Adams, for his invaluable guidance and mentorship. We began working together when he returned to academia after I was a couple years into my graduate work, and he helped me significantly expand the lens through which I viewed our field. He tends to have excellent insight into problems and keeps a fun lab culture. Aside from our research, I really enjoyed getting to teach with Ryan.

I also appreciate the insightful feedback I received from my thesis committee, which, aside from Ryan, consisted of Szymon Rusinkiewicz, Olga Russakovsky, Abigail Doyle, and Tom Griffiths.

I have had the opportunity to work with some truly remarkable collaborators. I extend my sincere gratitude to Alex Beatson, Wenda Zhou, Daniel Suo, Farhan Damani, Abigail Doyle, Yaniv Ovadia, Nick Richardson, Chenyi Chen, Alain Kornhauser, Jianxiong Xiao, Han Liu, Fisher Yu, Yinda Zhang, Shuran Song, and Thomas Funkhouser. I also thank Sebastian Seung with whom it was a pleasure to instruct a course on neural networks.

Serving as a community associate for three years with the Office of Student Life was a truly rewarding experience. Thank you to my fellow CAs and especially Lily Secora and Kevin Fleming for always helping us run fun events.

I am also grateful to my incredible labmates over the years for making it easy to not take work too seriously. I will miss our retreats, super smash sessions, riddles, and philosophical debates. Thank you Alex, Daniel, Andy, Jad, Sam, Sulin, Geoffrey, Yaniv, David, Greg, Jordan, Diana, and many others.

I was surprised at just how fun doing a PhD would turn out to be, and in large part I have to thank the incredible friends I made at Princeton outside of work. Thank

you Brett, Chase, Sachin, Irene, Joe, Ari, Reyhan, Sotiris, Fermi, Yair, Archit, Nina, Luca, and many others.

Prior to graduate school, I had a fellowship at the National Institutes of Health. Le Lu, my advisor there, served as my first bridge into the field of machine learning, imparting on me an appreciation for both methodology and careful applied work. Thank you Le.

Lastly, but most importantly, I would like to thank my mom, dad, sister, and Elmira for always being there for me. Without their love and unending support I would not be in the position I am today.

This work was largely supported by the National Defense Science and Engineering Fellowship, the DataX Fund from the Schmidt Futures Foundation, and generous fellowships from Princeton University.

To my family.

Contents

Abstract iii

Acknowledgements v

List of Tables xi

List of Figures xii

1 Introduction **1**

2 Discrete Object Generation with Reversible Inductive Construction **6**

2.1 Introduction 6

2.2 Reversible Inductive Construction 9

2.2.1 Learning the Markov kernel 10

2.2.2 Sampling training sequences 11

2.2.3 Fixed corrupter 12

2.2.4 Reconstruction distribution 13

2.3 Application: Molecules 15

2.3.1 Legal operations 15

2.3.2 Data 16

2.3.3 Distributional statistics 16

2.4 Application: Laman Graphs 18

2.4.1 Legal operations 20

2.4.2 Data 21

2.4.3 Distributional statistics 21
2.5 Conclusion and Future Work 22
2.6 Appendix 23
2.6.1 Geometric Distribution for Corrupter 23
2.6.2 Molecular Reconstruction Model Operations 23
2.6.3 Training Details 24
2.6.4 Sampling Details 26
2.6.5 Dataset Details 27
2.6.6 Corrupter Details 27
2.6.7 GuacaMol Benchmarks 29
2.6.8 Reconstructed vs. Corrupted Samples 29
2.6.9 Example Chains 30

3 SketchGraphs: A Large-Scale Dataset for Modeling Relational Geometry in Computer-Aided Design 34

3.1 Introduction 34
3.2 Related Work 39
3.3 The SketchGraphs Dataset 41
3.3.1 Acquisition 41
3.3.2 Geometric constraint graphs 43
3.3.3 Construction sequence extraction 44
3.4 Case Studies of Supported Applications 46
3.4.1 Autoconstrain 46
3.4.2 Generative modeling 52
3.4.3 Other potential applications 56
3.5 Conclusion and Future Work 57
3.6 Appendix 57
3.6.1 Primitive Parameters 57

3.6.2 Constraint Parameters 60
3.6.3 Example sketch constructions 63

4 Vitruvion: A Generative Model of Parametric CAD Sketches 67

4.1 Introduction 67
4.2 Related Work 70
4.3 Method 72
4.3.1 Primitive Model 73
4.3.2 Constraint Model 76
4.3.3 Context Conditioning 78
4.4 Experiments 80
4.4.1 Training Dataset 80
4.4.2 Baselines 82
4.4.3 Primitive Model Performance 82
4.4.4 Constraint Model Performance 86
4.5 Conclusion and Future Work 88
4.6 Appendix 88
4.6.1 Hand-drawn Noise Model 88
4.6.2 Experimental Details 89
4.6.3 Compression Baseline 90

5 Open Directions 92

Bibliography 95

List of Tables

2.1 Molecular property distributional statistics 19

2.2 Laman graph distributional statistics 20

3.1 Frequencies of the most common primitives and constraints 40

3.2 Frequencies of the most common values observed for length and angle 62

4.1 Primitive parameterizations 74

4.2 Primitive model evaluation 82

4.3 Evaluation of image-conditional primitive model on hand-drawn sketches 84

4.4 Constraint model evaluation 87

List of Figures

2.1 Reconstruction model processing given an input molecule 8

2.2 Corruption and subsequent reconstruction of a molecular graph . . . 9

2.3 Distributions of QED (left), SA (middle), and logP (right) for sampled molecules and ZINC. 19

2.4 The legal inductive moves for Laman graphs 20

2.5 Distributional statistics when varying expected length of corruption sequences 23

2.6 Distributional statistics for reconstructed and corrupted molecules . . 30

2.7 Example chain. The molecule is displayed after each transition of the Markov chain. 31

2.8 Example chain. The molecule is displayed every five transitions. . . . 32

2.9 Example chain. The molecule is displayed every ten transitions. . . . 33

3.1 Example sketches 35

3.2 Example sketch and a portion of its geometric constraint graph . . . 38

3.3 Relationship of the number of primitives, constraints, and sequence position 42

3.4 Relationship of total degrees of freedom and those removed by constraints 44

3.5 Construction of a dataset sketch 46

3.6 Autoconstraining a sketch 51

3.7 Random samples from the generative model 55

3.8 Distributions of sampled and training set sketch sizes 55
3.9 Degrees of freedom statistics for sampled and training set sketches . . 56
3.10 Average number of primitives and constraints of each type per sketch in training set and sampled sketches. 56
3.11 Cumulative frequency of unique length and angle parameter values . . 63
3.12 Construction of a dataset sketch 63
3.13 Construction of a dataset sketch 64
3.14 Construction of a dataset sketch 65
3.15 Construction of a dataset sketch 66

4.1 Example CAD design workflow enabled by our model 68
4.2 Factorization of CAD sketch synthesis 69
4.3 Unconditional samples from our raw primitive model. 83
4.4 Image-conditional samples 85
4.5 Primer-conditional samples 86
4.6 Hand-drawn sketches converted to editable models 87

Chapter 1

Introduction

Pattern recognition lies at the heart of humankind's remarkable ability to innovate [65]. Experts across the scientific and engineering landscape identify and exploit regularities in natural and artificial phenomena in order to create new systems and technology. Whether it is the synthetic chemist developing new drugs by building upon prevalidated molecular scaffolds, or the mechanical engineer meeting a new target functionality by modifying a previous design, new solutions often intersect, or are combinations of, previous solutions.

But design can be tedious. Consider the engineer who must apply of hundreds of precise operations in computer-aided design (CAD) software in order to concretely realize an envisioned part. Finding and specifying the desired solution resembles a guided but time-consuming search through a large combinatorial space.

What role can learning systems play to mitigate these challenges pervasive across engineering and scientific discovery? In this thesis, we take the view that learning, particularly *generative modeling*, will allow us to automatically capture the patterns and regularities exhibited by these domains, enabling the development of new and more efficient design workflows. In contrast to discriminative modeling, which typically involves predicting some privileged label (e.g., a category) given an input exam-

ple, generative modeling aims to jointly model all observed dimensions of the data. Rather than treating the domain simply as input, a generative model has something to say about the creation of the data itself.

Concretely, a generative model approximates an observed data distribution, $p_{\text{data}}(\mathbf{x})$, with some parameterized distribution, $p_\theta(\mathbf{x})$. This follows one of two forms: *explicit* density estimation, where $p_\theta(\mathbf{x})$ is directly defined and evaluated, or *implicit* density estimation, where samples may be efficiently drawn from $p_\theta(\mathbf{x})$ but tractable evaluation is not accessible.

Both of these generative paradigms have the potential to aid design processes. When direct sampling is possible, novel data may be fantasized that maintains the key characteristics of the training domain. However, rather than drawing from the full joint distribution, specific applications may target selected conditional distributions, sampling unknown dimensions given a subset of fixed variables. This may be applied, for instance, in a CAD "autocomplete" tool that suggests next construction steps to the user conditioned on a partial design. Although the trained model is not told explicitly what the user is attempting to build, the model's exposure to similar patterns during training allows it to infer completions with high probability. In the explicit density case, a "spellcheck" model could offer corrections of implausible operations (those resulting in a low probability evaluation).

The promise of generative models to aid in engineering and design draws direct inspiration from other domains where artificial data generation has proven possible. For example, generative models have been successfully applied to the synthesis of continuous data such as speech audio [87] and high-resolution images [6]. In discrete settings, large sequence models have been used to advance natural language generation tasks [9].

But in many domains, *irregular structure* often presents additional modeling challenges. Here, we use irregular structure to refer to graph-based representations that

include one (but often multiple) of the following properties: nodes with varying degrees (irregular graphs), parallel edges (multigraphs), hyperedges (hypergraphs), or heterogeneous, multi-dimensional node and edge features. Prominent examples include geometric constraint graphs (found in CAD and robotics) and molecular graphs. These domains sit in contrast to Euclidean or grid-like data (e.g., images) where modeling success has, in part, been attributed to simple inductive biases like convolution that can account for invariances of the data [7]. Analogous inductive biases in the non-Euclidean setting are not as straightforward, and there are often additional challenges such as strict validity constraints (e.g., rules constituting a valid molecule) and representation ambiguity (e.g., up to $n!$ adjacency matrices for a graph with n nodes).

In addition, machine learning as applied to engineering is a fairly nascent effort and lacks the maturity of more established disciplines like computer vision. Just as ImageNet [20] helped spur large advancements in computer vision and deep learning research by providing large-scale, high quality training data, so too is there a need for curated datasets, benchmarks, and shared pipelines to aid applied machine learning research across the engineering domains.

In this thesis, we consider how to make the overlapping fields of design, engineering, and scientific discovery amenable to learned models. We approach this question both from an algorithmic perspective, proposing a novel modeling framework for general discrete-structured domains, and via an applications perspective, building a new dataset and processing pipeline relevant to mechanical design and using it to train design assistance models. The contributions are as follows:

- In Chapter 2, we propose a novel generative model for discrete domains, which we call reversible inductive construction, that addresses some of their unique challenges. Building off of generative interpretations of denoising autoencoders [3], the model parameterizes a Markov chain where learning transitions are restricted to a set of local inductive moves that preserve object validity. In ad-

dition, by only requiring the learner to discover local modifications to discrete objects, this approach avoids marginalization over an unknown and potentially large space of construction histories. We test the proposed approach on two complex discrete domains, molecules and a restricted class of geometric constraint graphs, and show that the proposed method can effectively model these distributions while guaranteeing adherence to strict validity constraints. This work was presented in Seff et al. [77].

- In Chapter 3, we present a new dataset, SketchGraphs, intended to foster machine learning research in mechanical design. It consists of 15 million real-world CAD sketches paired with their ground truth geometric constraint graphs. Unlike previously available CAD data, the construction steps and constraints used to produce the sketches are provided, giving explicit supervision regarding the original creator's design intent. In contrast to, e.g., general modeling of voxel shapes, this data enables the training of models which operate on the same data structures used in standard CAD software, opening the door to new design workflows where samples may be directly edited. In addition to the data, we provide an open source processing pipeline and establish benchmarks for two initial use cases of the dataset: unconditional generative modeling of sketches and conditional generation of likely constraints given unconstrained geometry. This work was presented in Seff et al. [78].

- In Chapter 4, we introduce Vitruvion, a generative model trained to synthesize coherent CAD sketches by autoregressively sampling geometric primitives, with initial coordinates, and constraints that reference back to the sampled primitives. The model, trained on SketchGraphs, leverages transformer-based attention [88] to flexibly propagate information about already-present primitives and constraints to a next-step prediction module. Following sampling,

the produced geometric constraint graph is handed to a separate solver that identifies the final configuration. We demonstrate the model's ability to aid in two application scenarios when conditioned on a) partial sketches (primers) and b) hand drawings for inference of plausible complete sketches. This work is currently in preparation as Seff et al. [79].

- Finally, we conclude by identifying some interesting avenues for future work in Chapter 5.

Chapter 2

Discrete Object Generation with Reversible Inductive Construction

2.1 Introduction

Many applied domains of optimization and design would benefit from accurate generative modeling of structured discrete objects. For example, a generative model of molecular structures may aid drug or material discovery by enabling an inexpensive search for stable molecules with desired properties. Similarly, in computer-aided design (CAD), generative models may allow an engineer to sample new parts or conditionally complete partially-specified geometry. Indeed, recent work has aimed to extend the success of learned generative models in continuous domains, such as images and audio, to discrete data including graphs [95, 57], molecules [30, 52], and program source code [94, 71].

However, discrete domains present particular challenges to generative modeling. Discrete data structures often exhibit non-unique representations, e.g., up to $n!$ equivalent adjacency matrix representations for a graph with n nodes. Models that perform additive construction—incrementally building a graph from scratch [95, 57]—are

flexible but face the prospect of reasoning over an intractable number of possible construction paths. For example, You et al. [95] leverage a breadth-first-search (BFS) to reduce the number of possible construction sequences, while Simonovsky and Komodakis [83] avoid additive construction and instead directly decode an adjacency matrix from a latent space, at the cost of requiring approximate graph matching to compute reconstruction error.

In addition, discrete domains are often accompanied by prespecified hard constraints denoting what constitutes a *valid* object. For example, molecular structures represented as SMILES strings [90] must follow strict syntactic rules in order to be decoded to a real compound. Recent work has aimed to improve the validity of generated samples by leveraging the SMILES grammar [52, 17] or encouraging validity via reinforcement learning [42]. Operating directly on chemical graphs, Jin et al. [44] leverage chemical substructures encountered during training to build valid molecular graphs and De Cao and Kipf [18] encourage validity for small molecules via adversarial training. In other graph-structured domains, strict topological constraints may be encountered. For example, Laman graphs [54], a class of geometric constraint graphs, require the relative number of nodes and edges in each subgraph to meet certain conditions in order to represent well-constrained geometry.

In this work we take the broad view that graphs provide a universal abstraction for reasoning about structure and constraints on discrete spaces. This is not a new take on discrete spaces: graph-based representations such as factor graphs [50], error-correcting codes [28], constraint graphs [69], and conditional random fields [53] are all examples of ways that hard and soft constraints are regularly imposed on structured prediction tasks, satisfiability problems, and sets of random variables.

We propose to model discrete objects by constructing a Markov chain where each possible state corresponds to a valid object. Learned transitions are restricted to a set of local *inductive* moves, here defined as minimal insert and delete operations

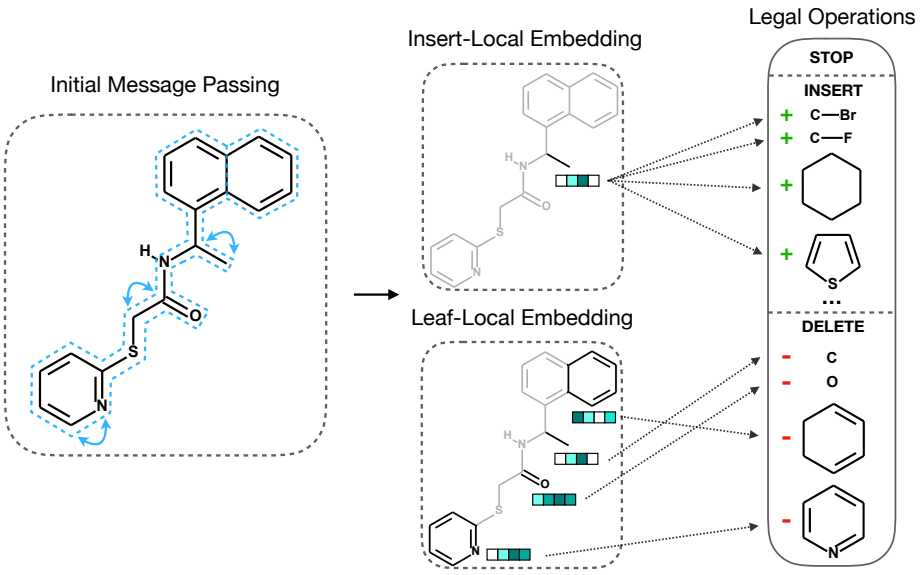


Figure 2.1: Reconstruction model processing given an input molecule. Location-specific representations computed via message passing are passed through fully-connected layers outputting probabilities for each legal operation.

that maintain validity. Leveraging the generative model interpretation of denoising autoencoders [3], the chain employed here alternatingly samples from two conditional distributions: a fixed distribution over corrupting sequences and a learned distribution over reconstruction sequences. The equilibrium distribution of the chain serves as the generative model, approximating the target data-generating distribution.

This simple framework allows the learned component—the reconstruction model—to be treated as a standard supervised learning problem for multi-class classification. Each reconstruction step is parameterized as a categorical distribution over adjacent objects, those that are one inductive move away from the input object. Given a local corrupter, the target reconstruction distribution is also local, containing fewer modes and potentially being easier to learn than the full data-generating distribution [3]. In addition, various hard constraints, such as validity rules or requiring the inclusion of a specific substructure, are incorporated naturally.

One limitation of the proposed approach is its expensive sampling procedure, requiring Gibbs sampling at deployment time. Nevertheless, in many areas of engineering and design, it is the downstream tasks following initial proposals that serve as

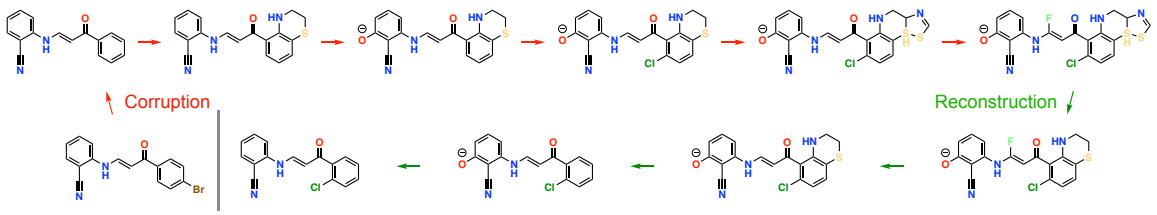


Figure 2.2: Corruption and subsequent reconstruction of a molecular graph. Our method generates discrete objects by running a Markov chain that alternates between sampling from fixed corruption and learned reconstruction distributions that respect validity constraints.

the time bottleneck. For example, in drug design, wet lab experiments and controlled clinical trials are far more time intensive than empirically adequate mixing for the proposed method's Markov chain. In addition, as an implicit generative model, the proposed approach is not equipped to explicitly provide access to predictive probabilities. We compare statistics for a host of semantically meaningful features from sets of generated samples with the corresponding empirical distributions in order to evaluate the model's generative capabilities.

We test the proposed approach on two complex discrete domains: molecules and Laman graphs [54], a class of geometric constraint graphs applied in CAD, robotics, and polymer physics. Quantitative evaluation indicates that the proposed method can effectively model highly structured discrete distributions while adhering to strict validity constraints.

2.2 Reversible Inductive Construction

Let $p(x)$ be an unknown probability mass function over a discrete domain, D, from which we have observed data. We assume there are constraints on what constitutes a valid object, where $V \subseteq D$ is the subset of valid objects in D, and $\forall x \notin V, p(x) = 0$. For example, in the case of molecular graphs, an invalid object may violate atom-specific valence rules. Our goal is to learn a generative model $p_\theta(x)$, approximating $p(x)$, with support restricted to the valid subset.

We formulate our approach, generative reversible inductive construction (GenRIC),[1] as the equilibrium distribution of a Markov chain that only visits valid objects, without a need for inefficient rejection sampling. The chain's transitions are restricted to legal inductive moves. Here, an inductive move is a local insert or delete operation that, when executed on a valid object, results in another valid object. The Markov kernel then needs to be learned such that its equilibrium distribution approximates $p(x)$ over the valid subspace.

2.2.1 Learning the Markov kernel

The desired Markov kernel is formulated as successive sampling between two conditional distributions, one fixed and one learned, a setup originally proposed to extract the generative model implicit in denoising autoencoders [3]. A single transition of the Markov chain involves first sampling from a fixed corrupting distribution $c(\tilde{x} \mid x)$ and then sampling from a learned reconstruction distribution $p_\theta(x \mid \tilde{x})$. While the corrupter is free to damage x (push x off the data manifold), validity constraints are built into both conditional distributions. The joint data-generating distribution over original and corrupted samples is defined as $p(x, \tilde{x}) = c(\tilde{x} \mid x)p(x)$, which is also uniquely defined by the corrupting distribution and the target reconstruction distribution, $p(x \mid \tilde{x})$. We use supervised learning to train a reconstruction distribution model $p_\theta(x \mid \tilde{x})$ to approximate $p(x \mid \tilde{x})$. Together, the corruption and learned reconstruction distributions define a Gibbs sampling procedure that asymptotically samples from marginal $p_\theta(x)$, approximating the data marginal $p(x)$.

Given a reasonable set of conditions on the support of these two conditionals and the consistency of the employed learning algorithm, the learned joint distribution can be shown to be asymptotically consistent over the Markov chain, converging to the true data-generating distribution in the limit of infinite training data and modeling

[1] `https://github.com/PrincetonLIPS/reversible-inductive-construction`

capacity [3]. However, in the more realistic case of estimation with finite training data and capacity, a valid concern arises regarding the effect of an imperfect reconstruction model on the chain's equilibrium distribution. To this end, Alain et al. [1] adapts a result from perturbation theory [76] for finite state Markov chains to show that as the learned transition matrix becomes arbitrarily close to the target transition matrix, the equilibrium distribution also becomes arbitrarily close to the target joint distribution. For the discrete domains of interest here, we can enforce a finite state space by simply setting a maximum object size.

2.2.2 Sampling training sequences

Let $c(s \mid x)$ be a fixed conditional distribution over a sequence of corrupting operations $s = [s_1, s_2, ..., s_k]$ where k is a random variable representing the total number of steps and each $s_i \in \text{Ind}(\tilde{x}_i)$ where $\text{Ind}(\tilde{x}_i)$ is a set of legal inductive moves for a given $\tilde{x}_i$. The probability of arriving at corrupted sample $\tilde{x}$ from x is

$$c(\tilde{x} \mid x) = \sum_s c(\tilde{x}, s \mid x) = \sum_{s \in S(x,\tilde{x})} c(s \mid x), \tag{2.1}$$

where $S(x, \tilde{x})$ denotes the set of all corrupting sequences from x to $\tilde{x}$. Thus, the joint data-generating distribution is

$$p(x, s, \tilde{x}) = c(\tilde{x}, s \mid x)p(x) \tag{2.2}$$

where $c(\tilde{x}, s \mid x) = 0$ if $s \notin S(x, \tilde{x})$.

