Improving Data-Driven Design and Exploration of Digital Musical Instruments

Chris Laguna
University of California, San Diego and Princeton University
cplaguna@ucsd.edu

DREU Mentor: Rebecca Fiebrink, Princeton University

Abstract

We are developing a system that automatically generates transformations from an input space to an output space. This system can be used to design digital instruments—a process that requires a transformation between physical gestures (a user playing an instrument) and sound synthesis (the sounds that instrument makes). Once generated, the transformations (typically called "mappings" in the research literature) can be run in real time, allowing users to play instruments. We aim to support designers who do not yet have a complete understanding of the input and output spaces and want a tool to that helps them explore possible relationships between the spaces. Studies of the Wekinator, another tool for creating real-time mappings, motivate us to design new methods for generating mappings given less initial information. We implemented a graphical user interface allowing users to generate mappings using our new methods. Following design principles for supporting creative thinking, suggested by Resnick et al., this interface encourages users to explore multiple mappings by supporting side-by-side mapping creation and evaluation. Future work will implement the stash and history tree, tools that encourage designers to explore alternate mappings and alternate input and output spaces without risking loss of current work. Further studies will also provide user evaluations of our system.

Introduction

Digital instrument designers must decide what it means to play an instrument: what gestures should control sound, and to what extent will variations from these gestures result in different sounds? This is essentially a mapping problem between input features (often extracted from sensors tracking physical gestures) and output parameters (that will drive sound synthesis or audio playback). Sometimes, digital instruments attempt to look and feel like existing acoustic instruments, in which case the designer has a preconceived notion about how the input gestures should be mapped to output parameters. Other times, the designer is creating a completely new instrument, and may not have strong ideas about what input gestures to use or how they should affect output parameters. These designers may want to explore many different potential mappings. Our focus is on the latter group of designers.

First, we give an overview of the Wekinator, a creative support tool with a similar target user as us, and we make observations about the Wekinator that motivate our work. Then, we give our system overview, explaining the new metaphors we designed that help the user understand our system in a way that encourages the user to explore different mappings side by side. Finally, we describe our graphical user interface that allows users to automatically generate edit, and compare mappings side by side.

Related Work

Our work is inspired by the Wekinator [Fiebrink, 2011], a creative support tool targeted at digital instrument designers. The Wekinator supports the creation of mappings through interactive supervised learning.

Users map input gestures to output parameters by providing labeled training data. Users can enter new training data by manually choosing output labels and then demonstrating input gestures using their input devices. Alternatively, users can provide a playback score that details how output labels change over time, and then play their instrument along with the score to generate labeled training data. After providing training data and training the learning system (a set of models, each model driving a different output parameter), the user can evaluate the system through real-time control of sound.

Workflow in the Wekinator is shown in the figure below. Notice that refining the learning system is an iterative process; users configure and train the learning system, evaluate the learning system, and then reconfigure the learning system based on feedback from the evaluation. While the Wekinator's incorporation of interactive supervised learning into their workflow is quite novel, this iterative design process—evaluating a system and making changes to that system—is common among creative domains [Resnick et al., 2005].

