AIMC 2023

NeuralMidiFx: A Wrapper Template for Deploying Neural Networks as VST3 Plugins

Behzad Haki Julian Lenz Sergi Jorda

URL: https://aimc2023.pubpub.org/pub/givwzz98

ABSTRACT

Proper research, development and evaluation of AI-based generative systems of music that focus on performance or composition require active user-system interactions. To include a diverse group of users that can properly engage with a given system, researchers should provide easy access to their developed systems. Given that many users (i.e. musicians) are non-technical to the field of AI and the development frameworks involved, the researchers should aim to make their systems accessible within the environments commonly used in production/composition workflows (e.g. in the form of plugins hosted in digital audio workstations). Unfortunately, deploying generative systems in this manner is highly expensive. As such, researchers with limited resources are often unable to provide easy access to their works, and subsequently, are not able to properly evaluate and encourage active engagement with their systems. Facing these limitations, we have been working on a solution that allows for easy, effective and accessible deployment of generative systems. To this end, we propose a wrapper/template called **NeuralMidiFx**, which streamlines the deployment of neural network based symbolic music generation systems as VST3 plugins. The proposed wrapper is intended to allow researchers to develop plugins with ease while requiring minimal familiarity with plugin development.

1. Introduction

There has been, in recent years, a rapid surge of neural network models intended for content generation in a variety of domains. In the past two years specifically, the advancement of the field has been so prominent that many of these models, previously intended primarily for the research community, have now been increasingly adopted by the public. As such, an ever-growing number of people are becoming more engaged in discussions about these technologies.

Similar to other domains, research and development of AI-based music generation tools should actively invite users outside of the research community to participate. This involvement can take many forms, including subjective evaluation of model outputs, or in cases such as compositional assistant tools, direct engagement in the creation process. In either regard, to encourage a broad and diverse set of viewpoints, developed systems should be deployed in a manner that non-expert users with minimal familiarity with the technical tools involved can easily access the models to subsequently interact with them.

It should be noted that our work is specially intended for the deployment of models during research development stages (as opposed to commercial development). Our aim is to provide researchers of generative models, with efficient deployment tools that will make these models more easily accessible to musicians to test and interact with them.

1.1 Deployment Frameworks

Deploying systems in an easily accessible format is an extremely costly and time-consuming process that requires a great deal of domain-specific knowledge of frameworks which is often outside of the expertise of the researchers. Consequently, researchers with limited resources typically have to deploy their models in more efficient ways; for instance, allowing access to models through a unique API usable through the command-line, or cloud-based interactive computing platforms like Google Colab [1][2][3][4]. If resources are limited, these methods are very valuable as they still allow users to access the developed systems. However, they can have a limiting effect on engagement as some level of technical familiarity with the deployment environment is required from the users. Moreover, the impractical user-interfaces available in these environments may discourage users to persistently employ these tools in their creative workflows.

An alternative method of deployment that allows for easy access without technical familiarity is through an interactive webpage. To name a few, folk-rnn [5], MuseNet [6], Latent Space Exploration [7] and Magenta.js [8] invite users to interact via a dedicated web-page. While this method of deployment is certainly more accessible and user-friendly compared to the previous methods, such implementations require significant development time, and are still largely removed from the software environments that musicians typically utilise in their creative workflows.

Tools such as RhythmVAE [9], Regroove , DeepBach [10], DrumNet [11], Anticipation-RNN [12] and Magenta Studio [13] have deployed generative models either directly as standalone applications or as plugins hosted in audio production environments such as Digital Audio Workstations (DAW), MaxMSP [14], PureData [15] and Musescore. The benefit of this approach is that the developed systems can be easily employed in production environments with which composers/producers are most familiar. However, these bespoke software packages are written to host a specific model and would require extensive re-programming to host new research models.

Lastly, in the context of audio-to-audio transformations, there are a number of applications that allow for running neural networks wrapped as host-specific plugins (provided they meet certain technical criteria). For instance, nn~, allows for running certain generative models as MaxMSP or PureData extensions. Moreover, Neutone can wrap a variety of deep learning models as Virtual Studio Technology (VST) [16] plugins to be hosted in DAWs. As such, these applications allow for easier integration of neural audio models in production friendly environments. We recognize the positive contribution these tools have on future research, but are presently unaware of any equivalent technology designed for symbolic music generation tasks.

1.2 Motivation

Within the context of music production applications, one of the most universal frameworks is VST. VST applications are highly popular as they can be used as plugins within DAWs, allowing users to seamlessly load a variety of third-party tools within their music production environment. While from the user point of view these plugins are convenient to access, from the researchers' perspective, they require a great deal of time and expertise to implement.

For the past two years, we, as researchers working on performance-oriented symbolic music generation tasks, have provided a number of prototypes of our work to musicians so as to allow them to interact with and evaluate our systems prior to developing subsequent revisions. These prototypes generally consisted of a graphical front-end communicating with a separate python back-end accessible through the command-line [17] [18]. Throughout this process, we realized that (1) many users had difficulties in running the prototypes and (2) these difficulties were often preventing persistent inclusion in our testers' production workflows. We concluded that, in order to encourage reliable interactions with the system, we must package and deploy our models in a manner that allowed the users to quickly setup the software and run it whenever inspiration struck. As a result, we decided to re-develop our systems as VST3 plugins using Jules' Utility Class Extensions (JUCE) [19] framework.