Given a corrupted sample, we aim to train a reconstruction distribution model $p_\theta(x \mid \tilde{x})$ to maximize the expected conditional probability of recovering the original, uncorrupted sample. Thus, we wish to find the parameters θ^* that mini-

mize the expected KL divergence between the true $p(x, s \mid \tilde{x})$ and learned $p_\theta(x, s \mid \tilde{x})$,

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \, \mathbb{E}_{p(x,s,\tilde{x})} \left[D_{\mathrm{KL}}(p(s, x \mid \tilde{x}) \parallel p_\theta(s, x \mid \tilde{x})) \right], \tag{2.3}$$

which amounts to maximum likelihood estimation of $p_\theta(s, x \mid \tilde{x})$ and likewise $p_\theta(x \mid \tilde{x})$. The above is an expectation over the joint data-generating distribution, $p(x, s, \tilde{x})$, which we can sample from by drawing a data sample and then conditionally drawing a corruption sequence:

$$x \sim p(x), \quad \tilde{x}, s \sim c(\tilde{x}, s \mid x). \tag{2.4}$$

2.2.3 Fixed corrupter

In general, we are afforded flexibility when selecting a corruption distribution, given certain conditions for ergodicity are met. We implement a simple fixed distribution over corrupting sequences approximately following these steps: 1) Sample a number of moves k from a geometric distribution. 2) For each move, sample a move type from {Insert, Delete}. 3) Sample from among the legal operations available for the given move type. We make minor adjustments to the weighting of available operations for specific domains. See Section 2.6.6 for full details.

The geometric distribution over corruption sequence length ensures exponentially decreasing support with edit distance, and likewise the support of the target reconstruction distribution is local to the conditioned corrupted object. The globally non-zero (yet exponentially decreasing) support of both the corruption and reconstruction distributions trivially satisfy the conditions required in Corollary A2 from Alain et al. [1] for the chain defined by the corresponding Gibbs sampler to be ergodic. Alternatively, one could employ conditional distributions with truncated support after some edit distance and still satisfy ergodicity conditions via the stronger Corollary A3 from Alain et al. [1].

Unless otherwise stated, the results reported in Sections 2.3 and 2.4, use a geometric distribution with five expected steps for the corruption sequence length. In general, we observe shorter corruption lengths lead to better samples, though we did not seek to specially optimize this hyperparameter for generation quality. See Section 2.6.1 for some results with other step lengths.

2.2.4 Reconstruction distribution

A sequence of corrupting operations $s = [s_1, s_2, ..., s_k]$ corresponds to a sequence of visited corrupted objects $[\tilde{x}_1, \tilde{x}_2, ..., \tilde{x}_k]$ after execution on an initial sample x. We enforce the corrupter to be Markov such that its distribution over the next corruption operation to perform depends only on the current object. Likewise, the target reconstruction distribution is then also Markov, and we factorize the learned reconstruction sequence model as the product of memoryless transitions culminating with a stop token:

$$p_\theta(s_{\text{rev}} \mid \tilde{x}) = p_\theta(\text{stop} \mid x) p_\theta(x \mid \tilde{x}_1) \prod_{i=1}^{k-1} p_\theta(\tilde{x}_i \mid \tilde{x}_{i+1}) \tag{2.5}$$

where $s_{\text{rev}} = [s_{k_{\text{rev}}}, s_{k-1_{\text{rev}}}, ..., s_{1_{\text{rev}}}, \text{stop}]$, the reverse of the corrupting operation sequence. If a stop token is sampled from the model, reconstruction ceases and the next corruption sequence begins. For the molecule model, an additional "revisit" stop criterion is also used: the reconstruction ceases when a molecule is revisited (see Section 2.6.4 for details).

For each individual step, the reconstruction model outputs a large conditional categorical distribution over $\text{Ind}(\tilde{x})$, the set of legal modification operations that can be performed on an input $\tilde{x}$. We describe the general architecture employed and include domain-specific details in Sections 2.3 and 2.4.

Any operation in $\mathrm{Ind}(\tilde{x})$ may be defined in a general sense by a *location* on the object $\tilde{x}$ where the operation is performed and a *vocabulary* element describing which vocabulary item (if any) is involved (Fig. 2.1). The prespecified vocabulary consists of domain-specific substructures, a subset of which may be legally inserted or deleted from a given object. The model induces a distribution over all legal operations (which may be described as a subset of the Cartesian product of the locations and vocabulary elements) by computing location embeddings for an object and comparing those to learned embeddings for each vocabulary element.

For the graph-structured domains explored here, location embeddings are generated using a message passing neural network structure similar to Duvenaud et al. [22], Gilmer et al. [29] (see Section 2.6.3). In parallel, the set of vocabulary elements is also given a learned embedding vector. The unnormalized log-probability for a given modification is then obtained by computing the dot product of the embedding of the location where the modification is performed and the embedding of the vocabulary element involved. For most objects from the molecule and Laman graph domains, this defines a distribution over a discrete set of operations with cardinality in the tens of thousands.

We note that although our model induces a distribution over a large discrete set, it does not do so through a traditional fully-connected softmax layer. Indeed, the action space of the model is heavily factorized, ensuring that the computation is efficient. The factorization is present at two levels: the actions are separated into broad categories (e.g., insert at atom, insert at bond, delete, for molecules), that do not interact except through the normalization. Additionally, actions are further factorized through a location component and vocabulary component, that only interact through a dot product, further simplifying the model.

2.3 Application: Molecules

Molecular structures can be defined by graphs where nodes represent individual atoms and edges represent bonds. In order for such graphs to be considered valid molecular structures by standard chemical informatics toolkits (e.g., RDKit [55]), certain constraints must be satisfied. For example, aromatic bonds can only exist within aromatic rings, and an atom can only engage in as many bonds as permitted by its valence. By restricting the corruption and reconstruction operations to a set of modifications that respect these rules, we ensure that the resulting Markov chain will only visit valid molecular graphs.

2.3.1 Legal operations

When altering one valid molecular graph into another, we restrict the set of possible modifications to the insertion and deletion of valid substructures. The vocabulary of substructures consists of non-ring bonds, simple rings, and bridged compounds (simple rings with more than two shared atoms) present in training data. This is the same type of vocabulary proposed in Jin et al. [44]. The legal insertion and deletion operations are set as follows:

Insertion For each atom and bond of a molecular graph, we determine the subset of the vocabulary that would be chemically compatible for attachment. Then, for each compatible vocabulary substructure, the possible assemblies of it with the atom or bond of interest are enumerated (keeping its already-connected neighbors fixed). For example, when inserting a ring from the vocabulary via one of its bonds, there is often more than one valid bond to select from. Here, we only specify the 2D configuration of the molecular graph and do not account for stereochemistry.

Deletion We define the leaves of a molecule to be those substructures that can be removed while the rest of the molecular graph remains connected. Here, the set of

leaves consists of either non-ring bonds, rings, or bridged compounds whose neighbors have a non-zero atom intersection. The set of possible deletions is fully specified by the set of leaf substructures. To perform a deletion, a leaf is selected and the atoms whose bonds are fully contained within the leaf node substructure are removed from the graph.

These two minimal operations provide enough support for the resulting Markov chain to be ergodic within the set of all valid molecular graphs constructible via the extracted vocabulary. As Jin et al. [44] find, although an arbitrary molecule may not be reachable, empirically the finite vocabulary provides broad coverage over organic molecules. Further details on the location and vocabulary representations for each possible operation are given in the appendix.

2.3.2 Data

For molecules we test the proposed approach on the ZINC dataset, which contains about 250K drug-like molecules from the ZINC database [85]. The model is trained on 220K molecules according to the same train/test split as in Jin et al. [44], Kusner et al. [52].

2.3.3 Distributional statistics

While predictive probabilities are not available from the implicit generative model, we can perform posterior predictive checks on various semantically relevant metrics to compare our model's learned distribution to the data distribution. Here, we leverage three commonly used quantities when assessing drug molecules: the *quantitative estimate of drug-likeness* (QED) score (between 0 and 1) [5], the *synthetic accessibility* (SA) score (between 1 and 10) [27], and the *log octanol-water partition coefficient* (logP) [15]. For QED, a higher value indicates a molecule is more likely to be drug-like, while for SA, a lower value indicates a molecule is more likely to be easily

synthesizable. logP measures the hydrophobicity of a molecule, with a higher value indicating more hydrophobic. Together these metrics take into account a wide array molecular features (ring count, charge, etc.), allowing for an aggregate comparison of distributional statistics.

Our goal is not to optimize these statistics but to evaluate the quality of our generative model by comparing the distribution that our model implies over these quantities to those in the original data. A good generative model would have novel molecules but those molecules would have similar aggregate statistics to real compounds. In Fig. 2.3, we display Gaussian kernel density estimates (KDE) of the above metrics for generated sets of molecules from seven baseline methods, in addition to our own (see Section 2.6.4 for chain sampling details). A normalized histogram of the ZINC training distribution is shown for visual comparison. For each method, we obtain 20K samples either by running pre-trained models [44, 30, 52], by accessing pre-sampled sets [61, 83, 57], or by training models from scratch [80].[2] Only novel molecules (those not appearing in the ZINC training set) are included in the metric computation, to avoid rewarding memorization of the training data. In addition, Table 2.1 displays bootstrapped Kolmogorov–Smirnov (KS) distances between the samples for each method and the ZINC training set.

Our method is capable of generating novel molecules that have statistics closely matched to the empirical QED and logP distributions. The SA distribution seems to be more challenging, although we still report lower mean KS distance than some recent methods. Because we allow the corrupter to uniformly select from the vocabulary, even if a particular vocabulary element occurs very rarely in training data, it can sometimes introduce molecules without an accessible synthetic route that the reconstructor does not immediately recover from. One could alter the corrupter and have it favor commonly appearing vocabulary items to mitigate this. We also note that

[2]We use the implementation provided by [8] for the SMILES LSTM [80].

our approach lends itself to Markov chain transitions reflecting known (or learned) chemical reactions.

Interestingly, the SMILES-based LSTM model [80] is effective at matching the ZINC dataset statistics, producing a substantially better-matched SA distribution than the other methods. However, as noted in [61], by operating on the linear SMILES representation, the LSTM has limited ability to incorporate structural constraints, e.g., enforcing the presence of a particular substructure.

In addition to the above metrics, we report a validity score (the percentage of samples that are chemically valid) for each method in Table 2.1. A sample is considered to be valid if it can be successfully parsed by RDKit [55]. The validity scores displayed are the self-reported values from each method. Our method, like Jin et al. [44], Liu et al. [61], enforces valid molecular samples, and the model does not have to learn these constraints. See Section 2.6.7 for additional evaluation using the GuacaMol distribution-learning benchmarks [8].

We might also inquire how the reconstructed samples of the Markov chain compare to the corrupted samples. See Fig. 2.6 in the appendix for a comparison. On average, we observe corrupted samples that are less druglike and less synthesizable than their reconstructed counterparts. In particular, the output reconstructed molecule has a 21% higher QED relative to the input corrupted molecule on average. Running the corrupter repeatedly (with no reconstruction) leads to samples that severely diverge from the data distribution.

2.4 Application: Laman Graphs

Geometric constraint graphs are widely employed in CAD, molecular modeling, and robotics. They consist of nodes that represent geometric primitives (e.g., points, lines) and edges that represent geometric constraints between primitives (e.g., specifying

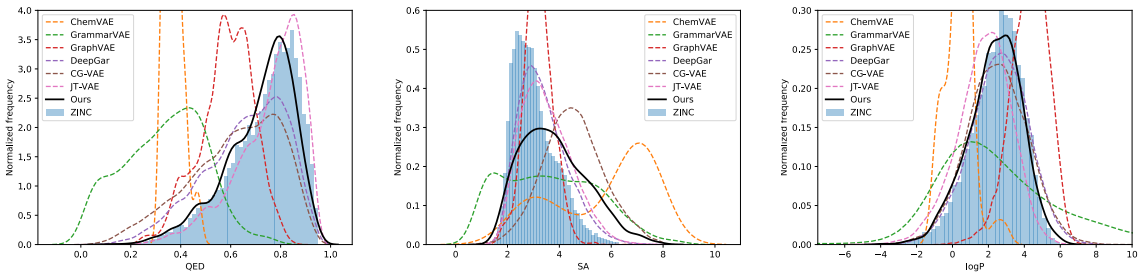


Figure 2.3: Distributions of QED (left), SA (middle), and logP (right) for sampled molecules and ZINC.

Source	QED KS	SA KS	logP KS	% valid
ChemVAE [30]	1.00 (0.00)	1.00 (0.00)	1.00 (0.00)	0.7
GrammarVAE [52]	0.94 (0.00)	0.95 (0.00)	0.95 (0.00)	7.2
GraphVAE [83]	0.52 (0.00)	0.23 (0.00)	0.54 (0.00)	13.5
DeepGAR [57]	0.20 (0.00)	0.15 (0.00)	0.062 (0.002)	89.2
SMILES LSTM [80]	0.022 (0.003)	0.051 (0.004)	0.052 (0.004)	96.1
JT-VAE [44]	0.090 (0.003)	0.21 (0.00)	0.20 (0.00)	100
CG-VAE [61]	0.27 (0.00)	0.56 (0.00)	0.064 (0.002)	100
GenRIC	0.045 (0.003)	0.28 (0.00)	0.057 (0.002)	100

Table 2.1: Molecular property distributional statistics. For each source, 20K molecules are sampled and compared to the ZINC dataset. For SA, QED, and logP, we compute the two-sample Kolmogorov-Smirnov statistic (and its bootstrapped standard error) compared to the ZINC dataset. (Lower is better for the KS statistic.) Self-reported validity percentages are also shown (the value for [30] is obtained from [52]).

perpendicularity between two lines). To allow for easy editing and change propagation, best practices in parametric CAD encourage keeping a part well-constrained at all stages of design [4]. A useful generative model over CAD models should ideally be restricted to sampling well-constrained geometry.

Laman graphs describe two-dimensional geometry where the primitives have two degrees of freedom and the edges restrict one degree of freedom (e.g., a system of rods and joints) [54]. Representing minimally rigid systems, Laman graphs have the property that if any single edge is removed, the system becomes under-constrained. For a graph with n nodes to be a valid Laman graph, the following two simple con-

Source	DoD KS	% valid
E-R [26]	0.95 (0.03)	0.08 (0.02)
GraphRNN [95]	0.96 (0.00)	0.15 (0.03)
GenRIC	0.33 (0.01)	100 (0.00)

Table 2.2: Laman graph distributional statistics. The mean and, in parentheses, the standard deviation, of the bootstrapped KS distance between the DoD distribution for each set of sampled graphs and the training data are shown. In addition, we display mean and standard deviations for bootstrapped validity scores.

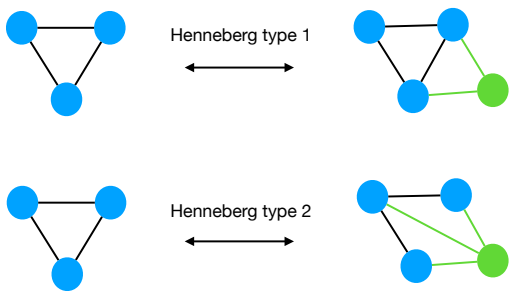


Figure 2.4: The legal inductive moves for Laman graphs, derived from Henneberg construction [38]. A single node (and accompanying edges) may be inserted or deleted in one of two ways.

ditions are necessary and sufficient: 1) the graph must have exactly $2n-3$ edges, and 2) each node-induced subgraph of k nodes can have no more than $2k-3$ edges. Together, these conditions ensure that all structural degrees of freedom are removed (given that the constraints are all independent), leaving one rotational and two translational degrees of freedom. In 3D, although the corresponding Laman conditions are no longer sufficient, they remain necessary for well-constrained geometry.

2.4.1 Legal operations

Henneberg [38] describes two types of node-insertion operations, known as *Henneberg moves*, that can be used to inductively construct any Laman graph (Fig. 2.4). We make these moves and their inverses (the delete versions) available to both the corrupter and reconstruction model. While moves #1 and #2 can always be reversed for any nodes of degree 2 and 3 respectively, a check has to be performed to determine where the missing edge can be inserted for reverse move #2 [35]. Here, we use the $O(n^2)$ Laman satisfaction check described in [41] to determine the set of legal neighbors. At the rigidity transition, it runs in only $O(n^{1.15})$.

2.4.2 Data

For Laman graphs, we generate synthetic graphs randomly via Algorithm 7 from Moussaoui [70], originally proposed for evaluating geometric constraint solvers embedded within CAD programs. This synthetic generator allows us to approximately control a produced graph's degree of decomposability (DoD), a metric which indicates to what extent a Laman graph is composed of well-constrained subgraphs. Such subsystems are encountered in various applications, e.g., they correspond to individual components in a CAD model or rigid substructures in a protein. The degree of decomposability is defined as $\text{DoD} = g/n$, where g is the number of well-constrained, node-induced subgraphs and n is the total number of nodes. We generate 100K graphs each for a low and high decomposability setting (see Section 2.6.5 for full details).

2.4.3 Distributional statistics

Table 2.2 displays statistics for Laman graphs generated by our model as well as by two baseline methods all trained on the low decomposability dataset (we observe similar results in the high decomposability setting). For each method, 20K graphs are sampled. The validity metric is defined the same as for molecules (Section 2.3.3). In addition, bootstrapped KS distance between the sampled graphs and training data for DoD distribution is shown for each method.

While it is unsurprising that the simple Erdős–Rényi model [26] fails to meet validity requirements ($< 0.1\%$ valid), we see that the recently proposed GraphRNN [95] fails to do much better. While deep graph generative models have proven to be very effective at reproducing a host of graph statistics, Laman graphs represent a particularly strict topological constraint, imposing necessary conditions on every subgraph. Today's flexible graph generative models, while effective at matching local statistics, are ill-equipped to handle this kind of global constraint. By leveraging domain-specific inductive moves, the proposed method does not have to learn what a

valid Laman graph is, and instead learns to match the distributional DoD statistics within the set of valid graphs.

2.5 Conclusion and Future Work

In this work we have proposed a new method for modeling distributions of discrete objects, which consists of training a model to undo a series of local corrupting operations. The key to this method is to build both the corruption and reconstruction steps with support for reversible inductive moves that preserve possibly-complicated validity constraints. Experimental evaluation demonstrates that this simple approach can effectively capture relevant distributional statistics over complex and highly structured domains, including molecules and Laman graphs, while always producing valid structures. One weakness of this approach, however, is that the inductive moves must be identified and specified for each new domain; one direction of future work is to learn these moves from data. In the case of molecules, restricting the Markov chain's transitions to learned chemical reactions could improve the synthesizability of generated samples. Future work can also explore enforcing additional hard constraints besides structural validity. For example, if a particular core structure or scaffold with some desired baseline functionality (e.g., benzodiazepines) should be included in a molecule, chain transitions can be masked to respect this. Coupled with other techniques such as virtual screening, conditional generation may enable efficient searching of candidate drug compounds.

2.6 Appendix

2.6.1 Geometric Distribution for Corrupter

Fig. 2.3 displays distributions for molecular samples from models trained with varying geometric distributions for the corruption sequence length. As the sequence length increases (and the corruptions become less local), the models produces worse samples. A short average corruption sequence length of one step seems to lead to a better-matched SA distribution, albeit with slower observed mixing for the Markov chain.

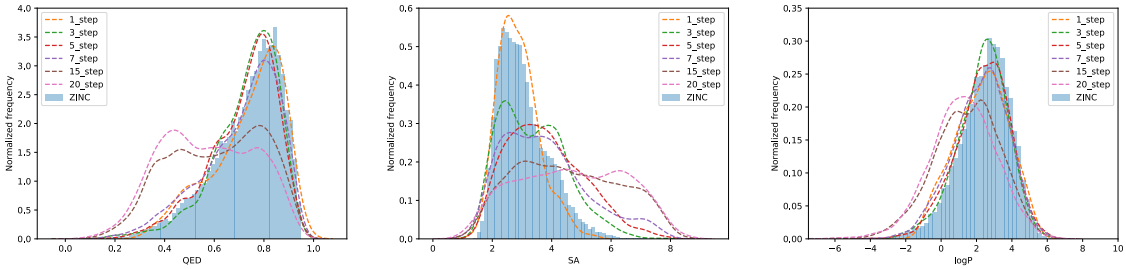


Figure 2.5: Distributions of QED (left), SA (middle), and logP (right) for the ZINC dataset and models trained with varying expected length corruption sequences.

2.6.2 Molecular Reconstruction Model Operations

We describe the representation assigned to each inductive operation. As described in Section 2.3, each modification is associated with a location (molecule dependent) and an operation type (molecule independent).

1. Stop: a global operation, naturally associated with the entire molecule. The location embedding is produced by embedding the entire molecule.

2. Delete atom leaf: a deletion operation where the deletion target is a single atom. The vocabulary is unique, and the location is associated with a single atom.

3. Delete ring leaf: a deletion operation where the deletion target is a ring or bridged compound. The vocabulary is unique, and the location is associated with a ring.

4. Insert via atom fusion: an insertion operation where the insertion is performed by attaching at an existing atom. The vocabulary is given by all atoms in each molecule of the vocabulary, and the location is associated with a single atom.

5. Insert via bond fusion: an insertion operation where the insertion is performed by attaching at an existing bond. The vocabulary is given by all bonds belonging to rings in each molecule of the vocabulary, and the location is associated with a single bond in a ring.

Embeddings for locations are computed in the following fashion. We follow a message passing architecture similar to Duvenaud et al. [22], Gilmer et al. [29], which produces a message for each bond and for each atom. The atom messages are transformed and pooled to produce the molecule embedding (used for `stop` prediction). Messages for each leaf atom are also transformed to produce embeddings for `delete leaf atom` actions. Messages for each bond in a leaf ring are transformed and pooled to produce embeddings for `delete leaf ring`. Messages for atoms and bonds are transformed to produce embeddings for `insert via atom fusion` and `insert via bond fusion`.

2.6.3 Training Details

In this section we give a brief description of the choices of parameters in training. We refer the reader to the source code[3] for a full description of the model architecture and parameters.

[3] `https://github.com/PrincetonLIPS/reversible-inductive-construction`

Molecule model

Molecules are converted into graphs in a manner identical to the representation used by [44]. The message passing model runs five steps of message passing. An embedding for the molecule is produced by transforming atom-level messages through a two-layer fully connected network, and aggregating the result through an average-pooling and a max-pooling operation (concatenated). For each task-relevant location, an embedding is produced by transforming and pooling the relevant messages, concatenating those with the molecule representation, and transforming with a two-layer fully-connected network. All messages and hidden layers have size 384.

We train each model for 50 epochs, with the Adamax optimizer and a base learning rate of 2×10^{-3} at batch size 128. The base learning rate is scaled linearly with the batch size. We also apply a learning rate schedule, dividing the learning rate by 10 after epochs 12, 24 and 36. Additionally, we apply learning rate warm-up by linearly scaling the learning rate from 0 to its base value during the first five epoch. The training is performed with a batch size of 1024, although we did not see any difference with smaller batch sizes (we did notice some issues with larger batches).