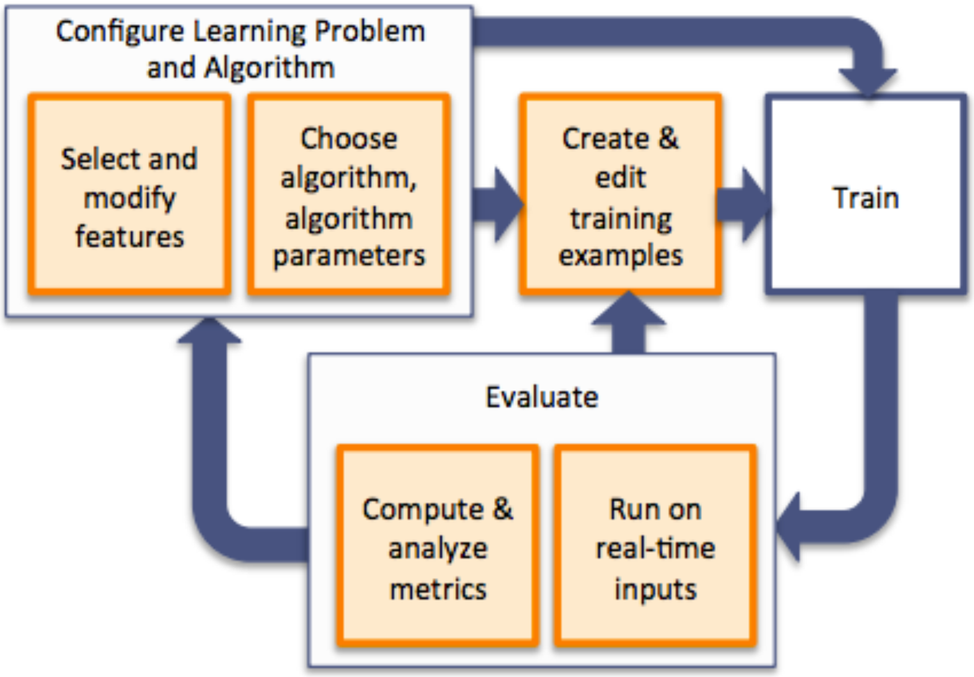

Configure Learning Problem and Algorithm
Select and modify features
Choose algorithm, algorithm parameters
Create & edit training examples
Train
Evaluate
Compute & analyze metrics
Run on real-time inputs

Figure 1 (taken from [Fiebrink, 2011]): Workflow in the Wekinator

The Wekinator workflow encourages the development of a single train of thought; allowing for easy navigation between modes for providing training examples and evaluation supports the iterative design process.

However, the Wekinator workflow does not address two important problems in creative support tools: (1) A designer may not have a good idea of how input should map to output and (2) A designer may have many competing ideas about how input should map to output, and might want to explore these ideas side by side.

Requiring the user to provide output labels for each feature vector should be seen as an unnecessary restriction on the design process. It forces the user to begin the design process by imagining an alleged relationship between the input feature space and the output feature space. For designers creating completely new instruments, this relationship might not be clear.

The Wekinator supports developing multiple mappings by allowing users to load and save instruments to files. This limited support prohibits users from editing multiple mappings or evaluating multiple mappings in quick succession. Users are unlikely to explore multiple mappings since no part of the interface suggests that they should explore multiple mappings. Additionally, in Wekinator, if you change the input feature setup, you have to create a completely new training data set. The extra effort necessary to recreate the data set discourages users from exploring different input feature setups.

To address these problems, we explore alternative algorithms for generating mappings given less initial information. Our graphical user interface supports side-by-side creation, modification, and evaluation of these mappings.

System Overview

Our system can generate mappings given different amounts of information from the user. This allows users to both rapidly develop exploratory mappings and carefully develop performance-ready mappings. To make our system as extensible as possible, we developed data structures that generalize mapping generation.

In our graphical user interface, we use "instrument" to refer to our mapping, as a term more identifiable to musicians. In order to compute output parameters, we assign each output parameter a "tunnel." A tunnel is a data structure comprised of five other data structures: A feature mapper, preprocessors, a function, a function family, and a postprocessor. A feature mapper represents the subset of the input features that should affect computation for the current tunnel. The preprocessors are a set of functions that do any necessary initial processing on the input features (examples might be smoothing functions or type conversions). The function is the main computational unit that computes an output based on the inputs. The function comes from a family of related functions, all of which can do computations on the same type of inputs and output. The function family might correspond to a machine learning algorithm and require training data in order to "instantiate" a function (a function might correspond to a machine learning model), or the function family might not need training data in order to instantiate a function. The postprocessor does any necessary processing on the output to produce the final output parameter.

To "run" an instrument in real time, the system receives Open Sound Control (OSC) packets with addresses that correspond to each input feature and arguments that correspond to input values. The input values get "tunneled" through each tunnel: filtered by the feature mapper, processed by preprocessors, seen as inputs to the function which produces an output, and finally processed by the postprocessor to achieve the final output parameter. Once the output parameters are computed, they are sent as OSC packets to the computer and address space specified by the user.