Laman model

Laman graphs are encoded for the message passing model with a single node degree feature. That feature is encoded with a Fourier encoding of the node degree. The message passing model runs five step of message passing. An embedding for the graph is produced by transforming the node messages with a two-layer fully-connected network, and aggregated using average and max pooling. Location-specific embeddings are produced in the same fashion as in the molecule model.

We train each model for 30 epochs, with the same optimizer settings as in the molecule model. We use a batch size of 256.

All models are trained using a Nvidia Titan X Pascal (12 GB) graphics card.

2.6.4 Sampling Details

Our proposed models require a Markov sampling step. We describe the details below.

For both the molecule and Laman models, we sample from the chain defined by the trained reconstructor by starting from a random object in the training dataset. The chain then alternatively samples sequences from the corrupter and the reconstructor. In both cases, the results reported in the main text use a corrupter that performs an average of 5 moves (with a geometric distribution).

As we sample from a Markov chain, we do not gather i.i.d. samples. In fact, sometimes the reconstructor returns to the same molecule on adjacent transitions due to perfect reconstruction. The results reported here use every sample from the Markov chain without thinning.

Although validity is maintained through the inductive moves, for both the molecule and Laman models, we in fact encode an action space slightly larger than the true set of valid inductive moves (to make the space more regular). When such an invalid action is sampled by the reconstructor, it is ignored, and another sample is taken. In some very rare instances, the reconstructor repeatedly samples invalid actions, in which case the entire transition (including the corruption) is resampled.

For both the molecule and Laman model, a minimal size is set (one leaf for molecules, and three nodes for Laman), to prevent the chain from deleting the entire object (which would cause problems in terms of the representation). For molecules, we also set a maximum size (in terms of the number of atoms), at 25 atoms, although we found values between 25 and 35 to have little effect on the results.

Revisit Stop Criterion

In the molecule setting, we make use of an additional stop criterion which is necessary as our model exhibits high precision and does not have access to any recurrent state

which would enable it to increase the probability of stopping as the length of the reconstruction sequence increases.

At each reconstruction step, we keep a history of all the molecules visited by the reconstructor so far, and stop the reconstruction process when the output of the reconstruction model already exists in its history.

We interpret this "revisit" as the model implicitly indicating that the obtained molecule is realistic enough (on average) that it is willing to return to it, despite not indicating stop itself due to the high precision of the model.

2.6.5 Dataset Details

Laman

As we did not find high-quality real-world datasets for Laman graphs, we considered some synthetic datasets generated by inductively sampling Henneberg moves in a random fashion. More explicitly, each graph in the dataset is generated using Algorithm 7 of [70], reproduced here as Algorithm 1, where the size n and the probability of selecting Henneberg type I moves p are sampled randomly. For all datasets, we sample n from a normal distribution with mean 30 and standard deviation 5. The distribution of p determines the distribution of the degree of decomposability of the graphs in the dataset. We choose the following distributions of p for each dataset: $p \sim \mathcal{U}(0, 0.1)$ for low decomposability and $p \sim \mathcal{U}(0.9, 1)$ for high decomposability.

2.6.6 Corrupter Details

Molecule Corrupter

We use a single fixed corrupter for all molecule models. To corrupt a molecule, we sample a number of corruption steps from a geometric distribution with the given mean, and iteratively apply Algorithm 2 to the molecule. We made no attempt to

Algorithm 1: Procedure for generating a Laman graph

input : n, the number of nodes in the graph. p, the probability of choosing a move of type I.
output: A Laman Graph.
Initialize from complete graph on three elements;
$G \leftarrow K_3$;
for $i \leftarrow 4$ **to** n **do**
    move $\leftarrow$ `Sample` (Bernoulli(p));
    if move $= 0$ **then**
        $G \leftarrow$ `ApplyRandomTypeI` (G) ;
    else
        $G \leftarrow$ `ApplyRandomTypeII` (G) ;
    end
end

Algorithm 2: Algorithm for single molecule corruption step

input : mol: molecule to corrupt
output: mol: corrupted molecule
if `uniform`() < 0.5 **then**
    mol $\leftarrow$ `DeleteRandomLeaf` (mol);
else
    atom $\leftarrow$ `GetRandomAtom` (mol);
    if `IsInRing` *(atom) and* `uniform`() < 0.25 **then**
        bond $\leftarrow$ `GetRandomBondAtAtom` (atom);
        mol $\leftarrow$ `InsertRandomAtBond` (mol, bond);
    else
        mol $\leftarrow$ `InsertRandomAtAtom` (mol, atom);
    end
end

optimize the corrupter to produce better samples from the generative model or ease the learning process.

Laman Corrupter

We use a single a single corrupter for all our Laman models. For a single corruption sub-step, this corrupter first chooses among the four action types (Henneberg type I and type II, and their inverses) uniformly at random, and then uniformly samples among the valid actions of the chosen type. As with the molecule corrupter, the number of sub-steps is sampled from a geometric distribution with given mean.

2.6.7 GuacaMol Benchmarks

We also evaluate our model on the new GuacaMol distribution-learning benchmarks [8] after training on the ChEMBL dataset [66]. Using the same hyperparameters as for the ZINC model, we obtain validity: 1.0, uniqueness: 0.933, novelty: 0.942, KL divergence: 0.771, and FCD: 0.058 (see [8] for a description of each metric and comparisons with a few other methods). Note that the FCD score [73] is not directly applicable to our model's samples due to inherent autocorrelation in the generated chains. In part, the FCD score assesses diversity of samples compared to the training set by computing the activation covariance of the penultimate layer of ChemNet. The autocorrelation limits the sample diversity but may be addressed by standard techniques for Markov chains such as thinning. Here, we report results using the same sampling framework as for the ZINC model.

2.6.8 Reconstructed vs. Corrupted Samples

In Fig. 2.6, we display QED and SA score distributions for the reconstructed molecules (x) and the corrupted molecules ($\tilde{x}$) visited during Gibbs sampling as well as molecules

generated by solely running the corrupter (with no reconstruction). The corruption-only samples severely diverge from the data distribution.

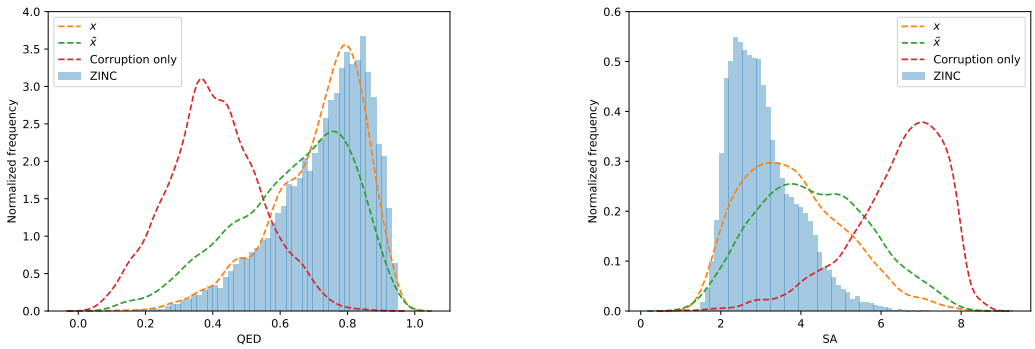


Figure 2.6: Distributions of QED (left) and SA (right) scores for reconstructed molecules (x) and corrupted molecules ($\tilde{x}$) visited during Gibbs sampling as well as molecules generated via corruption-only.

2.6.9 Example Chains

Below, we display three example chains for the molecular model.

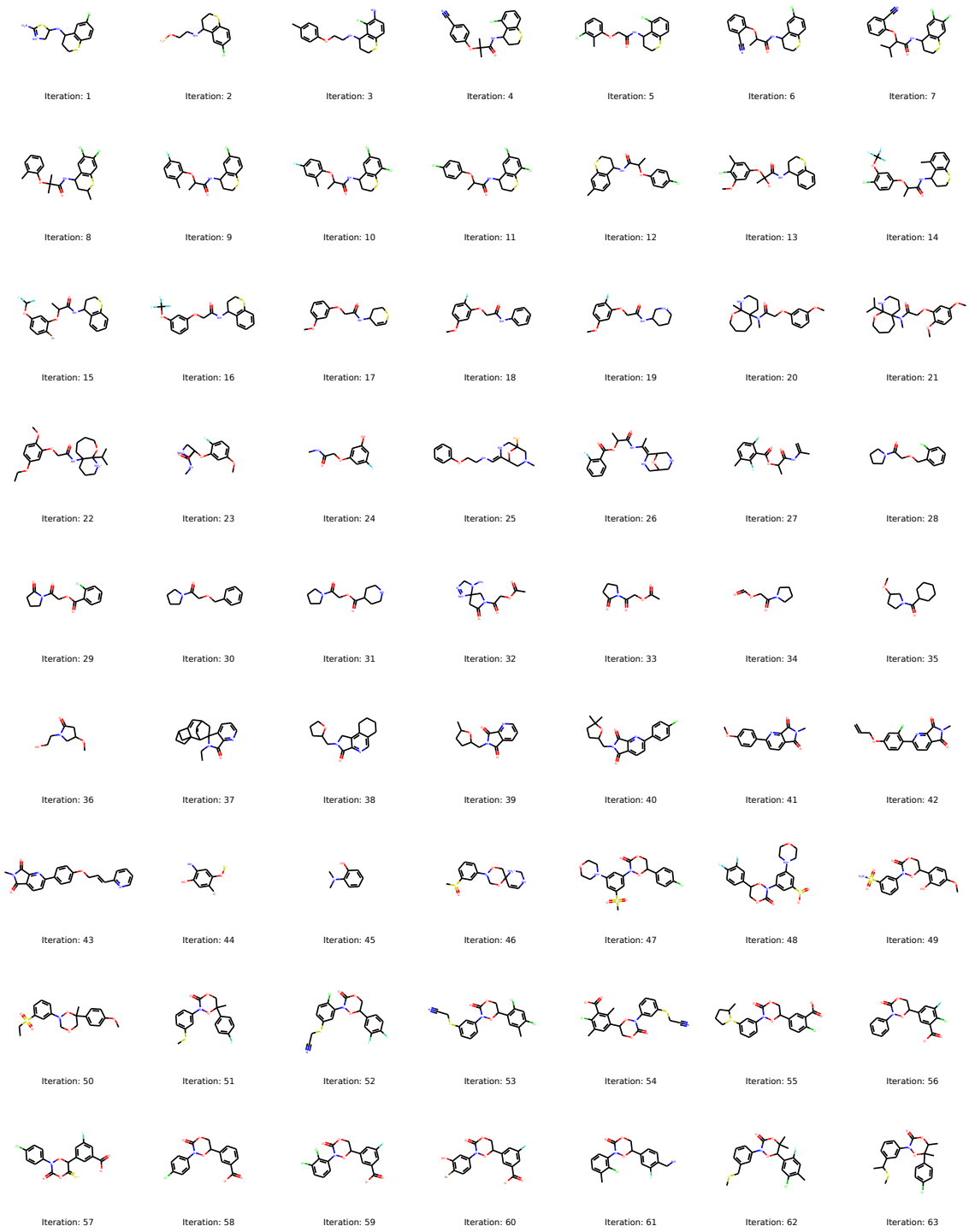


Figure 2.7: Example chain. The molecule is displayed after each transition of the Markov chain.

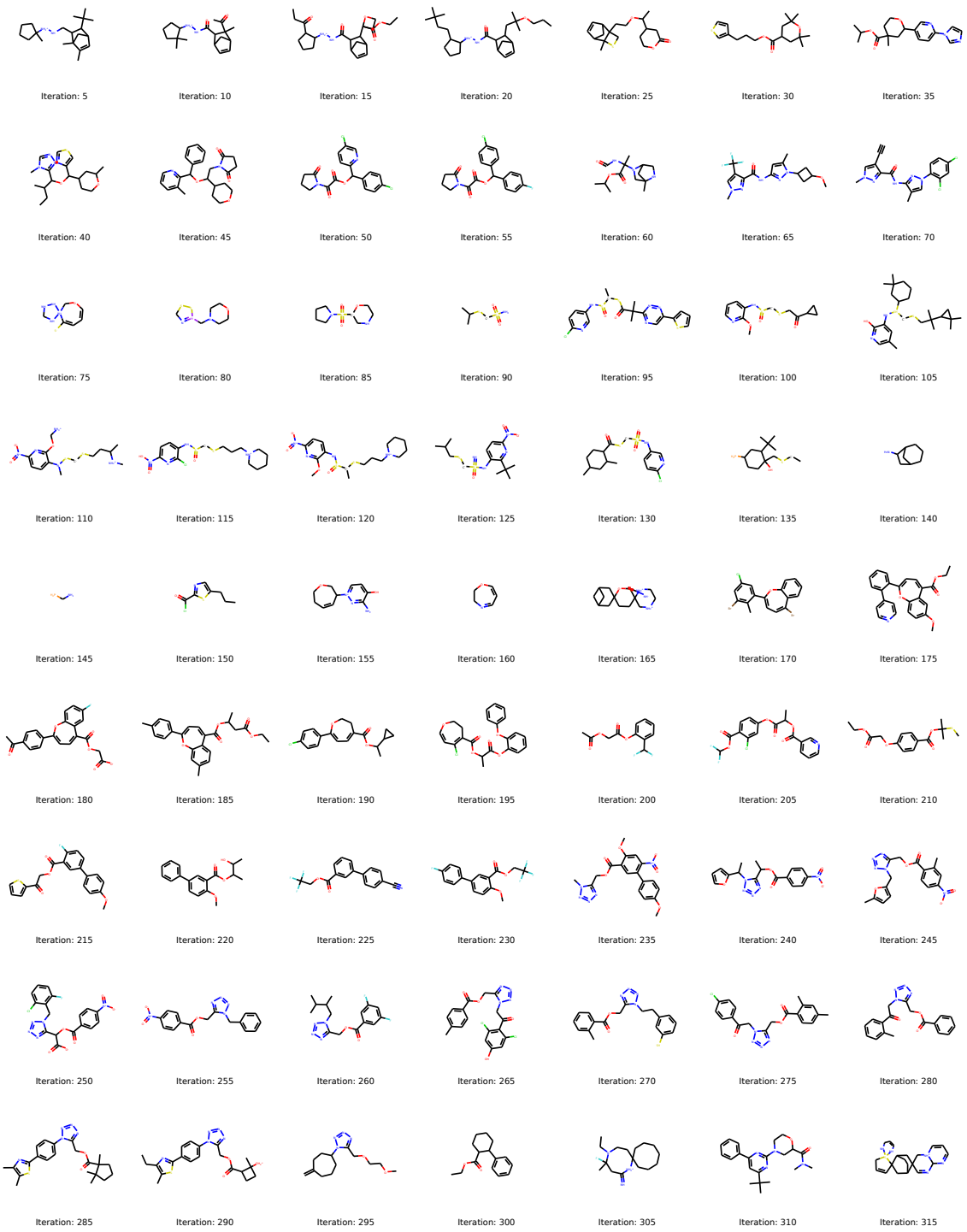


Figure 2.8: Example chain. The molecule is displayed every five transitions.

Figure 2.9: Example chain. The molecule is displayed every ten transitions.

Chapter 3

SketchGraphs: A Large-Scale Dataset for Modeling Relational Geometry in Computer-Aided Design

3.1 Introduction

The modern design paradigm for physical objects typically resembles modular programming, where simple subcomponents are connected to yield a composed part/assembly with more complex properties [4]. In parametric computer-aided design (CAD), parts generally begin as a collection of 2D sketches composed of geometric primitives (line segments, circles, etc.) with associated parameters (coordinates, radius, etc.). Primitives and parameters interact via imposed constraints (e.g., equality, symmetry, perpendicularity, coincidence) determining their final configuration. Edits made to any parameter will propagate along these specified dependencies, updating other properties of the sketch accordingly. A collection of 3D

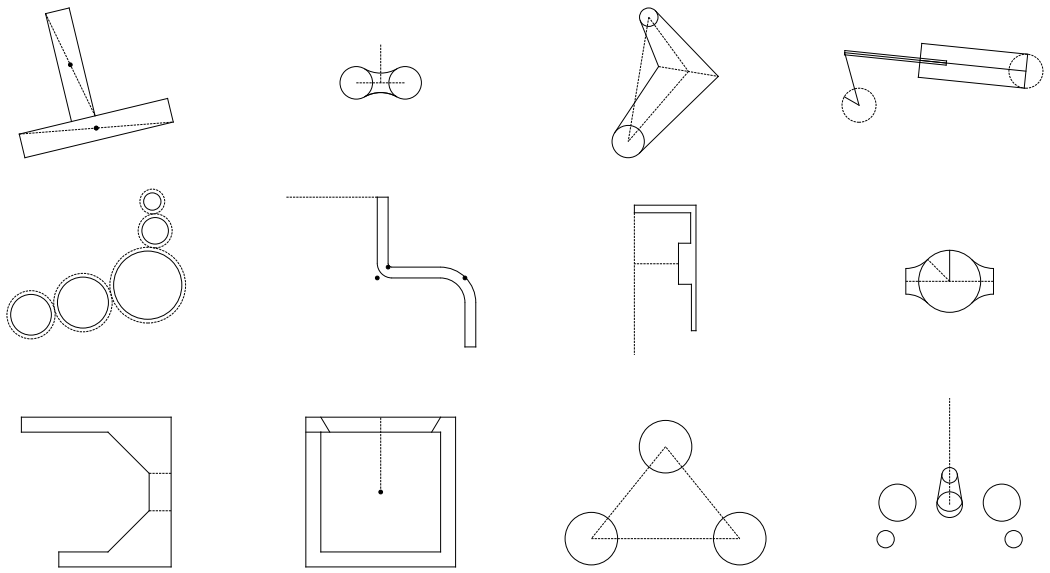


Figure 3.1: Example sketches from the dataset containing at least six geometric primitives. Dashed lines indicate *construction* geometry, which is used as a reference for other primitives but not physically realized.

operations, e.g., extruding a circle into a cylinder, then enable the creation of solids and surfaces from these 2D sketches.

Training machine learning models to construct and reason about object designs has the potential to enable new and more efficient design workflows. This is an important but challenging domain that sits at the interface of graphics, relational reasoning, and program synthesis. Recent progress in probabilistic generative modeling of both continuous (e.g., images and audio [48, 6, 87]) and discrete (e.g., graphs and source code [95, 57, 71]) domains has demonstrated the potential for both sampling high-dimensional objects and, in the case of explicit models, estimating their densities/probabilities. If adapted to CAD, such models may be incorporated into the design workflow by, for example, suggesting next steps based on partially specified geometry or offering corrections of implausible operations. In addition, if provided with visual observations of a part or sketch, a model may be trained to infer the underlying feature history, allowing for direct modification in CAD software.

Beyond the applicability to design itself, reasoning about human-designed structures is fundamental to artificial intelligence research. One of the challenges of computer vision, for example, is developing rich priors of objects in scenes. It is important to construct priors for the kind of "stuff" encountered in the world, which may exhibit significant symmetry and modularity, properties that are difficult to capture directly. By developing models for the design process and learning to reason about the creation of objects, it may become possible to identify hierarchical structures, long-range symmetries, and functional constraints that would be difficult to infer from vision alone. Outside of computer vision, a long-horizon goal of artificial intelligence is the task of program induction [58, 40, 19]. In program induction, the objective is to discover computer programs from examples of their inputs and outputs. Closely related to this are the ideas of programming by demonstration (e.g., Cypher and Halbert [16], Menon et al. [67]) and the problem of program synthesis (e.g., Ellis et al. [23], Gulwani et al. [33]). Part of the challenge of program induction is identifying a rewarding "sweet spot" in the coupled space of program representation and induction task; it is difficult to move beyond simple toy problems. We can view the design of physical objects in a parametric CAD system, however, as an example of constraint programming in which rich geometric structures are specified by an implicit program rather than, e.g., imperatively. It is a highly appealing domain for the study of program induction/synthesis because it is relatively well-scoped and low-order, but clearly requires the discovery of modularity to be effective. Moreover, progress on CAD program synthesis and induction leads to useful tools on a relatively short horizon.

To support these community research goals, we introduce SketchGraphs,[1] a dataset of 15 million sketches extracted from parametric CAD models hosted on Onshape,[2] a cloud-based CAD platform. Each sketch is represented with the ground truth geometric constraint graph specifying its construction, where edges denote

[1] `https://github.com/PrincetonLIPS/SketchGraphs`
[2] `https://www.onshape.com`

precise relationships imposed by the designer that must be preserved between specific primitives, the nodes of the graph (Fig. 3.2). Along with the dataset, we provide an open-source tool suite for data processing, removing obstacles to model development for other researchers.

Existing CAD datasets of voxel or mesh-based representations of 3D geometry [12, 93] have enabled work on sampling realistic 3D shapes [92, 62, 72]. Samples from such models, while impressive, are not modifiable in a parametric design setting and therefore are not directly usable in most engineering workflows. The recent ABC dataset [49] extracts parametric CAD models from Onshape's public platform, as do we, but is geared towards 3D modeling of curves and surfaces, supporting tasks such as patch segmentation and normal estimation. Explicit modeling of the relational structure exhibited by parametric CAD sketches, the target of SketchGraphs, is currently underexplored.

The SketchGraphs dataset may be used to train models directly for various target applications aiding the design workflow, including conditional completion (autocompleting partially specified geometry) and automatically applying natural constraints reflecting likely design intent (autoconstrain). In addition, by providing a set of rendering functions for sketches, we aim to enable work on CAD inference from images. Off-the-shelf noisy rendering options allow for sketches to appear hand-drawn. This setup is similar to that proposed in Ellis et al. [24], where TikZ figures are rendered with a hand-drawn appearance to provide training data for a model inferring LaTeX code from images. Here, our sketches are not synthetically produced but rather extracted from real-world parametric CAD models. See Section 3.4 for further details on the target applications supported by SketchGraphs.

This chapter makes the following contributions:

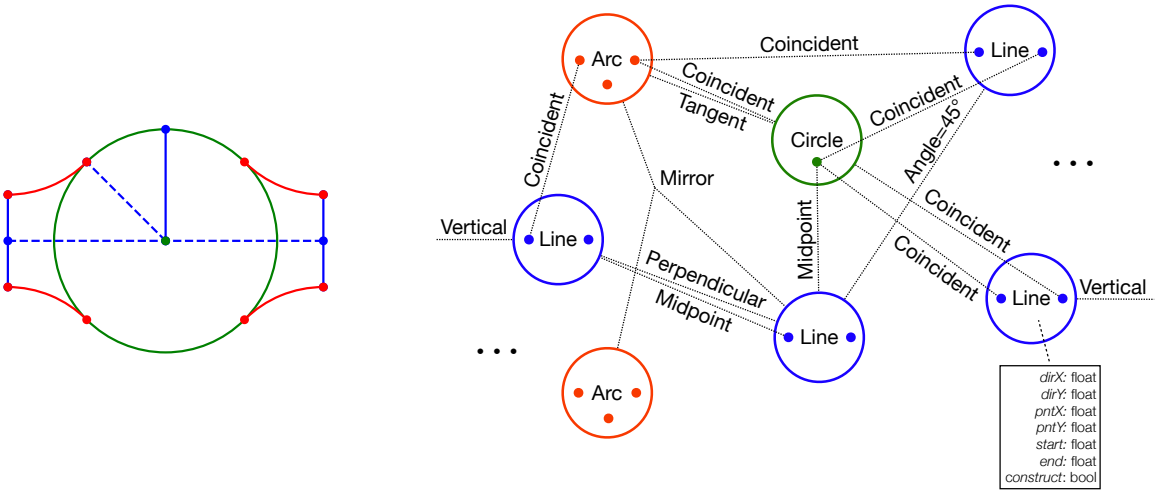


Figure 3.2: Example sketch (left) and a portion of its geometric constraint graph (right). Constraints are denoted as edges that either act on a primitive as a whole or some subcomponent of the primitive. Dots represent either a primitive's endpoints (left and right dots) or its center point (bottom dot).

- We collect a dataset of 15 million parametric CAD sketches including ground truth geometric constraint graphs denoting the primitives present and their imposed relationships.
- We develop an open-source pipeline for data processing, conversion, and integration with deep learning frameworks in Python. These technical contributions include:
  - Specific domain types to enable manipulation of the sketches and their representations in a structured manner.
  - A local renderer to enable visualization of sketches obtained from the dataset or generated by models.
- We establish baseline models for two initial use cases of the dataset: generative modeling of sketches and inferring likely constraints conditioned on unconstrained geometry.

3.2 Related Work

CAD datasets Existing large-scale CAD datasets have tended to focus on 3D shape modeling. ModelNet [93] and ShapeNet [12], for example, offer voxel and mesh-based representations of 3D geometry, enabling the generative modeling work of Wu et al. [92], Liu et al. [62], Nash et al. [72]. Note that these representations do not store the construction route employed by the original designer of each CAD model. The ABC dataset [49], in contrast to the above, contains parametric representations of 3D CAD models. Like SketchGraphs, the ABC models are obtained from public Onshape documents. However, the processing pipeline and benchmarks of the ABC dataset support 3D modeling of curves and surfaces. In contrast, we extract the 2D sketches that form the basis of 3D CAD models, enabling modeling of the geometric constraints used to ensure design intent.

Sketch datasets The word *sketch* is a term of art in the context of CAD, referring specifically to the 2D basis of 3D CAD models, storing both geometric primitives and imposed constraints. Large-scale datasets related instead to the colloquial usage of this term have been proposed in recent years, specifically focusing on hand-drawn sketches of general image categories (cat, bus, etc.). The QuickDraw dataset was constructed in Ha and Eck [34] from the *Quick, Draw!* online game [45] to train an RNN to produce vector image sketches as sequences of pen strokes. Pairings of pixel-based natural images and corresponding vector sketches are collected in the Sketchy dataset [74], intended to train models for sketch-based image retrieval. Like these datasets based on vector images, our SketchGraphs dataset is also fundamentally focused on the construction of sketches, not simply their raster appearance. However, here we focus on the relational geometry underlying parametric CAD sketches, not drawings of general categories.

Primitive type	%
Line	68.47
Circle	9.97
Arc	9.45
Point	8.58
Spline	2.57
Ellipse	0.08

Constraint type	%
Coincident	42.17
Projected	9.71
Distance	6.72
Horizontal	6.45
Mirror	5.54
Vertical	4.78
Parallel	4.37
Length	3.68
Perpendicular	3.24
Tangent	2.94

Table 3.1: Frequencies of the most common primitives (left) and constraints (right).

Graph-structured generative modeling Modern message passing networks (e.g., Gilmer et al. [29], Duvenaud et al. [22] extending the earlier work of Scarselli et al. [75]) have enabled progress in modeling of domains that exhibit relational structure, e.g., molecular chemistry and social networks. In the context of generative modeling, these networks are often coupled with node and edge-specific prediction modules that incrementally build a graph [57, 61]. We take a similar approach for two example applications demonstrated in Section 3.4. Several alternative architectures have been studied, such as LSTMs on linearized adjacency matrices [95] or decoding soft adjacency matrices with elements containing probabilities of edge existence [83]. In general, graph modeling is subject to significant representation ambiguity (up to $n!$ node orderings for a graph containing n nodes). Recent work leveraging normalizing flows [60] proposes a permutation-invariant approach for generating node features. We identify the CAD sketch domain as one which admits a natural ordering over construction operations for the underlying geometric constraint graphs, which we describe in Section 3.3.

Geometric program induction Geometric program induction comprises a practical subset of problems in program induction where the goal is to learn to infer a

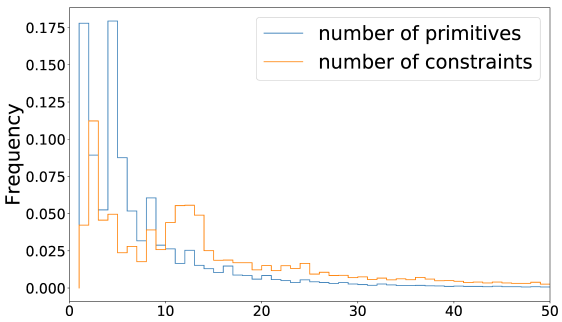
set of instructions to reconstruct input geometry. For example, Ellis et al. [24] couples a learned shape detector with program search to infer LaTeX code for synthetic images of TikZ figures and Sharma et al. [82] trains a reinforcement learning agent to reconstruct simple 3D shapes with constructive solid geometry. We view this as a fundamental area of study in order to develop machine learning models that can aid in design and engineering. Work on generating programs interactively, e.g., allowing a model to assess the current program's output [25], shows particular promise. Our processing pipeline includes functionality for sketch rendering and querying a geometric constraint solver to aid research in this direction.

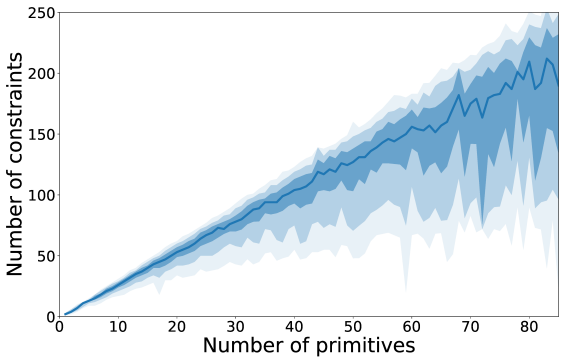
3.3 The SketchGraphs Dataset

SketchGraphs is aimed towards questions not just concerning the *what* but in particular the *how* of CAD design; that is, not simply what geometry is present but how was it constructed. To this end, we leverage a data source that provides some insight into the actual operations and commands selected by the designer at each stage of construction. Whereas generic CAD file formats are widely available online, e.g., STEP for 3D models and DXF for 2D drawings, these do not store any information regarding constraints. In recent years, the cloud-based CAD platform Onshape has amassed a large collection of publicly available models from which detailed construction histories may be queried. For each CAD sketch, we extract ground truth construction operations regarding both the geometric primitives present and constraints applied to them.

3.3.1 Acquisition

Using Onshape's API, we gather metadata for all public documents created within a five-year period from 2015 to 2020, leading to over two million unique document IDs.





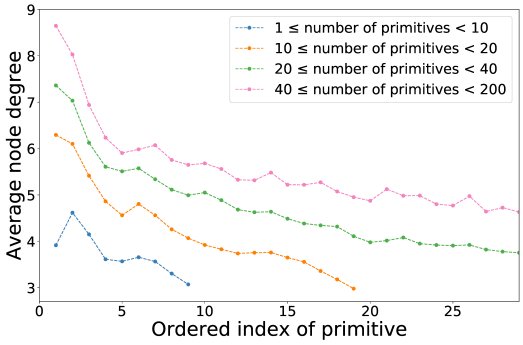


Figure 3.3: **Top-Left:** Histogram of sketch sizes as measured by the number of primitives and constraints they contain. **Top-Right:** Number of constraints as a function of number of primitives in the sketch (opacity represents respectively median, 69th, 84th and 93rd percentile). **Lower-Left:** Average node degree depicted as a function of sequence position. Note that primitives placed earlier in the sequence tend to serve as common anchors for subsequent primitives to constrain against.

Each document may contain multiple *PartStudios*, each specifying the design of an individual component of a CAD model. We download each PartStudio and extract all sketches present, resulting in over 15 million sketches. Note that the PartStudios also contain non-sketch features, e.g., 3D operations, that we do not store in our final dataset. Here, we focus only on the 2D sketches comprising each part and their underlying constraint graph representations.

To be included in the dataset, each sketch must contain at least one geometric primitive and one constraint. The dataset thus ranges from those sketches with larger constraint graphs, which tend to be more visually interesting, to some very simple sketches, e.g., sketches comprised of a single circle. See Section 3.3.1 and Table 3.1 for an overview of the sketch sizes and other dataset statistics.

3.3.2 Geometric constraint graphs

Geometric constraint graphs offer a succinct representation of 2D CAD sketches. For each sketch, we extract a graph $G = (V, E)$ with a set of nodes, V, and a set of edges, E, denoting primitives and constraints between them, respectively. In general, these are *multi-hypergraphs*, where multiple edges are permitted to share the same member nodes and each edge may join one or more nodes. When a constraint operates on a single primitive, e.g., a scale constraint such as setting the radius of a circle, we represent the constraint as a loop, an edge connecting the node of interest with itself. In addition, hyperedges indicate constraints that operate on three or more nodes. For example, a mirror constraint must specify a third primitive to act as an axis of symmetry.

Primitives and constraints are described not just by their type but also by parameters dictating their behavior. For primitives, parameters consist of the coordinates denoting their placement within a sketch and an *isConstruction* Boolean indicating if a primitive is to be physically realized (when false) or serve as a reference for other primitives (when true). Note that the initial values of a primitive's coordinates do not necessarily satisfy any of the constraints present; rather, the task of adjusting primitives' coordinates is left to a geometric constraint solver included in standard CAD software.

Constraint parameters indicate the primitive(s) acted upon as well as any other numerical or categorical values necessary to fully specify their behavior. For instance, a distance constraint includes a number indicating the Euclidean distance that two primitives must satisfy. Further details on primitive and constraint parameters may be found in the appendix.

Often, constraints are applied to a specific point on a primitive. For example, two endpoints from different line segments may be constrained to be a certain distance

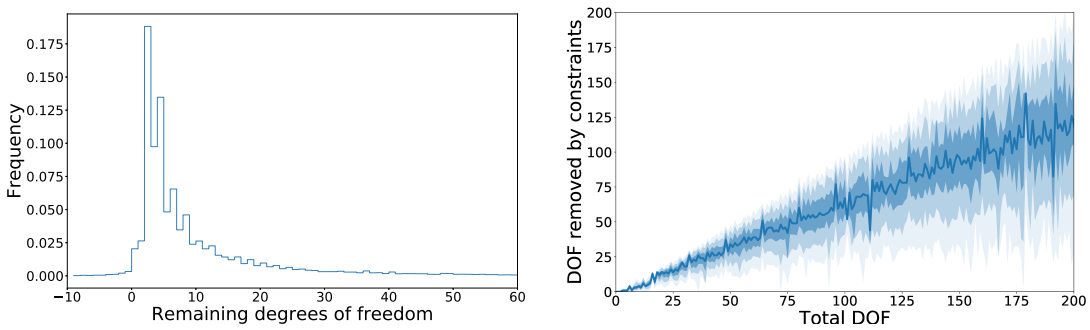


Figure 3.4: **Left:** Histogram of approximated degrees of freedom (DOF) remaining among sketches in the SketchGraphs dataset. **Right:** DOF removed by constraints in a sketch as a function of total DOF from its primitives before constraints are applied (opacity represents median, 69th, 84th, and 93rd percentiles).

apart. In order to unambiguously represent these constraints, we include these *sub-primitives* as separate nodes in the constraint graph.

Best practice in CAD encourages maintaining fully-constrained sketches, meaning a minimally sufficient set of constraints removes all degrees of freedom (DOF) from the sketch primitives [4]. This allows for edit propagation and better expression of design intent. Overall, we observe a Pearson correlation coefficient of 0.598 between the total DOF in each sketch and the total DOF removed by constraints[3] (Fig. 3.4). Users of SketchGraphs may query for sketches adhering to different thresholds of constrainedness depending on their application.

3.3.3 Construction sequence extraction

For certain problem settings, a sequence representation of the constraint graphs may be desired. In generative modeling, for example, graph-structured objects are often modeled autoregressively as sequences of construction steps [57, 95]. In the generic case, a canonical node ordering for graphs may not be available, leading to ambiguity regarding sequential modeling. Here, however, we have access to certain informa-

[3]We exclude constraints directly involving a sketch's axes in this calculation. Unfortunately, one limitation of our dataset is that these *external* constraints (constraints involving default geometry not defined by the user) are not currently retrievable via Onshape's API. This is consistent, however, with the common assumption that designs be fully-constrained up to rigid body transformation.

tion about each sketch's construction history that leads to natural sequence-based representations.

In particular, we may access the order in which primitives were added to a sketch by the user, conveying a ground truth node ordering (see Fig. 3.5 and Section 3.6.3 for examples). Rather than being an arbitrary choice among factorially many, we observe two trends that support this route: 1) Nodes with greater degree tend to occur earlier, serving as integral building blocks of the sketch (Section 3.3.1). 2) Adjacent nodes in the ordering have a greater probability of being adjacent in the graph than randomly selected nodes (0.70 vs. 0.38, respectively).

While the ordering of primitives and constraints are both known separately, the relative ordering (interleaving) of primitives and constraints is not recorded in the Onshape models. We canonicalize the entire sequence by placing an edge's insertion step immediately following the insertion of its member nodes. This emulates the standard design route of constraining primitives as they are added to a sketch. When there are ties, such as when multiple nodes share more than one edge, or multiple edges share the most recently added node, we may simply revert to the standalone edge ordering.

An alternative sequence option stems from the setting where unconstrained geometry is imported into CAD software, for instance from a drawing scan, and the software attempts to apply intended constraints. In this case, the corresponding sequence places all constraints at the end, after specifying all primitives. We note that there will likely be additional sequence extraction methods of interest to users of the dataset. Our pipeline may be easily extended to handle custom conversions.

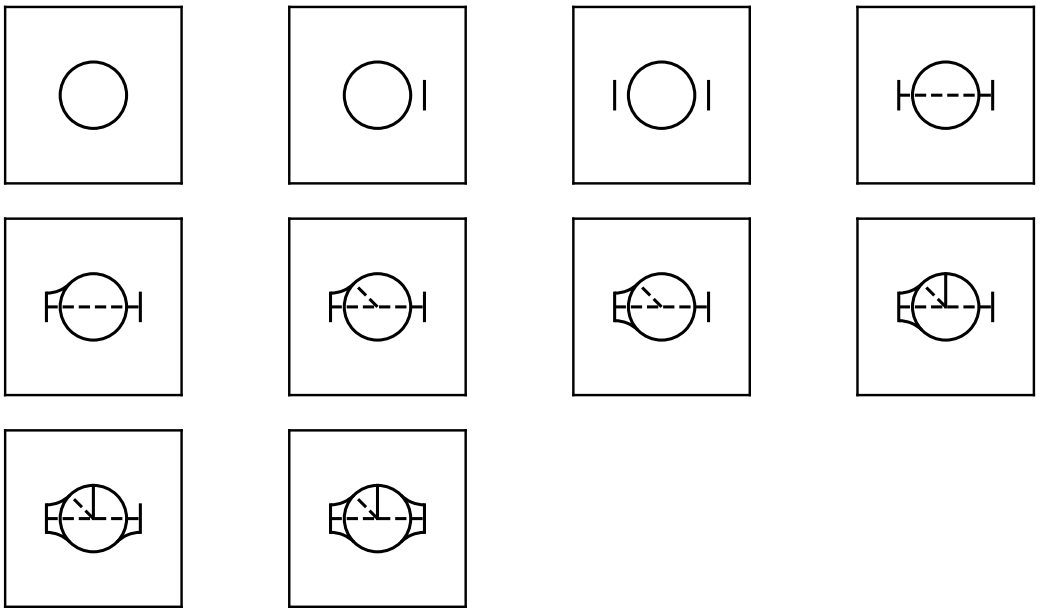


Figure 3.5: Construction of a dataset sketch proceeding from top left to bottom right.

3.4 Case Studies of Supported Applications

We identify several target applications for which SketchGraphs data may be used to train and develop models and describe some initial methods to tackle these applications. In addition to mechanical design-focused applications, a domain underexplored in the machine learning community, we note that these problems share properties with similar tasks in program synthesis and induction. We intend for SketchGraphs to serve as a test bed for these related lines of work and for the models below to provide baselines for future research.

3.4.1 Autoconstrain

CAD packages such as Onshape, AutoCAD, and Solidworks typically contain built-in constraint inferencing. However, these functions are based on manually-defined heuristics, catered towards interactive sketching (for example placing a coincidence constraint when a user drags a new primitive from an existing one). A sought-after

feature is the ability to upload unconstrained geometry, such as from a drawing or document scan, and infer the design intent and corresponding set of constraints. By treating the primitives in the dataset's sketches as input, the ground truth constraints may serve as a predictive target. This may be viewed as an instance of program induction in constraint programming.

The autoconstrain task, then, is to predict a set of constraints given an input configuration of geometric primitives. Here, we are particularly interested in predicting the "natural" set of constraints that would have been input by a human user. However, we note that other target constraint sets may be considered, by e.g., requiring them to be minimal in some mathematical sense [97, 96, 56]. The autoconstrain problem can also be viewed as an example of a *link prediction* task in which the induced relationships are the constraints; see e.g., Taskar et al. [86], Liben-Nowell and Kleinberg [59], Lü and Zhou [64].

Model

We propose an autoregressive model based on message passing networks (MPNs) [22, 29] where information about the input geometric primitives is propagated along a growing set of constraints, iteratively building a constraint graph. The model is tasked with predicting the sequence of edges corresponding to the given node sequence, and proceeds in a recurrent fashion by iteratively predicting the next edge (represented as a pair of the edge partner node and the edge label) for each node (or a `stop` token indicating to move to the next node).

Edge prediction is trained in a supervised fashion simultaneously via two related tasks:

- (partner prediction) Given the primitives (nodes) in the sketch, and the graph representing the constraints at the given step of the construction sequence, predict which node should be attached to the current node in order to create

a new constraint (or choose to move to the next node by predicting a sentinel node to attach).

- (constraint label prediction) Given the above, and the target partner of the current constraint, predict the type of the current constraint.

At inference time, the model is additionally given a mask indicating (approximately) which constraints are satisfied in the sketch (these may be satisfied because they were originally imposed in the dataset, or they may be a consequence of the original constraints). This ensures that the model only selects valid constraints and does not deform the sketch under consideration.

The model may conceptually be divided into three components: 1) an input representation component, responsible for embedding the features from the primitives and constraints, 2) a message-passing component, responsible for transforming these features using the graph structure, and 3) a readout component, which outputs probabilities for the specific tasks according to the transformed features. We describe each one in turn.

Input representation Constraints are only identified by their referenced primitives and their type. At the current stage, we embed their type using an m-dimensional[4] embedding layer. Primitives are a sequence of heterogenous discrete elements, and thus require a little more care. The type of the primitive (i.e. Point, Line etc.) is embedded similarly using an m-dimensional embedding layer. However, each primitive type may have a different number of parameters, which must represented similarly. Continuous parameters are quantized and represented by their quantized value, which is embedded using an m-dimensional layer. For a given primitive, all of its parameters are embedded and averaged to form a parameter embedding

[4]Our model is parameterized by a global complexity parameter m (we use $m = 384$ in the results we present).

of size m (if a primitive has no parameters, the parameter embedding is set to zero). The embedding for the primitive is then computed by concatenating the embedding corresponding to the type, and that of the parameters, and projecting this $2m$ vector onto one of size m through a dense layer.

Message passing Prior to message passing, the node embeddings are transformed with a 3-layer recurrent neural network employing the GRU architecture [13] (and with the hidden size set to m). The node embeddings are then recursively transformed using a message passing network, such that at stage s, we perform the update:

$$\mathbf{a}_v^{(s+1)} = \sum_{u:(u,v)\in E} f_e(\mathbf{m}_u^{(s)}, \mathbf{c}_{(u,v)}),$$
$$\mathbf{m}_v^{(s+1)} = f_n(\mathbf{a}_v^{(s+1)}, \mathbf{m}_v^{(s)}),$$

where here $\mathbf{c}_{(u,v)}$ denotes the representation for the constraint computed previously, and $\mathbf{m}_u^{(0)}$ is the representation for the primitive computed previously. We set f_e as a linear layer that concatenates $\mathbf{m}_u^{(s)}$ and $\mathbf{c}_{(u,v)}$ and set f_n to take the functional form of a GRU cell. We use 3 message passing steps in our presented results.

A global representation for the graph is also obtained by computing a weighted sum of the final node messages, where the weights are computed using a learned function comprised of a sigmoid applied to a linear layer. This representation is combined with the final state of the GRU using a linear layer again to obtain a final global representation for the problem.

Readout The readout for predicting the partner takes as input the final representations for each node in the graph, the final representation for the current node, and the global representation. These representations are concatenated and fed to a fully-connected two-layer network (with ReLU non-linearity) which produces a scalar value for each node in the graph. This value is interpreted as an unnormalized log-

probability for the partner selected being the given node, where an implicit 0 value is assigned to a sentinel node representing a request to move to the next node.

The label prediction readout takes as input the final representations for both the current node and partner node, as well as the global representation. These representations are concatenated and input to a three-layer fully-connected network. The output of that network is then interpreted as unnormalized log-probabilities for predicted constraint type.

Training

The model is trained on a subset of SketchGraphs consisting of about 2.2M sketches, limited to the most common types of primitives (Point, Line, Circle, and Arc) and at most 16 primitive primitives per sketch. We exclude hypergraphs from consideration here and only model two-node and single-node edges. During training, the model is presented with random edges (or "stop" edges representing a request to move to the next node), selected uniformly among all possible edges in the graphs (this implies that the training is weighted towards larger graphs). We use the Adam optimizer [47] with a batch size of 8192 and a learning rate of 10^{-5}, where the learning rate is understood to apply to the loss as expressed as a sum over the batch (rather than an average). The training is performed over 150 epochs of the data, with the learning rate decaying by a factor of 10 at epochs 50 and 150. The training is performed on a server equipped with 4 Nvidia Titan X (Pascal) and dual Intel Xeon E5-2667 v4 (total 32 logical cores) and takes 3 hours and 30 minutes.