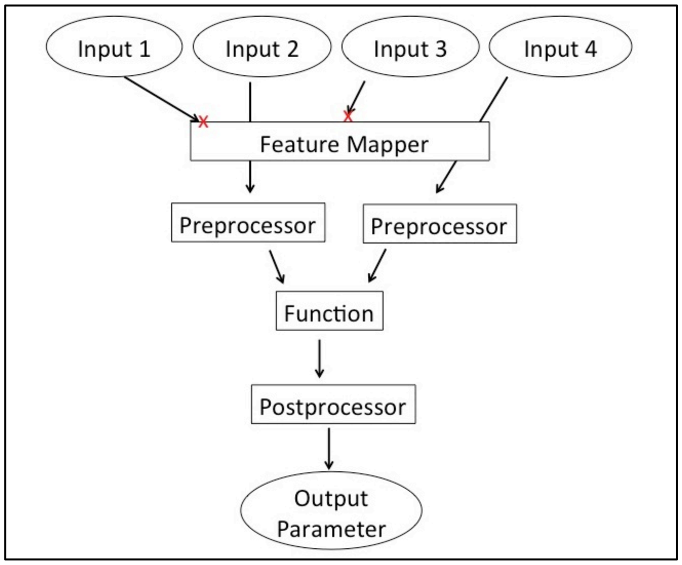


Figure 2: Flow of computation through a tunnel

Methods For Generating Mappings

When generating mappings, the system, rather than the user, is responsible for choosing feature mappers and function families for each tunnel. The system's choices require some randomness in order to ensure that different mappings can be generated and explored without changing the input/output setup or other initial conditions. The user can later go back and edit each tunnel's feature mapper, preprocessors, function family, function, or postprocessor.

In order to generate mappings, we pay attention to metadata entered by the user about inputs and outputs. Each input/output is considered to be continuous, discrete, or binary. All of these choices contain implications about what kind of functions can be used to compute output. The user can also choose whether to provide labeled, unlabeled, or no examples. This further reduces the amount of possible functions that can produce each output label. Our original heuristic for choosing functions will first find the subset of all possible functions in the system that can produce output based on these restrictions, and then chooses randomly between these functions. There is room for further research about (1) which functions will be most useful to users and (2) what smarter methods can we use to make choices about these functions.