Evaluation

We evaluate the model on a separate held-out set of 50K sketches. During the evaluation stage, invalid edge predictions are masked from the model. Although the model

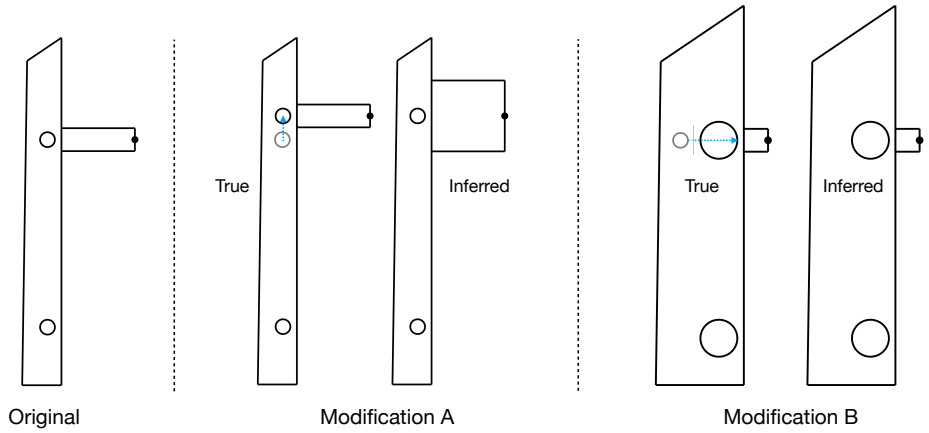


Figure 3.6: Autoconstraining a sketch. On the left is the original input sketch (only primitives are provided to the model). Two user modifications are shown with the blue arrows: dragging the top circle upwards (modification A) and both enlarging and dragging it to the right (modification B). Our model correctly picks up on a host of coincident, distance, equality, and other constraints. It fails to infer a 7 mm distance constraint between the top and bottom of the rectangle, but the circle correctly maintains midpoint alignment with it (modification A).

naturally operates in a factorized fashion:

$$P(\text{edge} \mid X) = P(\text{edge label} \mid \text{edge partner}, X)P(\text{edge partner} \mid X)$$

we note that the mask itself does not factorize in the given fashion (as the validity of a partner may depend on the specific constraint being considered). We thus reconstruct the full joint distribution over the candidate partners and labels according to the given conditionals, and mask and re-scale the predictions using the joint distribution directly.

We obtain an average edge recall of $0.74(\pm 0.00)$, an average edge precision of $0.74(\pm 0.00)$, and an average F1 score of $0.71(\pm 0.00)$ (where the brackets indicate the standard error of the estimate). We also evaluate the test negative log-likelihood at an average of 0.495 bits per edge. For reference, a uniform choice among valid constraints scores an average entropy of 6.09 bits per edge.

We also demonstrate the inferred constraints qualitatively by editing a test sketch and observing the resulting solved state Fig. 3.6.

3.4.2 Generative modeling

A variety of target tasks may be approached under the broader umbrella of generative modeling. By learning to predict sequences of sketch construction operations, for example, models may be employed for conditional completion, interactively suggesting next steps to a CAD user. In addition, *explicit* generative models, estimating probabilities (or densities) of examples, may be used to assess the overall plausibility of a sketch via its graph or construction sequence, offering corrections of dubious operations (similar to "autocorrect"). This also provides a path to a CAD analog of inductive programming support that has been deployed in Microsoft Excel [32].

Here we develop an initial model and benchmark for unconditional generative modeling. We train a model on full construction sequences for the sketch graphs, both nodes and edges. While we model constraint parameters (edge features), we only model the primitive type parameter for the nodes, leaving the task of determining the final configuration of primitive coordinates to a constraint solver.

Model

The model resembles that from the autoconstrain task above, with an additional node-adding module. When edge sampling for a given node has completed, the model outputs a distribution over possible primitive types to add to the graph. After a node's insertion, any sub-primitive nodes associated with the new node (e.g., endpoints of a line segment) are deterministically added to the graph with corresponding edges denoting the sub-primitive relationship. Alternatively, a `stop` token may be selected that ceases graph construction. We highlight the major differences from the autoconstrain model below.

Input representation The input representation is similar to that of the autoconstrain model, with the exception that primitive parameters are ignored whereas constraint parameters are represented.

Message passing The message passing process is similar to that of the autoconstrain model, except no recurrent model is used, and the node embeddings are used directly. In contrast to the autoconstrain problem, all nodes present in the generative problem are included in the graph, and hence we do not require a preprocessing step for the node embeddings.

Readout In addition to the readout networks presented for the autoconstrain model, readout models are present for the primitive prediction task and predicting constraint parameters. The primitive prediction readout is given the global embedding for the graph and is tasked to predict the type of the next primitive to be added to the construction sequence. It is implemented as a 3-layer fully-connected network. The constraint parameter readout is given the constraint type as well as a representation computed from the primitives participating in the constraint. The parameters are then read-out sequentially using a recurrent neural network.

Training

The training set is the same as that for the autoconstrain model. Numerical constraint parameters adhering to the most frequent schemas (Section 3.6.2) are included as targets. The model is trained using the Adam optimizer, with a batch size of 6144 and a learning rate of 10^{-5}, where the learning rate is understood to apply to the loss as expressed as a sum over the batch. The training is performed for 150 epochs of the data, with the learning rate decaying by a factor of 10 at epochs 50 and 150. The training is performed on a server equipped with 4 Nvidia Titan X (Pascal) and

dual Intel Xeon E5-3667 v4 (total 32 logical cores), although only 3 GPUs were used due to a CPU bottleneck. The training takes 3 hours and 50 minutes.

Evaluation

The evaluation of high-dimensional generative models is an open problem. Here we provide quantitative evaluation consisting of likelihood on the held-out test set as well as distributional statistics. Using the same train-test split as in Section 3.4.1, the average negative log-likelihood of test examples according to the trained model is 28.2 bits per sketch. In comparison, a standard LZMA compressor, applied to a short canonical representation of the data (explained below), yields an average entropy of 85.6 bits per sketch.

LZMA comparison To estimate the performance of the LZMA compressor on the data, we represent the construction sequence as a sequence of integers (representing all labels and features). Such sequences are then concatenated and compressed using Python's implementation of LZMA with preset 9 and `LZMA_EXTREME`. Letting s_n denote the compressed size (in bits) of the the first n elements of the dataset, we report an estimate of the average entropy rate by computing $(s_{100000} - s_{50000})/50000$.

Distributional statistics We compare a host of statistics for the ground-truth training dataset to statistics on 10K generated samples. Fig. 3.8 depicts sketch-size distributions in terms of primitive and constraint counts, and Fig. 3.9 depicts the distribution of degrees of freedom in each set of sketches. Error bars in these histograms represent 5th and 95th percentiles acquired by resampling the generated sketches with replacement 2K times. We also compare the distribution of primitive and constraint types in Fig. 3.10.

For depictions of generated sketches, see Fig. 3.7. While this baseline model produces primitive types, constraint types, and constraint parameters, it is not trained

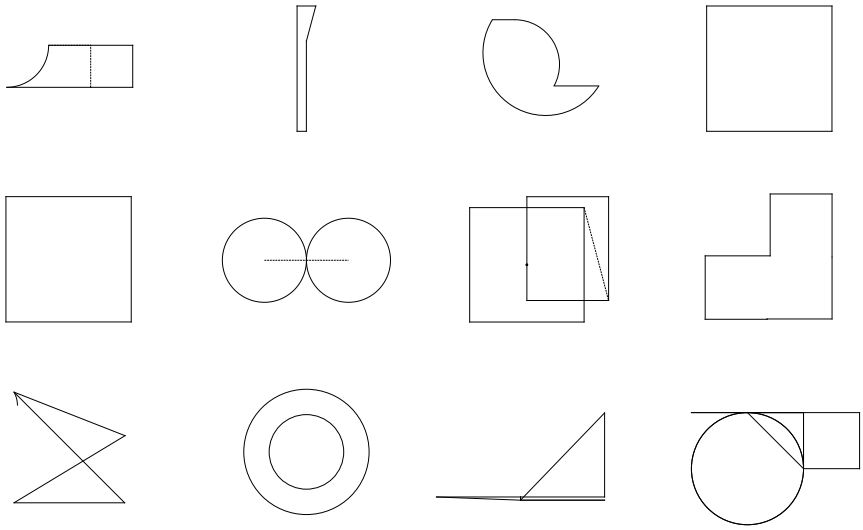


Figure 3.7: Random samples from the trained generative model containing at least two primitives. A solver is used to determine the final configuration of the sketch after the model samples the geometric constraint graph. Each primitive is initialized uniformly (e.g., all lines initially lie on the x-axis from 0 to 1) and their coordinates are updated by the solver. This baseline model, which does not output primitive parameters, is able to capture some of the patterns observed for small sketches but does not produce many sophisticated sketches.

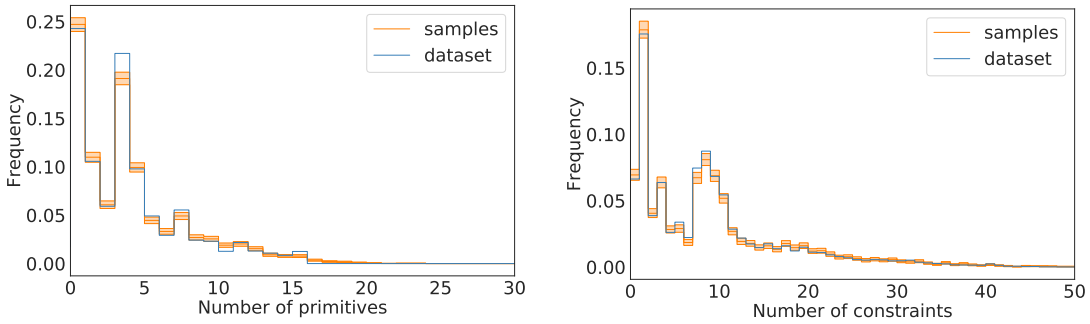


Figure 3.8: Distributions of sampled and training set sketch sizes. Error bars represent bootstrapped 5th and 95th percentiles.

to initialize the primitive coordinates. The solver determines the final configuration starting from a uniform initialization, which limits the visual diversity of the observed samples. Future work will include modeling the primitive coordinates.

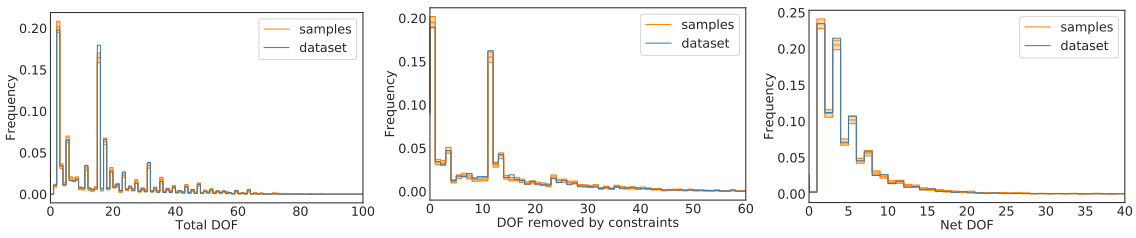


Figure 3.9: Degrees of freedom statistics for sampled and training set sketches. Error bars represent bootstrapped 5th and 95th percentiles.

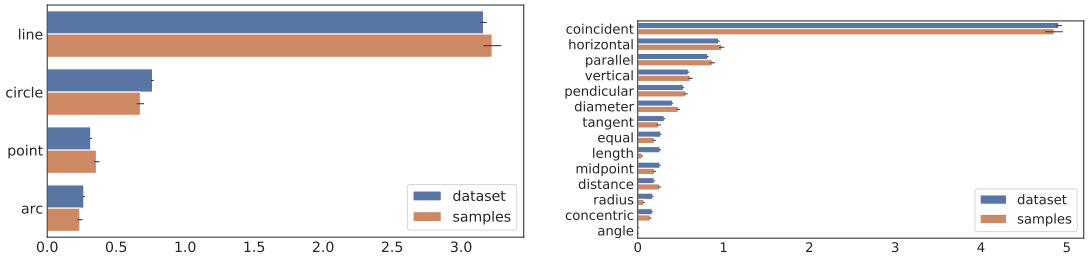


Figure 3.10: Average number of primitives and constraints of each type per sketch in training set and sampled sketches.

3.4.3 Other potential applications

CAD inference from images

A highly-sought feature for CAD software is the ability to input a noisy observation of an object (2D drawing, 3D scan, etc.) and infer its design steps, producing a plausible parametric CAD model. Inferring sketch primitives and constraints, which form the 2D basis of 3D CAD models, is an integral component of this application. Our pipeline includes rendering functions for producing images of the sketches, including noisy rendering to simulate a hand-drawn appearance. This allows generating millions of training pairs of rendered sketches and corresponding geometric constraint graphs.

Learning semantic representations

New models trained on the SketchGraphs data can lead to vectorial latent representations that capture important semantic content in sketches. Such vector representations have been developed for natural language processing [2, 68], speech recognition [37], computer vision [46], and computational chemistry [31]. These representations

have opened up a space of creative new possibilities for downstream tasks from search to content recommendation. Unsupervised learning on the SketchGraphs data enables such possibilities for CAD designs.

3.5 Conclusion and Future Work

This chapter has introduced SketchGraphs, a large-scale dataset of parametric CAD sketches and processing pipeline intended to facilitate research in ML-aided design and broader problems in relational reasoning and program induction. Each sketch is accompanied by the ground truth geometric constraint graph denoting its configuration. We demonstrate two use cases of the dataset, unconditional generative modeling and conditionally inferring constraints given primitives, providing initial benchmarks for these applications.

While we focus on 2D sketches here, which serve as the anchors for full parametric CAD models, future work will aim to make the complete set of construction operations accessible to modeling. We will also be providing benchmarks for additional applications supported by the dataset, including parametric CAD inference from images, a potentially powerful design aid.

3.6 Appendix

3.6.1 Primitive Parameters

Primitives are accompanied by both numerical and Boolean parameters specifying their initial positions within a sketch. For certain primitives, Onshape and other CAD programs employ an overparameterized description that aids in constraint solving. In our pipeline, the classes representing each primitive type contain attributes

corresponding to Onshape's parameterization but include methods for conversion to standard parameterizations.

As mentioned in the main text, all primitives have an *isConstruction* Boolean parameter indicating if a primitive is to be physically realized or simply serve as a reference for other primitives. We provide the remaining parameterization for common primitive types below.

Point (dof: 2)

- `x` (*float*): x coordinate
- `y` (*float*): y coordinate

Line (dof: 4)

- `dirX` (*float*): x component of unit direction vector
- `dirY` (*float*): y component of unit direction vector
- `pntX` (*float*): x coordinate of any point on the line
- `pntY` (*float*): y coordinate of the same point as above
- `startParam` (*float*): signed distance of starting point relative to (`pntX`, `pntY`)
- `endParam` (*float*): signed distance of ending point relative to (`pntX`, `pntY`)

Circle (dof: 3)

- `xCenter` (*float*): x coordinate of circle center
- `yCenter` (*float*): y coordinate of circle center
- `xDir` (*float*): x component of unit direction vector[5]
- `yDir` (*float*): y component of unit direction vector

[5]Circles are considered to have an angular direction in order to account for rotation of sketch components involving circles.

- `radius` (*float*): radius
- `clockwise` (*bool*): orientation of the unit direction vector

Arc (dof: 5)

- `xCenter` (*float*): x coordinate of corresponding circle's center
- `yCenter` (*float*): y coordinate of corresponding circle's center
- `xDir` (*float*): x component of unit direction vector
- `yDir` (*float*): y component of unit direction vector
- `radius` (*float*): radius of corresponding circle
- `clockwise` (*bool*): orientation of the unit direction vector
- `startParam` (*float*): starting angle relative to unit direction vector
- `endParam` (*float*): ending angle relative to unit direction vector

Ellipse (dof: 5)

- `xCenter` (*float*): x coordinate of ellipse's center
- `yCenter` (*float*): y coordinate of ellipse's center
- `xDir` (*float*): x component of unit direction vector
- `yDir` (*float*): y component of unit direction vector
- `radius` (*float*): greater (major) radius
- `minorRadius` (*float*): smaller radius
- `clockwise` (*bool*): orientation of the unit direction vector

3.6.2 Constraint Parameters

All constraints act on at least one primitive, indicated by the corresponding edge's member nodes. A subset of constraints require additional numerical, enumerated, or Boolean parameters to fully specify their behavior. Here we list the general parameters that may accompany constraints followed by the schemata for common constraint types. We exclude any external constraints (e.g., constraints involving projected geometry) and describe only those that act on user-defined geometry within a sketch. Numerical parameters follow user-specified units.

Note that constraint parameters are considered internal to Onshape and thus external documentation is sparse. We determine parameter functionality based on discussions with Onshape developers and usage of the solver.

Parameters

- `local#` (*reference*): a reference to a primitive. A constraint may have one or more of these (e.g., `local0`, `local1`, ...). The alternative parameter names `localFirst` and `localSecond` are used interchangeably in the data with `local0` and `local1`, respectively.
- `length` (*float*): quantity for a numerical constraint (e.g., a 3 cm distance)
- `angle` (*float*): quantity for an angular constraint (e.g., 45 degrees)
- `clockwise` (*bool*): orientation of an angular constraint
- `aligned` (*bool*): whether the start and end points of primitives in an angular constraint are aligned in the angle computation
- `direction` (*enum*): the measurement type for a distance value (must be on of `minimum`, `vertical`, or `horizontal`)

- `halfSpace#` (*enum*): the relative positioning to be maintained by primitives in distance-based constraints (must be one of `left` or `right`)

Schemata

Below, we list each parameter schema and the constraints adhering to it. Note that some constraints can appear with more than one schema. For example, a horizontal constraint may act on a single primitive (specifying only `local0`) or two primitives (specifying both `local0` and `local1`). Numerical constraints are listed here with their most frequent schema, although a few other schemas may appear in the dataset.

(`local0`)

Horizontal, Vertical

(`local0`, `local1`)

Coincident, Horizontal, Vertical, Parallel, Perpendicular, Tangent, Midpoint, Equal, Offset, Concentric

(`local0`, `local1`, `local2`)

Mirror

(`local0`, `length`)

Diameter, Radius

(`local0`, `direction`, `length`)

Length

(`local0`, `local1`, `direction`, `halfSpace0`, `halfSpace1`, `length`)

Distance

(`local0`, `local1`, `aligned`, `clockwise`, `angle`)

Angle

Length	%
5 mm	3.51
1 cm	3.35
3 mm	2.76
0.5 in	2.41
2 mm	2.30
1 in	2.29
2 cm	2.17
4 mm	2.03
8 mm	1.89
0.25 in	1.88

Angle (deg)	%
45	18.02
15	7.93
60	6.58
120	6.58
30	6.42
135	5.87
90	4.00
10	2.72
20	2.69
150	1.50

Table 3.2: Frequencies of the most common values observed for `length` (left) and `angle` (right) parameters. All values are converted to common units for frequency computation. For `length`, we display the values in the units requiring the fewest digits. Note that although the standalone `Perpendicular` constraint is generally used for 90-degree angles, perpendicularity is sometimes imposed with an angular constraint as seen here.

Numerical parameter distributions

We examine the values observed for the two constraint parameters specifying a quantity: `length` and `angle`. As described above, the `length` parameter is used in several numerical constraints.

See Table 3.2 for the frequencies of the most common parameter values. Unsurprisingly, the most common angles tend to evenly divide 360 degrees. The most common length parameters tend to correspond to standard sizes of common parts (e.g., a 5 mm screw). Fig. 3.11 displays the cumulative frequency of parameter values when sorted from most to least frequent. The 300 most frequent values for `angle` and `length` account for 95.8% and 82.1% of all occurrences, respectively. `angle`, as a scale-invariant parameter, exhibits a bit less diversity than `length`.

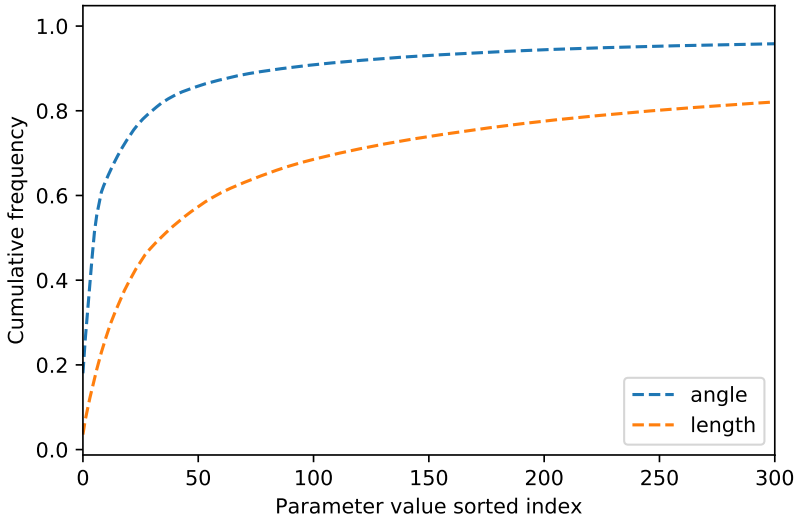


Figure 3.11: Cumulative frequency of unique parameter values when sorted from most to least frequent.

3.6.3 Example sketch constructions

Below, we render the construction steps for some of the example sketches in Fig. 3.1 according to the user-defined primitive orderings.

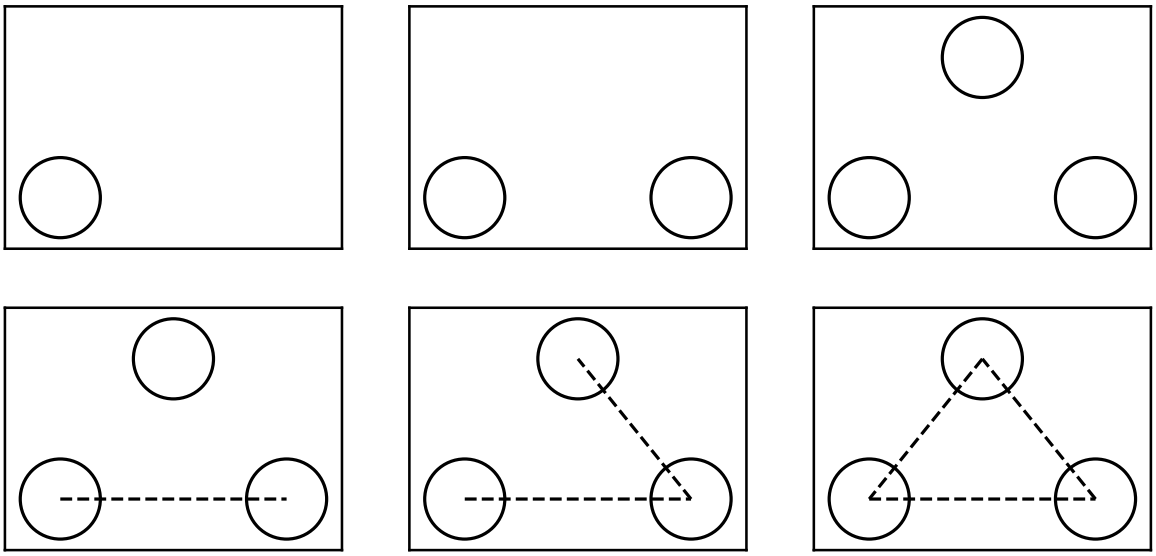


Figure 3.12: Construction of a dataset sketch proceeding from top left to bottom right.

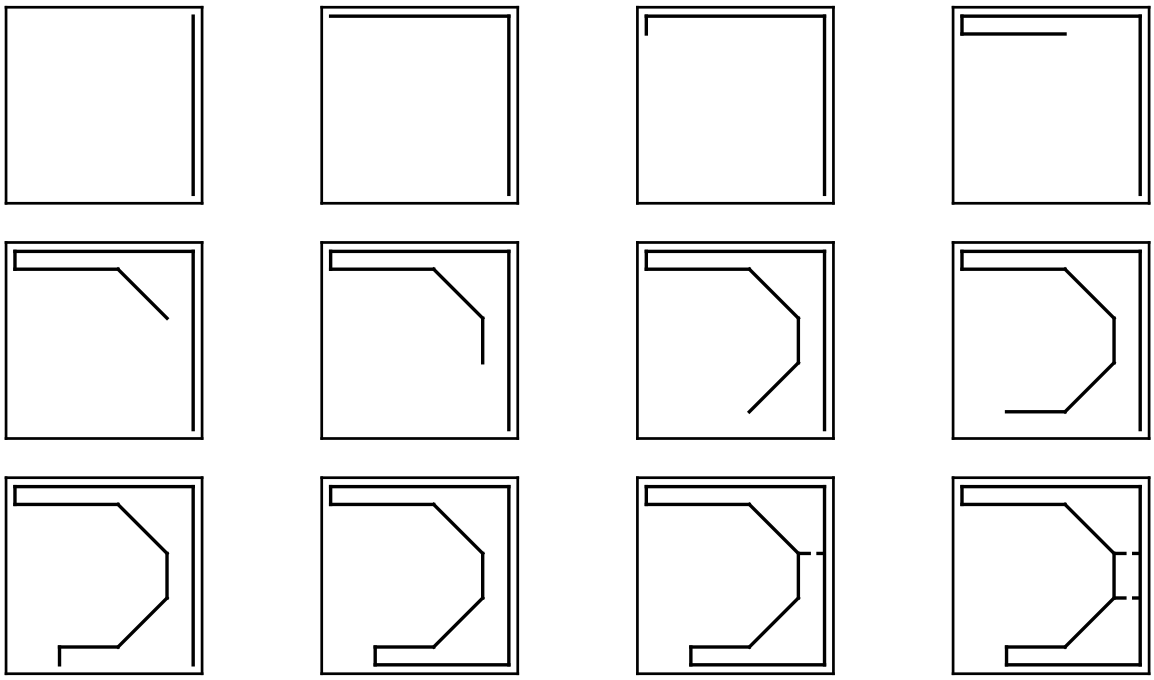


Figure 3.13: Construction of a dataset sketch proceeding from top left to bottom right.

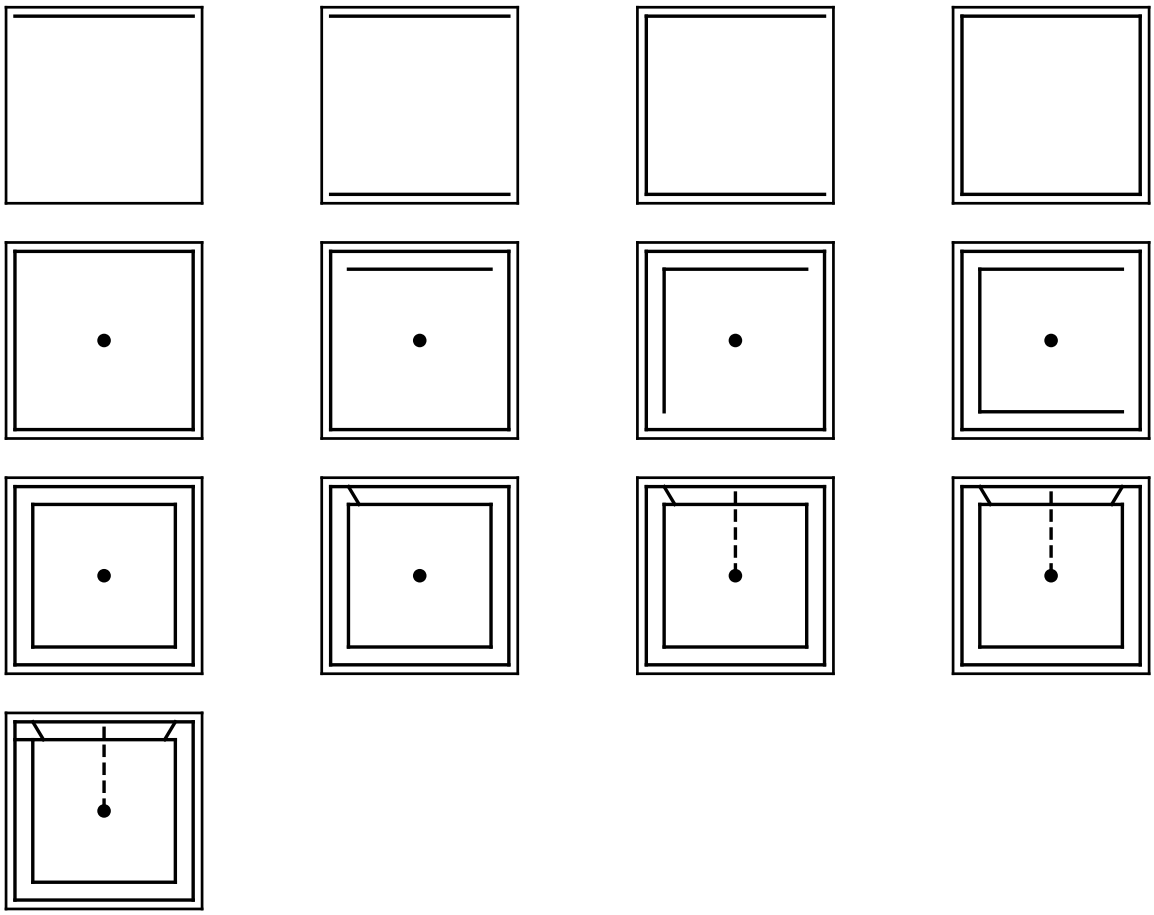


Figure 3.14: Construction of a dataset sketch proceeding from top left to bottom right.

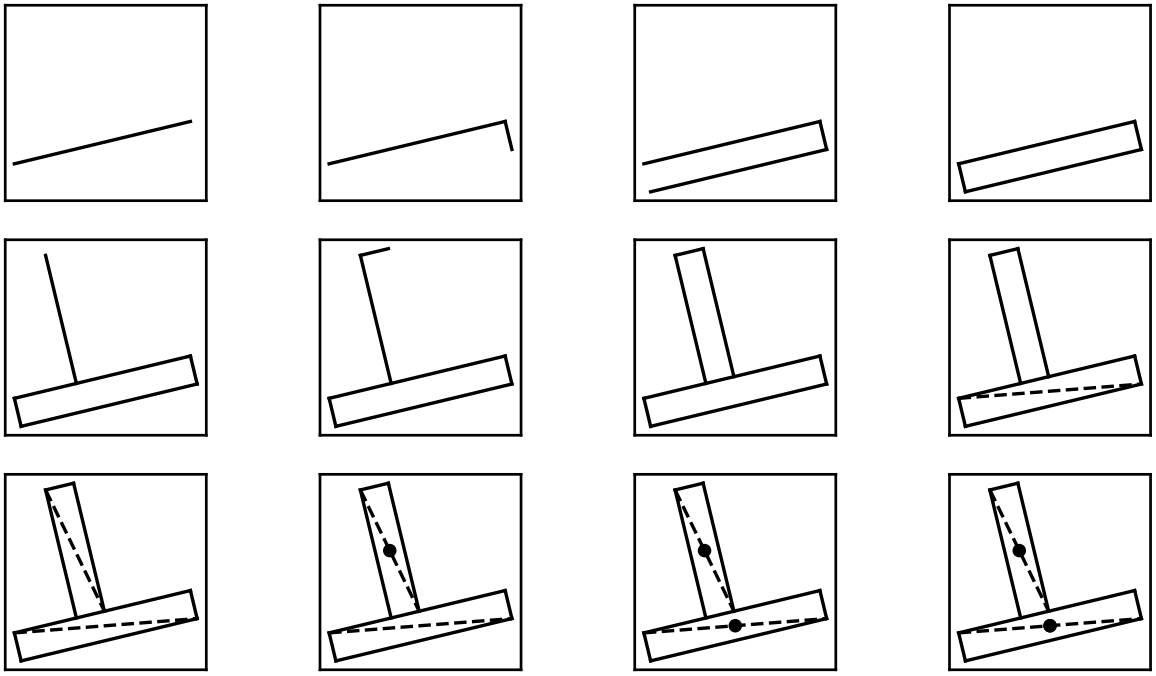


Figure 3.15: Construction of a dataset sketch proceeding from top left to bottom right.

Chapter 4

Vitruvion: A Generative Model of Parametric CAD Sketches

4.1 Introduction

Parametric computer-aided design (CAD) tools are at the heart of the design process in mechanical engineering, aerospace, architecture, and many other disciplines. These tools enable the specification of two- and three-dimensional structures in a way that empowers the user to explore parameterized variations on their designs, while also providing a structured representation for downstream tasks such as simulation and manufacturing. In the context of CAD, *parametric* refers to the dominant paradigm for professionals, in which the software tool essentially provides a graphical interface to the specification of a *constraint program* which can be then solved to provide a geometric configuration. This implicit programmatic workflow means that modifications to the design—alteration of dimensions, angles, etc.—propagate across the construction, even if it is an assembly with many hundreds or thousands of components.

At an operational level, parametric CAD starts with the specification of two-dimensional geometric representations, referred to as "sketches" in the CAD com-

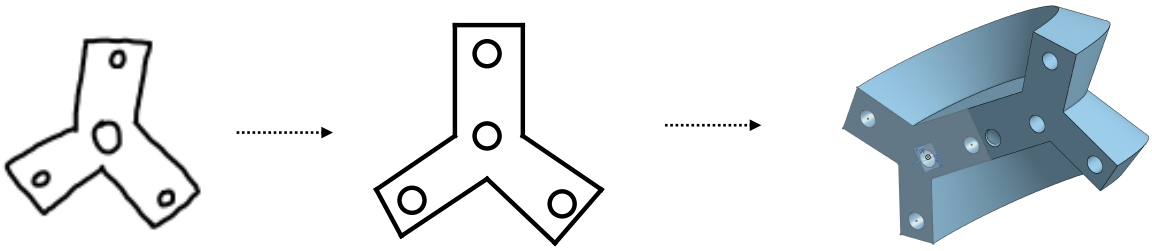


Figure 4.1: Example CAD design workflow enabled by our model. Our model first conditions on a raster image of a hand-drawn sketch (left) and generates sketch primitives (middle). Given the primitives, are set of constraints are then generated and the entire parametric sketch is imported, solved, and edited in CAD software, ultimately producing a 3D model (right).

munity [81, 10, 14]. A sketch consists of a collection of geometric primitives (e.g., lines, circles), along with a set of *constraints* that relate these primitives to one another. Examples of sketch constraints include parallelism, tangency, coincidence, and rotational symmetry. Constraints are central to the parametric CAD process as they reflect abstract design intent on the part of the user, making it possible for numerical modifications to coherently propagate into other parts of the design. These constraints are the basis of our view of parametric CAD as the specification of a program; indeed the actual file formats themselves are indistinguishable from conventional computer code.

This design paradigm, while powerful, is often a challenging and tedious process. An engineer may execute similar sets of operations many dozens of times per design. Furthermore, general motifs may be repeated across related parts, resulting in duplicated effort. Learning to accurately predict such patterns has the potential to mitigate the inefficiencies involved in repetitive manual design. In addition, engineers often begin visualizing a design by roughly drawing it on paper. The automatic and reliable conversion of such hand-drawings, or similarly noisy inputs such as 3D scans, to parametric, editable models remains a highly sought feature.

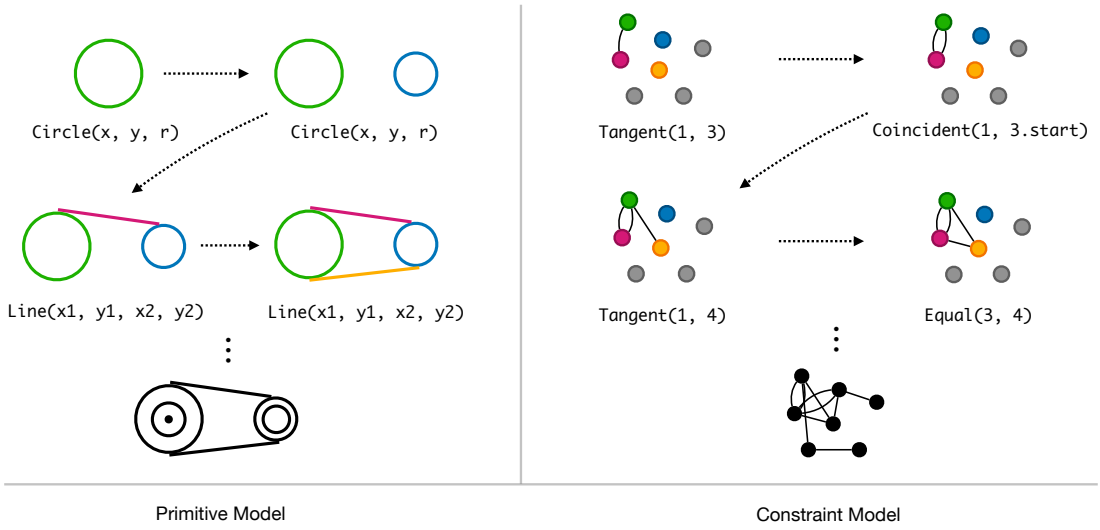


Figure 4.2: We factorize CAD sketch synthesis into two sequence modeling tasks: primitive generation (left) and constraint generation (right). The constraint model conditions on the present primitives and outputs a sequence of constraints, serving as edges in the corresponding geometric constraint graph. A separate solver (standard in CAD software) is used to adjust the primitive coordinates, if necessary.

In this work, we introduce Vitruvion,[1] a generative model trained to synthesize coherent CAD sketches by autoregressively sampling geometric primitives and constraints. The model employs self-attention [88] to flexibly propagate information about the current state of a sketch to a next-step prediction module. This next-step prediction scheme is iterated until the selection of a stop token signals a completed sample. The resultant geometric constraint graph is then handed to a solver that identifies the final configuration of the sketch primitives. By generating sketches via constraint graphs, we allow for automatic edit propagation in standard CAD software. Constraint supervision for the model is provided by the SketchGraphs dataset [78], which pairs millions of real-world sketches with their ground truth geometric constraint graphs.

In addition to evaluating the model via log-likelihood, we demonstrate the model's ability to aid in three application scenarios when conditioned on specific kinds of context. In **autocomplete**, we first prime the model with an incomplete sketch (e.g.,

[1]Our model's nickname, Vitruvion, is derived from "The Vitruvian Man", a 1490 drawing by Leonardo da Vinci depicting a man inscribed in a circle and square.

with several primitives missing) and query for plausible completion(s). In **autoconstrain**, the model is conditioned on a set of primitives (subject to position noise), and attempts to infer the intended constraints. We also explore **image-conditional synthesis**, where the model is first exposed to a raster image of a hand-drawn sketch. In this case, the model is tasked with inferring both the primitives and constraints corresponding to the sketch depicted in the image. Overall, we find the proposed model is capable of synthesizing realistic CAD sketches and exhibits potential to aid the mechanical design workflow.

4.2 Related Work

CAD sketch generation may be viewed via the broader lens of geometric program synthesis. This comprises a practical subset of program synthesis generally concerned with inferring a set of instructions (a program) for reconstructing geometry from some input specification. Recent examples include inferring TikZ drawing commands from images [24] and constructive solid geometry programs from voxel representations [82]. Related to these efforts is the highly-sought ability to accurately reverse engineer a mechanical part given underspecified and noisy observations of it (e.g,. scans, photos, or drawings). These applications depend on effective representation learning of raster and vector graphics [11]. We explore a variety of conditional settings in this work, for example, training a version of our model that generates parametric sketches when given a raster hand-drawing.

Generative modeling of CAD sketches has recently been studied using the SketchGraphs dataset [78], which provides ground truth geometric constraint graphs (and thus the original design steps) for millions of real-world CAD sketches. Initial models are trained in Seff et al. [78] for constraint graph generation, but without any learned attributes for the positions of the primitives in the sketch; the sketch configuration is

determined via constraints alone, limiting the diversity and sophistication of output samples. Willis et al. [91] models the primitive positions, but without specifying any geometric constraints. In contrast to both of the above, our model generates sketches via both learned primitive positions and constraints between them. The explicitly defined constraints enable edit propagation in standard CAD software; components of our generated sketches are updated appropriately in response to user-imposed changes.

This work is also related to modeling of graph structures, as we model CAD sketches as geometric constraint graphs. Graphs constitute a natural representation of relational structure across a variety of domains, including chemistry, social networks, and robotics. In recent years, message-passing networks [29, 22], building off of Scarselli et al. [75], have been extensively applied to generative modeling of graph-structured data, such as molecular graphs [43, 61]. These networks propagate information along edges, learning node or graph-level representations that serve as input to node and edge prediction modules.

While message passing between adjacent nodes can be a useful inductive bias, it can also be an impediment to effective learning when communication between distant nodes is necessary, e.g., in sparsely connected graphs [51]. The constraint graphs of concern in our work are sparse, as the constraints (edges) grow at most linearly in the number of primitives (nodes). Recent work leverages the flexible self-attention mechanism of transformers [88] to generate graph-structured data without the limitations of fixed, neighbor-to-neighbor communication paths, such as for polygonal mesh generation [72]. Our work similarly bypasses edge-specific message passing, using self-attention to train a generative model of geometric constraint graphs representing sketches.

4.3 Method

We aim to model a distribution over parametric CAD sketches. Each sketch is comprised of both a sequence of primitives, $\mathcal{P}$, and a sequence of constraints, $\mathcal{C}$. We note that these are sequences, as opposed to unordered sets, due to the ground truth orderings included in the data format [78]. Primitives and constraints are represented similarly in that both have a categorical type (e.g., Line, Arc, Coincidence) as well as a set of additional parameters which determine their placement/behavior in a sketch. Despite this similarity, primitives are fundamentally constructions which specify *what* is present, while constraints determine *how* these constructions are arranged by specifying geometric relationships between primitives. Rather than serving as static geometry, CAD sketches are intended to be dynamic—automatically updating in response to altered design parameters. Likewise, equipping our model with the ability to explicitly constrain generated primitives serves to ensure the preservation of various geometric relationships during downstream editing.

We divide the sketch generation task into three subtasks: primitive generation, constraint generation, and constraint solving. In our approach, the first two tasks utilize learned models, while the last step may be conducted by any standard geometric constraint solver.[2] We find that independently trained models for primitives and constraints allows for a simpler implementation. This setup is similar to that of Nash et al. [72], where distinct vertex and face models are trained for mesh generation.

The overall sketch generation proceeds as

$$\begin{aligned} p_\theta(\mathcal{P}, \mathcal{C} \mid \text{ctx}) &= p_\theta(\mathcal{C} \mid \mathcal{P}) p_\theta(\mathcal{P} \mid \text{ctx}) \\ \mathcal{S} &= \text{solve}(\mathcal{P}, \mathcal{C}) \end{aligned} \tag{4.1}$$

where:

[2]We use D-Cubed (Siemens PLM) for geometric constraint solving via Onshape's public API.

- $\mathcal{P}$ is a sequence of primitives, including parameters specifying their positions;
- $\mathcal{C}$ is a sequence of constraints, imposing mandatory relationships between primitives;
- $\mathcal{S}$ is the resulting sketch, formed by invoking a solve routine on the $(\mathcal{P}, \mathcal{C})$ pair; and
- ctx is an optional context for conditioning, such as an image or prefix (primer).

This factorization assumes $\mathcal{C}$ is conditionally independent of the context given $\mathcal{P}$. For example, in an image-conditional setting, access to the raster representation of a sketch is assumed to be superfluous for the constraint model given accurate identification of the portrayed primitives and their respective positions in the sketch. Unconditional samples may be generated by providing a null context.

4.3.1 Primitive Model

The primitive model is tasked with autoregressive generation of a sequence of geometric primitives. For each sketch, we factorize the distribution as a product of successive conditionals,

$$p_\theta(\mathcal{P} \mid \text{ctx}) = \prod_{i=1}^{N_\mathcal{P}} p_\theta(\mathcal{P}_i \mid \{\mathcal{P}_j\}_{j<i}, \text{ctx}), \tag{4.2}$$

where $N_\mathcal{P}$ is the number of primitives in the sequence. Each primitive is represented by a tuple of tokens indicating the type of primitive and its parameters. The primitive type may be one of four possible shapes (here either arc, circle, line, or point), and the associated parameters include both continuous positional coordinates and an `isConstruction` Boolean.[3] We use *sequence* to emphasize that our model treats sketches as constructions arising from a step-by-step design process.

[3] *Construction* or *virtual* geometry is employed in CAD software to aid in applying constraints to regular geometry. `isConstruction` specifies whether the given primitive is to be physically realized (when false) or simply serve as a reference for constraints (when true).

Ordering. Abstractly, the mapping from constructions sequences to final sketch geometries is clearly non-injective; it is typical for many construction sequences to result in the same final geometry. Nevertheless, empirically, we know that the design routes employed by engineers often admit evident patterns, e.g., greater involvement of earlier "anchor" primitives in constraints [78]. In our training data from SketchGraphs, the order of the design steps taken by the original sketch creators is preserved. Thus, instead of enforcing order-invariance in our model, we specifically expose it to this ordering, training our model to autoregressively predict subsequent design steps conditioned on prefixes. By following an explicit representation of this ordering, our model remains amenable to applications in which conditioning on such a prefix of design steps is required, such as autocompleting a sketch. Finally, ordering substantially constrains the target distribution in the sense that the model is not required to distribute its capacity among all possible orderings.

`Arc`	$(x_0, y_0, x_{\text{mid}}, y_{\text{mid}}, x_1, y_1)$
`Circle`	(x, y, r)
`Line`	(x_0, y_0, x_1, y_1)
`Point`	(x, y)

Table 4.1: Primitive parameterizations. We convert the native Onshape numerical parameters to the above forms for modeling. Subscripts of 0, mid, and 1 indicate start, mid, and endpoints, respectively.

Parameterization. SketchGraphs provides the original parameterization for each primitive as employed by Onshape.[4] In order to accommodate the interface to the Onshape constraint solver, these are often *over-parameterizations*; line segments, for example, are defined by six continuous parameters rather than four. We compress the original Onshape parameterizations into minimal canonical parameterizations for modeling and inflate them back to the original encoding prior to constraint solving.

Normalization. The sketches in SketchGraphs exhibit widely varying scales in physical space, spanning orders of magnitude from millimeters to meters. In addition,

[4] `https://www.onshape.com`

because Onshape users are free to place sketches anywhere in the coordinate plane, they are often not centered with respect to the origin in the coordinate space. To reduce modeling ambiguity, for each sketch we modify the primitive coordinates such that the sketch's square bounding box has a width of one meter and is centered at the origin.

Quantization. The sketch primitives are inherently described by continuous parameters such as x, y Cartesian coordinates, radii, etc. As is now a common approach in various continuous domains, we quantize these continuous variables and treat them as discrete. This allows for more flexible, yet tractable, modeling of arbitrarily shaped distributions [87, 72]. Following the sketch normalization procedure described in Section 4.3.1, we apply 6-bit uniform quantization to the primitive parameters. Lossy quantization is tolerable as the constraint model and solver are ultimately responsible for the final sketch configuration.