GUI Support for Side-By-Side Creation of Mappings

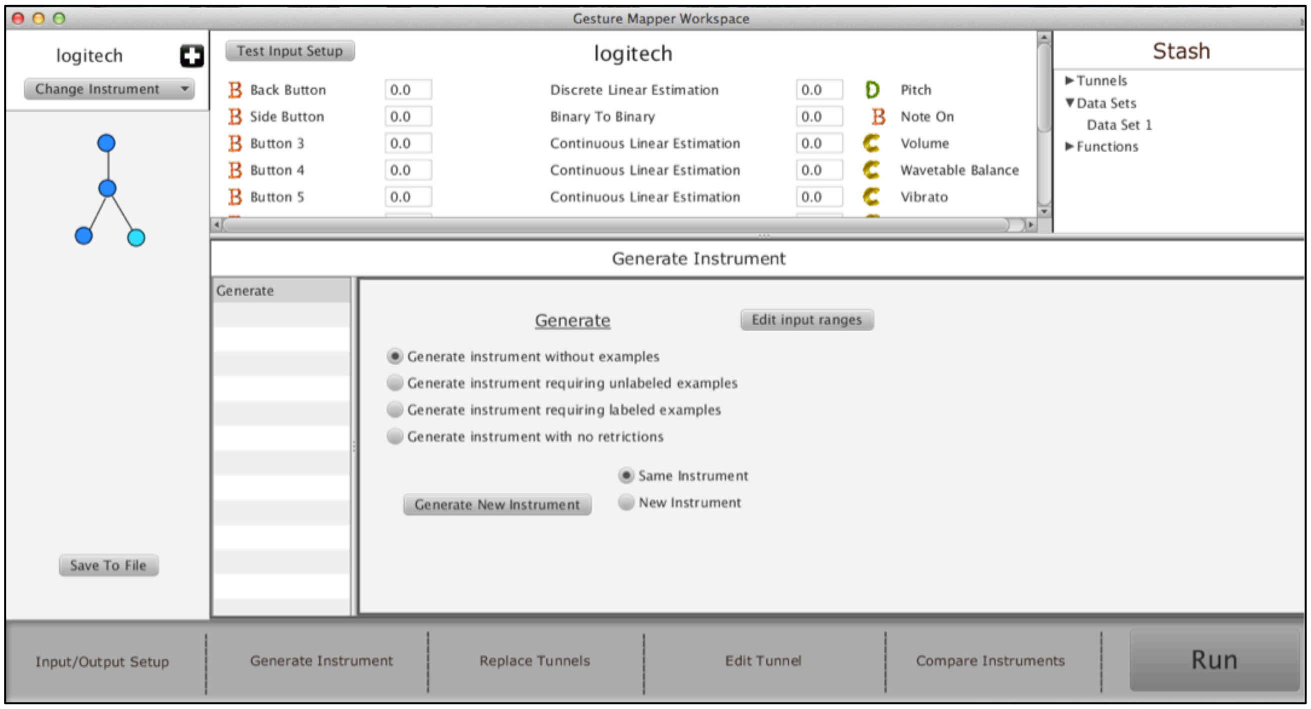
1. The Workspace Setup

Our interface is set up as a workspace that contains multiple instruments. At any time, users can switch between instruments via a combo box on the left side of the window, allowing instruments to be created side by side.

2. Generate Instrument

To generate an instrument, the computer chooses each tunnel's feature mapper and function family. The user prompts the computer to choose function families that require labeled examples, unlabeled examples, or no examples. If "no examples" was selected, a function can be instantiated from the function family automatically, and the user can run the instrument immediately. Otherwise, users must provide training data and instantiate functions before they can run the instrument.

Figure 3: The tool on the Generate Instrument screen. A digital prototype of the history tree is shown on the left, with the current node shown in light blue. A digital prototype of the stash is shown in the top-right corner of the window.



B Back Button	0.0	Discrete Linear Estimation	0.0	D	Pitch
B Side Button	0.0	Binary To Binary	0.0	B	Note On
B Button 3	0.0	Continuous Linear Estimation	0.0	C	Volume
B Button 4	0.0	Continuous Linear Estimation	0.0	C	Wavetable Balance
B Button 5	0.0	Continuous Linear Estimation	0.0	C	Vibrato

[✓ Generate instrument without examples]
Generate instrument requiring unlabeled examples
Generate instrument requiring labeled examples
Generate instrument with no retrictions

[✓ Same Instrument]
New Instrument

Users can also "replace tunnels," a mechanism identical to generating an instrument except the user chooses a subset of tunnels to generate.

By delegating the choice of feature mappers and function families to the computer, we allow the user to begin exploring the input feature space and output parameter space without forcing the user to find a relationship between the spaces. By choosing "no examples", the user can create an instrument with the click of a button. Since the computer's choice is pseudo-random, the user can generate different mappings simply with successive clicks of the same button. This is a quick way for a user without preconceived notions of how the input feature space should map to the output parameter space to explore many different mappings.

3. Stash

The stash is a global space where functions, data sets, and tunnels can be saved and copied into other instruments. Knowing that they can save valuable parts of an instrument, users will be able to go on to (1) modify that instrument without worrying how the valuable parts were affected, and (2) create "composite" instruments, where some parts of the instrument came from other instruments.

4. History Tree

The history tree will save all previous states of an instrument, each state at a different node in the tree. When the user makes any changes to an instrument, a new branch in the tree will be automatically created. The user will be able to access any node in the tree at any time. This has two advantages. First, similar to the stash, it allows users to explore any new ideas they have without worrying about losing their current progress. Secondly, if users have many paths they want to explore from the same node, they can simply explore one path, travel back to that node, and then explore the new path. The result will be two leaf nodes that the user will be able to compare side by side in the Compare Instruments screen.

Interface for Comparing Mappings

The Compare Instruments screen is a screen where users can select multiple instruments to run in quick succession or simultaneously. By allowing users to run instruments in quick succession, we give users a convenient way to compare different instruments.

By allowing users to play instruments simultaneously, we encourage them to design for ensembles. We also encourage "composite" instruments of a different sort, where the true instrument is made up of many "instruments" within the system.