Tokenization. Each parameter (whether categorical or numeric) in the input primitive sequence is represented using a three-tuple comprised of three different token types: value, ID, and position. Value tokens play a dual role. First, a small portion of the range for value tokens is reserved to enumerate the primitive types. Second, the remainder of the range is treated separately as a set of (binned) values to specify the numerical value of some parameter, for example, the radius of a Circle primitive. ID tokens specify the type of parameter being described by each value token. For example, an ID token `xCenter` indicates that the corresponding value token is the x coordinate of a circle's center. Position tokens specify the ordered index of the primitive that the current element belongs to (e.g., a position of 3 means the given token is part of the third primitive).

Embeddings. We use learned embeddings for all tokens. That is, each value, ID, and position token associated with a parameter is independently embedded into a fixed-length vector. Then, we compute an element-wise sum across each of these three token embedding vectors to produce a single, fixed-length embedding to represent the parameter embedding. The sequence of parameter embeddings are then fed as input to the decoder described below.

Architecture. The unconditional primitive model is based on a decoder-only transformer [88] operating on the flattened primitive sequences. To respect the temporal dependencies of the primitive sequence during parallel training, the transformer decoder leverages standard triangular masking during self-attention. The last layer of the transformer outputs a final representation for each token that then passes through a linear layer to produce a logit for each possible value token. Taking a softmax over these logits gives the categorical distribution for the next token prediction task. We discuss conditional variants in Section 4.3.3. See the appendix for further architectural and hyperparameter details.

4.3.2 Constraint Model

Given a sequence of primitives, the constraint model is tasked with autoregressive generation of a sequence of constraints. As with the primitive model, we factorize the constraint model as a product of successive conditionals,

$$p_\theta(\mathcal{C} \mid \mathcal{P}) = \prod_{i=1}^{N_\mathcal{C}} p_\theta(\mathcal{C}_i \mid \{\mathcal{C}_j\}_{j<i}, \mathcal{P}), \tag{4.3}$$

where $N_\mathcal{C}$ is the number of constraints. Each constraint is represented by a tuple of tokens indicating its type and parameters.

Here, we focus our constraint modeling on handling all categorical constraints containing one or two reference parameters. This omits constraints with numerical

parameters (such as scale constraints) as well as "hyperedge" constraints that have more than two references.

Ordering. We canonicalize the order of constraints according to the order of the primitives they act upon, similar to Seff et al. [78]. That is, constraints are sorted according to their latest member primitive. For each constraint type, we arbitrarily canonicalize the ordering of its tuple of parameters. Constraint tuples specify the type of constraint and its reference parameters.

Tokenization. The constraint model employs a similar tokenization scheme to the primitive model in order to obtain a standardized sequence of integers as input. For each parameter, a triple of value, ID, and position tokens indicate the parameter's value, what type of parameter it is, and which ordered constraint it belongs to, respectively.

Architecture. The constraint model employs an encoder-decoder transformer architecture which allows it to condition on the input sequence of primitives. A challenge with the constraint model is that it must be able to specify references to the primitives that each constraint acts upon, but the number of primitives varies among sketches. To that end, we utilize variable-length softmax distributions (via pointer networks [89]) over the primitives in the sketch to specify which primitives the constraint references. This is similar to the methodology utilized in PolyGen [72], but importantly distinct in that our constraint model is equipped to reference hierarchical entities (i.e., primitives or sub-primitives).

Embeddings. Two embedding schemes are required for the constraint model. Both the input primitive token sequence as well as the target output sequence (constraint tokens) must be embedded. The input primitive tokens are embedded in identical

fashion as with the standalone primitive model, leveraging standard lookup tables. A transformer encoder then produces a sequence of final representations for the primitive tokens. These representation vectors serve a dual purpose: conditioning the constraint model and embedding references in the constraint sequence. The architecture of the encoder is similar to that of the primitive model, except it does not have the output linear layer and softmax.

From the tuple of representations for each primitive, we extract reference embeddings for both the primitive as a whole (e.g., a line segment) and each of its sub-primitives (e.g., a line endpoint) that may be involved in constraints.

Noise injection. The constraint model must account for potentially imperfect generations from the preceding primitive model. Accordingly, the constraint model is conditioned on primitives whose parameters are subjected to independent Gaussian noise during training.

4.3.3 Context Conditioning

Our model may be optionally conditioned on a context. As described below Eq. (4.1), we directly expose the primitive model to the context while the constraint model is only implicitly conditioned on it via the primitive sequence. Here, we consider two specific cases of context evocative of two applications: primers and images of hand-drawn sketches, corresponding to autocompletion and image-conditional generation applications, respectively.

Primer-conditional generation. Here, a primer is a sequence of primitives representing the prefix of an incomplete sketch. Conditioning on a primer consists of presenting the corresponding sequence prefix to a trained primitive model; the remainder of the primitives are sampled from the model until a stop token is sampled, indicating termination of the sequence. This emulates a component of an "auto-

complete" application, where CAD software interactively suggests next construction operations to the user.

Image-conditional generation. In the image-conditional setting, we are interested in accurately recovering a parametric sketch from a raster image of a hand-drawn sketch. This setup is inspired by the fact that engineers often draw a design on paper or whiteboard from various orthogonal views prior to investing time building it in a CAD program. Accurate inference of parametric CAD models from images or scans remains a highly-sought feature in CAD software, with the potential to dramatically reduce the effort of translating from rough paper scribble to CAD model.

When conditioning on an image, the primitive model is augmented with an image encoder. We leverage an architecture similar to a vision transformer [21], first linearly projecting flattened local image patches to a sequence of initial embeddings, then using a transformer encoder to produce a learned sequence of context embeddings. In order to condition on the image, the primitive decoder cross-attends into the image context embeddings. The overall model is trained to maximize the probability of the ground truth primitives portrayed in the image. In preliminary experiments, we found conditioning on a sequence of patch embeddings to be more effective than a single, global embedding per image. This is a similar setup to the image-conditioning route taken in Nash et al. [72] for mesh generation, which also leverages a sequence of context embeddings (instead produced by residual networks [36]).

Hand-drawn simulation. We aim to enable generalization of the image-conditional model described above to hand-drawn sketches, which can potentially support a much wider array of applications than only conditioning on perfect renderings. This requires the specification of a noise model to emulate the distortions introduced by imprecise hand drawing, as compared to the precise rendering of a parametric sketch using software. Two reasonable noise injection targets are

the parameters underlying sketch primitives and the raster rendering procedure. Our noise model takes a hybrid approach, subjecting sketch primitives to random translations/rotations in sketch space, and augmenting the raster rendering with a Gaussian process model. A full description is provided in Section 4.6.1.

4.4 Experiments

We evaluate several versions of our model in both primitive generation and constraint generation settings. Quantitative assessment is conducted by measuring negative log-likelihood (NLL) and predictive accuracy on a held-out test set. We also examine the model's performance on conditional synthesis tasks.

4.4.1 Training Dataset

Our models are trained on the SketchGraphs dataset [78], which consists of several million CAD sketches collected from Onshape. We use the filtered version, which is restricted to sketches comprised of the four most common primitives (arcs, circles, lines, and points), a maximum of 16 primitives per sketch (convenient for mini-batch training), and a non-empty set of constraints. We further filter out any sketches with fewer than six primitives to eliminate most trivial sketches (e.g., a simple rectangle). We randomly divide the filtered collection of sketches into a 92.5% training, 2.5% validation, and 5% testing partition. Training is performed on a server with four Nvidia V100 GPUs, and our models take between three and six hours to train. See Section 4.6.2 for full details of the training procedure.

Deduplication. A portion of the examples in SketchGraphs may be considered to be duplicates. These can result from Onshape's native document-copying functionality, online tutorials, files imported from popular online CAD collections, or just by

coincidence. We note that there are several reasonable notions of what may constitute duplicate sketches.

- *Exact* duplicates, the most conservative concept of duplicates, refers to sketches whose construction sequences are identical (including primitive coordinates).
- *Translation* duplicates differ only by their placement in the coordinate plane.
- *Primitive* duplicates differ only by constraints, but with identical primitives.
- *Permutation* duplicates encompass any of the above, but allow for a different ordering (e.g., two sketches where permuting the primitives of one results in exact duplicates).

Sketches may also exhibit non-identical yet visually similar configurations of primitives.

To ensure a diversity of training data as well as distinct partitions of training, validation, and test sets, we devise a straightforward deduplication procedure. We first normalize each sketch via centering and uniform rescaling. The numerical parameters in the primitive sequence are quantized to six bits, effectively mapping their coordinates to a 64×64 grid. At this stage, two sketches with the same concatenated sequence of primitives are considered duplicates, and we keep only one, resulting in a collection of 1.7 million unique sketches.

The above procedure removes all sketches that have similar geometry with the same ordering of construction operations. We deliberately elect to not remove permutation duplicates from our training. As described in Section 4.3.1, the order in which engineers execute CAD operations often contains meaningful information. Modeling the sketches according to these orderings is helpful in applications, particularly autocomplete.

	Bits per Primitive	Bits per Sketch	Accuracy
Uniform	33.41 (10.13)	350.2 (111.4)	1.3%
Compression	15.34	160.7	— %
Unconditional	6.35 (7.15)	66.6 (41.7)	71.7%
Unconditional (permuted)	7.84 (6.92)	82.1 (42.4)	62.8%
Image-conditional	3.80 (5.06)	39.8 (34.5)	80.2%

Table 4.2: Primitive model evaluation. We express negative log-likelihood on the test set in both bits per primitive and bits per sketch. Next-step prediction accuracy (per-token average) is also provided. Parentheses denote standard deviation.

4.4.2 Baselines

Because the few existing methods aimed at CAD sketch generation do not handle the generality of the modeling task we consider, we report the performance of two baselines: 1) A uniform sampler that randomly selects tokens with equal probability at each time step. 2) A compression baseline based on a LZMA compressor (see Section 4.6.3 for details). These baselines are accompanied by several ablation and conditional variants of our model.

4.4.3 Primitive Model Performance

Unconditional model. The unconditional variant of our model is trained to approximate the raw distribution over primitive sequences without regard to any context. As shown in Table 4.2, the unconditional primitive model achieves a substantially lower NLL than both the uniform and compression baselines. We employ nucleus sampling [39] with a commonly used cumulative probability parameter of $p = 0.9$ to remove low-probability tokens at each sampling step and avoid sample degradation. Fig. 4.3 displays a set of random samples from the unconditional primitive model.

To assess the degree to which the predictive performance of the model depends on information exposed through the sequence order, we conduct the following ab-

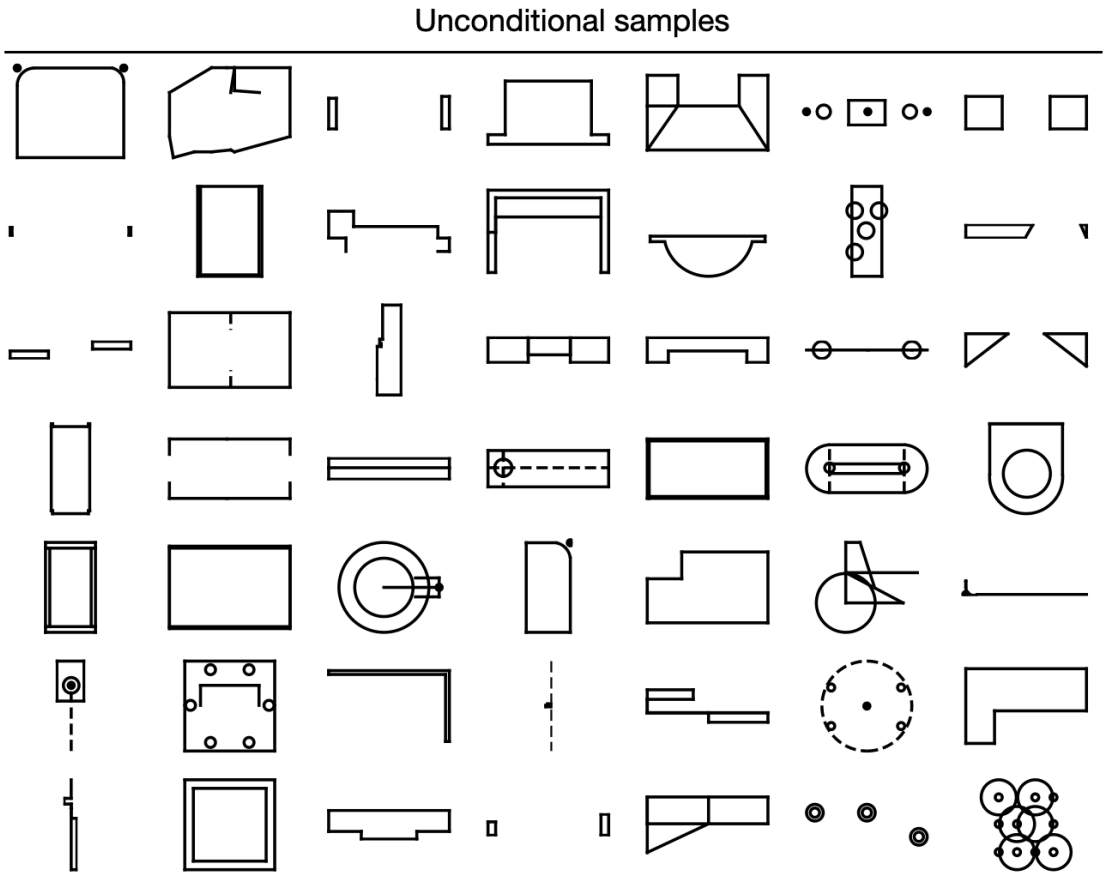


Figure 4.3: Unconditional samples from our raw primitive model.

lation study. We train an unconditional primitive model on sequences where the order of primitives has been shuffled uniformly randomly, removing the model's exposure to the ground truth ordering. As shown in Table 4.2, this results in significant performance degradation on next-step prediction. Importantly, the erasure of primitive ordering removes useful local hierarchical structure (e.g., high-level primitive constructions like rectangles) as well as global temporal structure.

Image-conditional generation. Table 4.2 displays the performance of the basic image-conditional variant of the primitive model. By taking a visual observation of the primitives as input, this model is able to place substantially higher probability on the target sequence.

	Bits per Primitive	Bits per Sketch	Accuracy
Image-conditional	22.8 (15.26)	292.3 (115.4)	47.0%
Image-conditional (hand-drawn)	10.59 (9.53)	135.5 (49.7)	65.5%
Image-conditional (hand-drawn + affine)	6.14 (6.80)	78.6 (36.8)	75.9%

Table 4.3: Image-conditional primitive model evaluated on real hand-drawn sketches. The hand-drawn variant includes noise injected by our Gaussian process noise model, while the affine augmentation introduces small affine transformations as a data augmentation strategy during training.

In Table 4.3, we evaluate the performance of several image-conditional models on a set of human-drawn sketches. These sketches were hand-drawn on a tablet computer in a 128×128-pixel bounding box, where the drawer first eyeballs a test set sketch, and then attempts to reproduce it. Likewise, each image is paired up with its ground truth primitive sequence, enabling log-likelihood evaluation.

We test three versions of the image-conditional model on the human hand-drawings which are trained on three different types of renderings: precise renderings, renderings from the hand-drawn simulator, and renderings with random affine augmentations of the hand-drawn simulator. Both the hand-drawn simulation and augmentations substantially improve performance. Despite never observing a real hand drawing during training, the model is able to effectively interpret these.

Figure 4.4 displays random samples from the primitive model when conditioned on images of hand-drawn sketches. For each drawing, four independent samples are generated.

Primer-conditional generation. We assess the model's ability to complete primitive sequences when only provided with an incomplete sketch. Here, we randomly select a subset of test set sketches, deleting 40% of the primitives from the suffix of the sequence. Four example, a sketch with ten primitives has the last four primitives deleted. The remaining sequence prefix then serves to prime the primitive model.

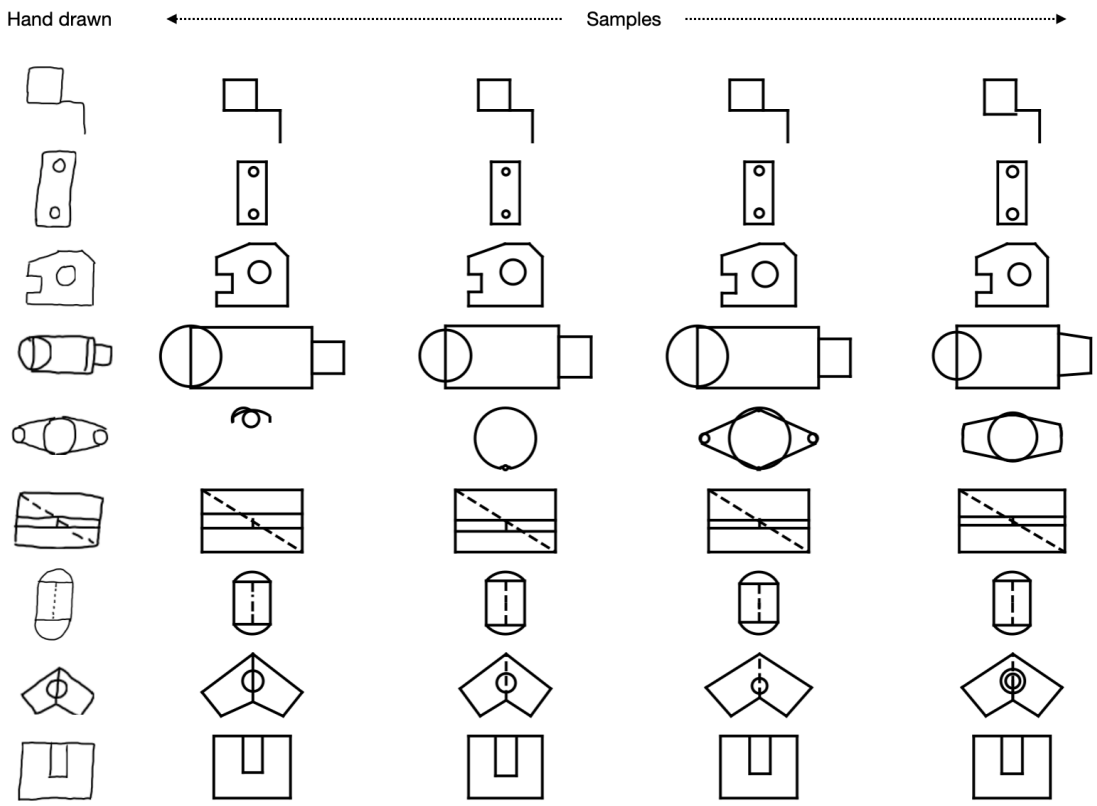


Figure 4.4: Image-conditional samples from our primitive model. Raster images of real hand-drawn sketches (left) are input to the primitive model. Four independent samples are then generated for each input. The model makes occassional mistakes, but tends to largely recover the depicted sketch.

Figure 4.5 displays random examples of priming the primitive model with incomplete sketches. Because just over half of the original primitives are present in the primer, there is a wide array of plausible completions. In some cases, despite only six completions for each input, the original sketch is recovered. We envision this type of conditioning as part of an interactive application where the user may query for k completions from a CAD program and then select one if it resembles their desired sketch.

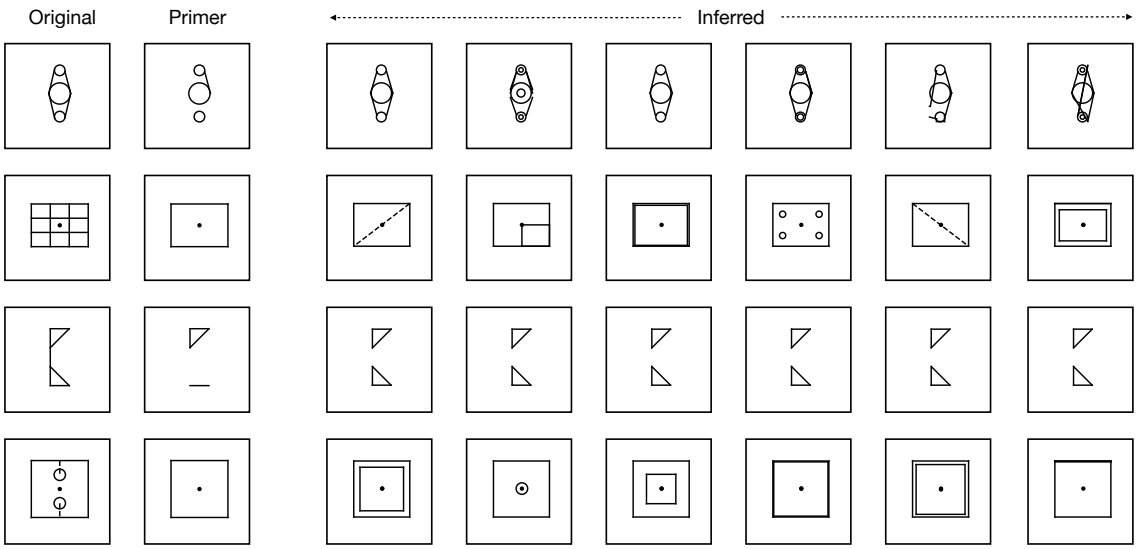


Figure 4.5: Primer-conditional generation. We take held-out sketches (left column) and delete 40% of their primitives, with the remaining prefix of primitives (second column from left) serving as a primer for the primitive model. To the right are the inferred completions, where the model is queried for additional primitives until selecting the stop token. Note that only the primitive model is used in these examples.

4.4.4 Constraint Model Performance

We train and test our constraint model with two different types of input primitive sequences: noiseless primitives and noise-injected primitives. Noise-injection proves to be a crucial augmentation in order for the constraint model to generalize to imprecise primitive placements (such as in the image-conditional setting). As shown in Table 4.4, while both versions of the model perform similarly on a noiseless test set, the model trained according to the original primitive locations fails to generalize outside this distribution. In contrast, exposing the model to primitive noise during training substantially improves performance.

Sketch editing. Figure 4.6 illustrates an end-to-end workflow enabled by our model. Our model is first used to infer primitives and constraints from a hand-drawn sketch. Subsequently, we show how editing the resulting solved sketch keeps

	Noiseless Testing		Noisy Testing	
Model	Perplexity	Accuracy	Perplexity	Accuracy
Uniform	4.984	0.6%	4.984	0.6%
Noiseless training	0.184	91.5%	0.903	68.6%
Noisy training	0.198	90.3%	0.203	90.6%

Table 4.4: Constraint model evaluation. We express per-token negative log-likelihood (perplexity) and accuracy, for both noisy and noiseless input. We note that a model trained only on noiseless input degrades very quickly in the presence of noise.