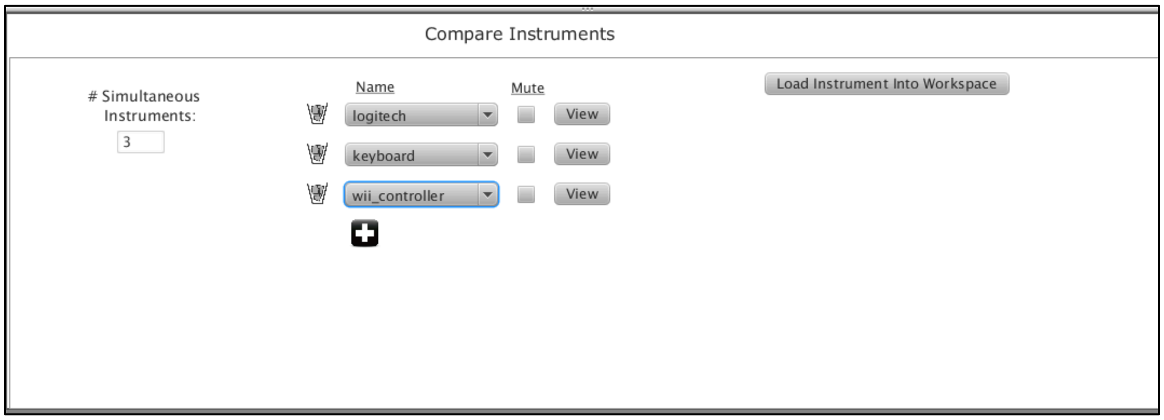


Figure 4: The Compare Instruments screen

Training Data Manipulation

Each function family has its own training data set, and a tunnel's function is instantiated based on the function family's training data (the training data may be ignored if the function family doesn't take examples). This design allows users to remove outputs without affecting other tunnels. It also allows users to add inputs and outputs to an instrument without affecting existing tunnels.

Further work should investigate how preprocessing and postprocessing can be used to maintain data sets past changes to input and output metadata. For example, suppose there exists a tunnel driving computation for a continuous output. If the user decides that output should be discrete, then postprocessing can be used to convert the function's continuous output to a discrete output. Without this conversion, the user would have had to generate the tunnel again, this time with a function that produces a discrete output. If the tunnel required examples, the user would have also had to provide new examples to train the tunnel.

Conclusion

We explore techniques for creating mappings from data in ways that are more appropriate and/or faster than supervised learning or manual coding. Our methods for creating mappings allow users to generate mappings either with or without providing examples. Our interface facilitates rapid and parallel mapping exploration.

We designed new methods for generating mappings given less initial information. We hope that by delegating many decisions about mapping generation to the computer (while still giving the user the option to override these decisions), designers will be able to better explore the input and output spaces.

We implemented a graphical user interface that allows users to automatically generate mappings side by side, edit the mappings, and compare them. Our hope is that tunnels, feature mappers, function families, and functions help users understand how inputs are mapped to outputs, while preprocessors, the stash, and the history tree will give the user enough safety and confidence to explore alternate mappings without the risk of losing their current progress.

Future work will incorporate preprocessors and postprocessors (which can help maintain data sets between changes in input/output spaces), the stash, and the history tree into the existing graphic user interface. There is also much potential for improving the system by investigating which functions best map inputs to outputs and by investigating smarter heuristics for automatically generating mappings. Finally, further studies will run user evaluations of the system.

Acknowledgements

Thanks to Assistant Professor Rebecca Fiebrink for mentoring me throughout my research experience, this work would not be possible without you. Thanks to graduate student KatieAnna Wolf for useful ideas, suggestions, and feedback on the system. Thanks to the Princeton Computer Science Department and the DREU organization for this opportunity.

References

Fiebrink, R. “Real-time human interaction with supervised learning algorithms for music composition and performance.” PhD thesis, Princeton University, 2011.

Resnick, M., Myers, B., Nakakoji, K., Shneiderman, B., Pausch, R., Selker, T., and Eisenberg, M. “Design principles for tools to support creative thinking.” In Report of Workshop on Creativity Support Tools, 2005.