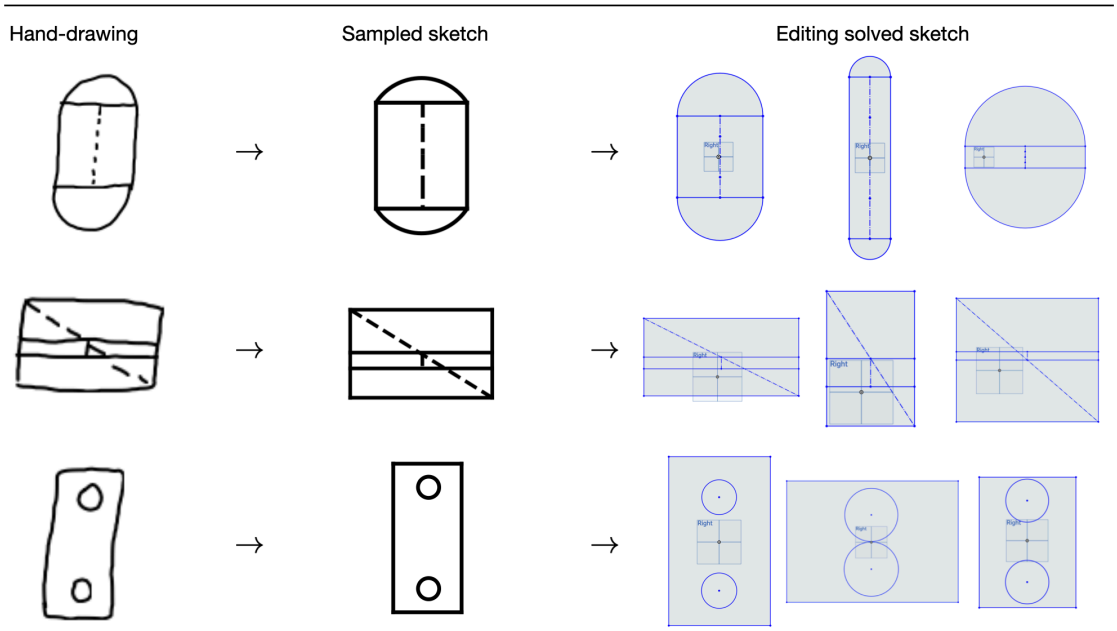


Figure 4.6: Hand-drawn sketches converted to Onshape editable models. We take raster images of hand-drawn sketches (left), infer the sketch using our image-conditional primitive model and constraint model (center), then solve the resultant sketch in Onshape and illustrate the edit propagation arising from the constraints (right).

the sketch in a coherent state due to edit propagation arising from the inferred constraints.

4.5 Conclusion and Future Work

In this work, we have introduced a method for modeling and synthesis of parametric CAD sketches, the fundamental building blocks of modern mechanical design. Adapting recent progress in modeling irregularly structured data with graph and attention-based models, our model leverages the flexibility of self-attention to propagate sketch information during autoregressive generation.

Avenues for future work include extending the generative modeling framework to capture the more general class of 3D inputs; including modeling 3D construction operations and synthesis conditioned on noisy 3D scans or voxel representations.

The goal of this work is to automate a tedious process and enable greater creativity in design, but we acknowledge the potential societal impact from eliminating human labor. Moreover, there is the potential risk in high-stakes applications for automatically synthesizing mechanical systems that do not meet human structural standards. We view this work, however, as being part of a larger human-in-the-loop process which will still involve human expertise for validation and deployment.

4.6 Appendix

4.6.1 Hand-drawn Noise Model

The noise model renders lines as drawn from a zero-mean Gaussian process with a Matérn-3/2 kernel that have been rotated into the correct orientation. Arcs are rendered as Matérn Gaussian process paths in polar coordinates with angle as the input and the random function modulating the radius. No additional observation noise is introduced beyond the standard "jitter" term. The length-scale and amplitude are chosen, and the Gaussian process truncated, so that scales are consistent regardless of the length of the line or arc.

4.6.2 Experimental Details

This section provides details of the model architecture and training regime.

Model Architecture

Our models all share a main transformer responsible for processing the sequence of primitives or constraints which is the target of the inference problem. This transformer architecture is identical across all of the models, using a standard transformer decoder with 12 layers, 8 attention heads, and a total embedding dimension of 256. No significant hyper-parameter optimization was performed. The raw primitive generative model is only equipped with the decoder on the primitive sequence, and is trained in an autoregressive fashion. The constraint model, in addition to the transformer decoder on the constraint sequence, also performs standard cross-attention into an encoded representation of the primitive sequence.

The image-to-primitive model is additionally equipped with an image encoder based on a vision transformer [21], which is tasked with producing a sequence of image representations into which the primitive decoder performs cross-attention. We use 128×128-sized images as input to the model. Non-overlapping square patches of size 16×16 are extracted from an input image and flattened, producing a sequence of 64 flattened patches. Each then undergoes a linear transformation to the model's embedding dimension (in this case, 256) before entering a standard transformer encoder. This transformer encoder has the same size as the decoder used in our other models.

Training

Training was performed on cluster servers equipped with four Nvidia V100 GPUs. Total training time was just under 2 hours for the primitive model, and 5.5 hours for the image-to-primitive model and the constraint model. The primitive model and the

constraint model were each trained for 30 epochs, and the image-to-primitive model was trained for 40 epochs, although the total number of epochs did not appear to cause significant differences in performance. The models were trained using the Adam optimizer with "decoupled weight decay regularization" [63], with the learning rate set according to a "one-cycle" learning rate schedule [84]. The initial initial learning rate was set to 3e-5 (at reference batch size 128, scaled linearly with the total batch size). The batch size was set according to the memory usage of the different models at 1024 / GPU for the raw primitive model, 512 / GPU for the image-to-primitive model, and 384 / GPU for the constraint model.

Image-to-primitive data

Prior to training the image-to-primitive model, we generate samples from our hand-drawn noise model (see Section 4.6.1) in a separate process. Five samples are generated for each sketch in the dataset, taking a total of approximately 160 CPU-hours. This process is parallelized on a computing cluster. During each training epoch, one random image (out of the five generated samples) is selected to be used by the model. An additional data augmentation process is used during training, whereby random affine transformations are applied to the images. The augmentation is chosen randomly among all affine transformations which translate the image by at most 8 pixels, rotate by at most 10 degrees, shear the image by at most 10 degrees, and scale the image by at most 20%.

4.6.3 Compression Baseline

The compression baseline is computed by first tokenizing primitives according to the tokenization scheme described in section 4.3.1, representing the token sequences as 8-bit integers, and compressing the given data using the LZMA compression algorithm. The compression is performed over the entire dataset (including training, validation

and test splits), and the reported quantities are averages over the entire dataset. In particular, note that we may expect the entropy rate of the data to be lower than reported.

Chapter 5

Open Directions

Learning-aided design is very much in its infancy. Systems that learn from domain-specific data and assist humans in engineering new solutions have the potential to fundamentally alter current design processes. But developing such systems is no simple task. A key challenge is the irregular and highly specific data structures often accompanying engineering and scientific domains.

In this thesis, we took a probabilistic perspective, examining the role that generative modeling over arbitrarily structured spaces may begin to play in this sphere. In Chapter 2, we discussed a general framework for modeling discrete objects with strict validity requirements and unknown construction histories. Such objects are prevalent in molecular and other graph-structured domains. In Chapters 3 and 4, we focused our attention on mechanical design, an area where learned systems are only just beginning to penetrate. We constructed SketchGraphs, the first large-scale CAD sketch dataset with geometric constraint supervision, in an effort to facilitate machine learning research in this underexplored domain. The dataset has already been used by several engineering firms to develop new models. We also presented Vitruvion, a model that is trained to autoregressively generate CAD sketches as a sequence of primitives and constraints, matching the data structure employed in

CAD software. This allows for direct importing and downstream editing. Evaluation demonstrated the potential for this approach to aid design, particularly in image and primer-conditional settings.

However, probabilistic generative modeling only addresses one piece of the puzzle. Such models are, in a sense, simply regurgitating the patterns observed during training. While their utility is substantial, they are ill-equipped for goal-oriented design on their own. Ultimately, a system should be capable of taking some high-level design criteria and proceeding to search for plausible solutions. Of course, given appropriate training data, we can always develop conditional generative models that construct attribute-specific examples, but these are not sufficient for finding an optimal design under several interacting constraints. Rather than serving as a standalone tool, generative models may be used in tandem with search techniques incorporating knowledge of functionality and causality to find promising designs while adhering to the "realistic" data manifold. This constitutes a compelling paradigm for future investigation.

Today's generative models are also quite inflexible. By default they must adhere to the factorization employed during training; a standard autoregressive model trained to output sequence tokens from left to right may only be directly primed on left-prefixes at deployment time. This often requires multiple versions of the same model, each with a different factorization, to be trained for separate applications on the same domain. For example, the factorization of our Vitruvion model expects all primitives to be generated prior to any constraints. In a real CAD product, one may desire a trained autocomplete model that efficiently conditions on any subset of a sketch. Ultimately, models that support this type of arbitrary conditioning at deployment will enable more flexible usage.

Aside from direct design applications, endowing machines with the capability to reason about "how stuff is made" may unlock new routes in tangential areas of ar-

tificial intelligence. The compositional reasoning required in design may lead to, for example, rich priors in computer vision and robotics or the discovery of modularity in general program induction tasks. The human imagination extends beyond the consideration of "what is", pondering "what may be". An artificial reproduction of human-like intelligence should be capable of the same.

Bibliography

[1] Guillaume Alain, Yoshua Bengio, Li Yao, Jason Yosinski, Eric Thibodeau-Laufer, Saizheng Zhang, and Pascal Vincent. GSNs: Generative stochastic networks. *Information and Inference*, 2016.

[2] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3 (Feb):1137–1155, 2003.

[3] Yoshua Bengio, Li Yao, Guillaume Alain, and Pascal Vincent. Generalized denoising auto-encoders as generative models. In *Advances in Neural Information Processing Systems*, 2013.

[4] Bernhard Bettig and Christoph M Hoffmann. Geometric constraint solving in parametric computer-aided design. *Journal of Computing and Information Science in Engineering*, 11(2):021001, 2011.

[5] G. Richard Bickerton, Gaia V. Paolini, Jérémy Besnard, Sorel Muresan, and Andrew L. Hopkins. Quantifying the chemical beauty of drugs. *Nature Chemistry*, 4, 2012.

[6] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale GAN training for high fidelity natural image synthesis. In *International Conference on Learning Representations*, 2019.

[7] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[8] Nathan Brown, Marco Fiscato, Marwin H.S. Segler, and Alain C. Vaucher. Guacamol: Benchmarking models for de novo molecular design. *Journal of Chemical Information and Modeling*, 59(3):1096–1108, 2019.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever,

and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, 2020.

[10] Jorge D Camba, Manuel Contero, and Pedro Company. Parametric CAD modeling: An analysis of strategies for design reusability. *Computer-Aided Design*, 74:18–31, 2016.

[11] Alexandre Carlier, Martin Danelljan, Alexandre Alahi, and Radu Timofte. DeepSVG: A hierarchical generative network for vector graphics animation. *arXiv preprint arXiv:2007.11301*, 2020.

[12] Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. ShapeNet: An Information-Rich 3D Model Repository. Technical Report arXiv:1512.03012 [cs.GR], Stanford University — Princeton University — Toyota Technological Institute at Chicago, 2015.

[13] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734. Association for Computational Linguistics, October 2014.

[14] Guk-Heon Choi, Du-Hwan Mun, and Soon-Hung Han. Exchange of CAD part models based on the macro-parametric approach. *International Journal of CAD/CAM*, 2(1):13–21, 2002.

[15] John Comer and Kin Tam. *Lipophilicity Profiles: Theory and Measurement*, pages 275–304. John Wiley & Sons, Ltd, 2007.

[16] Allen Cypher and Daniel Conrad Halbert. *Watch What I Do: Programming by Demonstration*. MIT press, 1993.

[17] Hanjun Dai, Yingtao Tian, Bo Dai, Steven Skiena, and Le Song. Syntax-directed variational autoencoder for structured data. In *International Conference on Learning Representations*, 2018.

[18] N. De Cao and T. Kipf. MolGAN: An implicit generative model for small molecular graphs. In *ICML Workshop on Theoretical Foundations and Applications of Deep Generative Models*, 2018.

[19] Eyal Dechter, Jon Malmaud, Ryan P Adams, and Joshua B Tenenbaum. Bootstrap learning via modular concept discovery. In *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

[20] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

[21] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.

[22] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alan Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in Neural Information Processing Systems*, 2015.

[23] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. Unsupervised learning by program synthesis. In *Advances in Neural Information Processing Systems*, pages 973–981, 2015.

[24] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. In *Advances in Neural Information Processing Systems 31*, 2018.

[25] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. In *Advances in Neural Information Processing Systems 32*, 2019.

[26] P. Erdös and A. Rényi. On random graphs i. *Publicationes Mathematicae Debrecen*, 6:290, 1959.

[27] Peter Ertl and Ansgar Schuffenhauer. Estimation of synthetic accessibility score of drug-like molecules based on molecular complexity and fragment contributions. *Journal of Cheminformatics*, 1:8, 2009.

[28] Robert Gallager. Low-density parity-check codes. *IRE Transactions on Information Theory*, 8(1):21–28, 1962.

[29] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, 2017.

[30] Rafael Gómez-Bombarelli, David K. Duvenaud, José Miguel Hernández-Lobato, Jorge Aguilera-Iparraguirre, Timothy D. Hirzel, Ryan P. Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. In *ACS Central Science*, 2018.

[31] Rafael Gómez-Bombarelli, Jennifer N Wei, David Duvenaud, José Miguel Hernández-Lobato, Benjamín Sánchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D Hirzel, Ryan P Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science*, 4(2):268–276, 2018.

[32] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H Muggleton, Ute Schmid, and Benjamin Zorn. Inductive programming meets the real world. *Communications of the ACM*, 58(11):90–99, 2015.

[33] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[34] David Ha and Douglas Eck. A neural representation of sketch drawings. In *International Conference on Learning Representations*, 2018.

[35] Ruth Haas, David Orden, Günter Rote, Francisco Santos, Brigitte Servatius, Herman Servatius, Diane Souvaine, Ileana Streinu, and Walter Whiteley. Planar minimally rigid graphs and pseudo-triangulations. *Computational Geometry*, 31 (1):31 – 61, 2005. Special Issue on the 19th Annual Symposium on Computational Geometry - SoCG 2003.

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *European Conference on Computer Vision*, 2016.

[37] Mikael Henaff, Kevin Jarrett, Koray Kavukcuoglu, and Yann LeCun. Unsupervised learning of sparse features for scalable audio classification. In *ISMIR*, volume 11, page 2011, 2011.

[38] L. Henneberg. Die graphische statik der starren systeme. *Johnson Reprint 1968*, 1911.

[39] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020.

[40] Irvin Hwang, Andreas Stuhlmüller, and Noah D Goodman. Inducing probabilistic programs by Bayesian program merging. *arXiv preprint arXiv:1110.5667*, 2011.

[41] Donald J. Jacobs and Bruce Hendrickson. An algorithm for two-dimensional rigidity percolation: The pebble game. *Journal of Computational Physics*, 137 (2):346 – 365, 1997.

[42] Dave Janz, Jos van der Westhuizen, Brooks Paige, Matt Kusner, and José Miguel Hernández-Lobato. Learning a generative model for validity in complex discrete structures. In *International Conference on Learning Representations*, 2018.

[43] Wengong Jin, Regina Barzilay, and Tommi S. Jaakkola. Junction tree variational autoencoder for molecular graph generation. *International Conference on Machine Learning*, 2018.

[44] Wengong Jin, Regina Barzilay, and Tommi S. Jaakkola. Junction tree variational autoencoder for molecular graph generation. *International Conference on Machine Learning*, 2018.

[45] Jonas Jongejan, Henry Rowley, Takashi Kawashima, Jongmin Kim, and Nick Fox-Gieg. The Quick, Draw! - A.I. experiment. `https://quickdraw.withgoogle.com`, 2016.

[46] Koray Kavukcuoglu, Marc'Aurelio Ranzato, Rob Fergus, and Yann LeCun. Learning invariant features through topographic filter maps. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1605–1612. IEEE, 2009.

[47] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015*, 2015.

[48] Durk P Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. In *Advances in Neural Information Processing Systems 31*, 2018.

[49] Sebastian Koch, Albert Matveev, Zhongshi Jiang, Francis Williams, Alexey Artemov, Evgeny Burnaev, Marc Alexa, Denis Zorin, and Daniele Panozzo. ABC: A big CAD model dataset for geometric deep learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[50] Frank R Kschischang, Brendan J Frey, Hans-Andrea Loeliger, et al. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, 2001.

[51] Vitaly Kurin, Maximilian Igl, Tim Rocktäschel, Wendelin Boehmer, and Shimon Whiteson. My body is a cage: the role of morphology in graph-based incompatible control. In *International Conference on Learning Representations*, 2021.

[52] Matt J. Kusner, Brooks Paige, and José Miguel Hernández-Lobato. Grammar variational autoencoder. In *International Conference on Machine Learning*, 2017.

[53] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *International Conference on Machine Learning*, pages 282–289, 2001.

[54] G. Laman. On graphs and rigidity of plane skeletal structures. *Journal of Engineering Mathematics*, 4(4):331–340, Oct 1970.

[55] Greg Landrum et al. RDKIT: Open-source cheminformatics, 2006.

[56] Ming Li, Frank C. Langbein, and Ralph R. Martin. Detecting design intent in approximate CAD models using symmetry. *Computer-Aided Design*, 42(3):183 – 201, 2010.

[57] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. Learning deep generative models of graphs. In *International Conference on Machine Learning*, 2018.

[58] Percy Liang, Michael I Jordan, and Dan Klein. Learning programs: A hierarchical Bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning*, pages 639–646, 2010.

[59] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7):1019–1031, 2007.

[60] Jenny Liu, Aviral Kumar, Jimmy Ba, Jamie Kiros, and Kevin Swersky. Graph normalizing flows. In *Advances in Neural Information Processing Systems 32*, 2019.

[61] Qi Liu, Miltiadis Allamanis, Marc Brockschmidt, and Alexander Gaunt. Constrained graph variational autoencoders for molecule design. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, pages 7795–7804, 2018.

[62] Shikun Liu, Lee Giles, and Alexander Ororbia. Learning a hierarchical latent-variable model of 3d shapes. In *2018 International Conference on 3D Vision (3DV)*. IEEE, 2018.

[63] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, 2019*, 2019.

[64] Linyuan Lü and Tao Zhou. Link prediction in complex networks: A survey. *Physica A: statistical mechanics and its applications*, 390(6):1150–1170, 2011.

[65] Mark P. Mattson. Superior pattern processing is the essence of the evolved human brain. *Frontiers in Neuroscience*, 2014. ISSN 1662-4548.

[66] David Mendez, Anna Gaulton, A Patrícia Bento, Jon Chambers, Marleen De Veij, Eloy Félix, María Paula Magariños, Juan F Mosquera, Prudence Mutowo, Michał Nowotka, María Gordillo-Marañón, Fiona Hunter, Laura Junco, Grace Mugumbate, Milagros Rodriguez-Lopez, Francis Atkinson, Nicolas Bosc, Chris J Radoux, Aldo Segura-Cabrera, Anne Hersey, and Andrew R Leach. ChEMBL: towards direct deposition of bioassay data. *Nucleic Acids Research*, 47(D1):D930–D940, 11 2018.

[67] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *International Conference on Machine Learning*, pages 187–195, 2013.

[68] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[69] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

[70] Adel Moussaoui. *Geometric Constraint Solver. (Solveur de systèmes de contraintes géométriques)*. PhD thesis, Ecole nationale Supérieure d'Informatique, Algiers, Algeria, 2016.

[71] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *International Conference on Learning Representations*, 2018.

[72] Charlie Nash, Yaroslav Ganin, S. M. Ali Eslami, and Peter W. Battaglia. PolyGen: An autoregressive generative model of 3D meshes. *ICML*, 2020.

[73] Kristina Preuer, Philipp Renz, Thomas Unterthiner, Sepp Hochreiter, and Günter Klambauer. Fréchet chemnet distance: A metric for generative models for molecules in drug discovery. *Journal of Chemical Information and Modeling*, 58(9):1736–1741, 2018.

[74] Patsorn Sangkloy, Nathan Burnell, Cusuh Ham, and James Hays. The sketchy database: Learning to retrieve badly drawn bunnies. *ACM Transactions on Graphics (proceedings of SIGGRAPH)*, 2016.

[75] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20:61–80, 2009.

[76] Paul J. Schweitzer. Perturbation theory and finite Markov chains. *Journal of Applied Probability*, 5(2):401–413, 1968.

[77] Ari Seff, Wenda Zhou, Farhan Damani, Abigail Doyle, and Ryan P. Adams. Discrete object generation with reversible inductive construction. In *Advances in Neural Information Processing Systems 32*, 2019.

[78] Ari Seff, Yaniv Ovadia, Wenda Zhou, and Ryan P. Adams. SketchGraphs: A large-scale dataset for modeling relational geometry in computer-aided design. In *ICML 2020 Workshop on Object-Oriented Learning*, 2020.

[79] Ari Seff, Wenda Zhou, Nick Richardson, and Ryan P. Adams. Vitruvion: A generative model of parametric CAD sketches. In preparation, 2021.

[80] Marwin H. S. Segler, Thierry Kogej, Christian Tyrchan, and Mark P. Waller. Generating focused molecule libraries for drug discovery with recurrent neural networks. *ACS Central Science*, 4(1):120–131, 2018.

[81] Jami J Shah. Designing with parametric CAD: Classification and comparison of construction techniques. In *International Workshop on Geometric Modelling*, pages 53–68. Springer, 1998.

[82] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. Csgnet: Neural shape parser for constructive solid geometry. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[83] Martin Simonovsky and Nikos Komodakis. GraphVAE: Towards generation of small graphs using variational autoencoders. In *ICAN*, 2018.

[84] Leslie Smith and Nicholay Topin. Super-convergence: Very fast training of neural networks using large learning rates. *arXiv preprint*, 2018.

[85] Teague Sterling and John J. Irwin. ZINC 15 – ligand discovery for everyone. *Journal of Chemical Information and Modeling*, 55(11):2324–2337, 2015.

[86] Ben Taskar, Ming-Fai Wong, Pieter Abbeel, and Daphne Koller. Link prediction in relational data. In *Advances in Neural Information Processing Systems*, pages 659–666, 2004.

[87] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alexander Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. In *Arxiv*, 2016. URL `https://arxiv.org/abs/1609.03499`.

[88] A Vaswani, N Shazeer, N Parmar, J Uszkoreit, L Jones, AN Gomez, L Kaiser, and I Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

[89] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Neural Information Processing Systems*, pages 2692–2700, 2015.

[90] David Weininger. SMILES, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of Chemical Information and Computer Sciences*, 28(1):31–36, 1988.

[91] Karl D. D. Willis, Pradeep Kumar Jayaraman, Joseph G. Lambourne, Hang Chu, and Yewen Pu. Engineering sketch generation for computer-aided design, 2021.

[92] Jiajun Wu, Chengkai Zhang, Tianfan Xue, William T Freeman, and Joshua B Tenenbaum. Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling. In *Advances in Neural Information Processing Systems*, 2016.

[93] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao. 3d shapenets: A deep representation for volumetric shapes. In *Computer Vision and Pattern Recognition*, 2015.

[94] Pengcheng Yin and Graham Neubig. A syntactic neural model for general purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2017.

[95] Jiaxuan You, Rex Ying, Xiang Ren, William Hamilton, and Jure Leskovec. GraphRNN: Generating realistic graphs with deep auto-regressive models. In *International Conference on Machine Learning*, 2018.

[96] Gui-Fang Zhang. Well-constrained completion for under-constrained geometric constraint problem based on connectivity analysis of graph. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, page 1094–1099, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450301138. doi: 10.1145/1982185.1982427.

[97] Gui-Fang Zhang and Xiao-Shan Gao. Well-constrained completion and decomposition for under-constrained geometric constaint problems. *International Journal of Computational Geometry & Applications*, 16(05n06):461–478, 2006. doi: 10.1142/S0218195906002142.