Given that we did not have any prior experience with this deployment approach, we spent many months and resources at the expense of conducting research on our main fields of interest. It became apparent that other researchers in this domain must choose between going through similarly time-consuming processes, or accepting a limited level of engagement from users. As such, we realized that the community at large can benefit from our findings, which we have used to develop a general template that allows for deploying symbolic generative neural network models of music. In this paper, we introduce the plugin template, its capabilities and architecture, and finally discuss some of its potential use-cases and its current limitations.

2. NeuralMidiFx

NeuralMidiFX is a VST3 wrapper template based on the JUCE framework [19] for deploying generative models of symbolic music. The intention behind this plugin is to facilitate the implementation of generative models of symbolic music in an easy-to-access plugin format whilst streamlining a number of technically challenging aspects of the deployment process, such as (1) multi-threaded design, (2) thread-safe data communication, (3) parameter control, and (4) interface design. In section 2.1, we start with a brief overview of the challenges involved in designing NeuralMidiFx, and subsequently, in section 2.2, we discuss the design decisions made for developing the wrapper based on the involved challenges.

The following video provides a brief overview of this section.

Visit the web version of this article to view interactive content.

Video 1. Challenges of Deploying Symbolic NN-based Models of Music As VST Plugins

2.1 Design Challenges and Considerations

The VST framework, and by extension JUCE, poses a number of limitations to be considered when the aim is to deploy neural network models of symbolic music that are often computationally intensive. Basic JUCE plugins consist of two main threads: (1) **AudioProcessor (AP)** and (2) **AudioProcessorEditor (APE)**. The AP thread is in charge of two-way communication with the plugin host, while the APE thread is in charge of the Graphical User Interface (GUI). In this framework, the AP thread is constructed once and is always active, while the APE thread is constructed only when the plugin's GUI is in ‘view’ mode. In this section, we discuss the challenges and requirements for developing a VST3 plugin from model deployment and user interface generation perspectives.

2.1.1 Multi-Threaded Design

The communication between a VST3 plugin and a compatible host is done on a per-buffer basis. In JUCE, a method (called **processBlock**) exists that is automatically called upon arrival of each new block of data. This method is used for receiving information about the state of the plugin host[1], receiving incoming audio/MIDI data, and also sending outgoing audio/MIDI data. The lifespan of this method is limited to the lifespan of a single buffer provided by the host[2], i.e. only limited amount of time is available to carry out computations within this method. As a result, intensive computations (such as generation using a neural network) must be carried out in the background using a separate thread. Multi-threaded design can be extremely complicated to develop and debug, hence, this design requirement is one of the major hurdles in the deployment process.

The multi-threaded design requires a mechanism to communicate data between the threads. An important consideration to make with regards to inter-thread communication is that the data communication needs to be both thread-safe and also lock-free: (1) thread-safe mechanisms ensure that data racing does not occur between the threads, and (2) the lock-free implementation ensures a thread accessing a shared resource will not potentially lock (or temporarily halt) the operations in another thread requiring access to the same resource (specifically important in the case of the *AudioProcessor* thread). Moreover, certain communication mechanisms should queue the incoming data in case the consumption of data is carried out at a lower rate. The effort involved in developing safe and flexible communication mechanisms that meet all these requirements can highly slow down the deployment process, specially when dealing with various data-types to be communicated.

2.1.2 User Interface

The interaction between a user and a model is generally done through communicated MIDI messages and/or through a selection of parameters that are controllable via a graphical interface. The selection of controllable parameters is quite dependent on the target task. Once these selections are made, plugin development frameworks like JUCE provide many valuable tools for creating graphical interfaces for the selected parameters. That said, creating even basic interactive components that are stable, automatable through the host, and easily and safely accessible from all non-visual processes of the plugin, requires a great deal of familiarity with the development framework.

Moreover, in the context intended for this work, the organization, aesthetics and the selection of the interactive elements are very task dependent and may require high levels of customization. If the aim of the wrapper is to significantly lower the cost for researchers to deploy their models as plugins, there should be a balance between ease-of-use[3] and customization. As such, the wrapper should provide tools that allow the researcher to easily specify the graphical elements needed (with some limitations) while also allowing for some level of customizable organization.

2.2 Design Decisions

The wrapper was to be developed for researchers that (1) do not have any prior experience with plugin development and (2) require support for a variety of generative tasks. To accommodate this second point, we concluded that any process that may be model specific should be the responsibility of the user of the template, while any other aspect of the development should be fully streamlined by NeuralMidiFx. In this context, a number of tasks are clearly the responsibility of the wrapper as they are model independent: handling the reception of incoming information from the host, streaming of generated content, creating globally accessible parameters, and rendering a graphical interface.

Additionally, model-dependent tasks can be divided into three stages: (1) **Input Preparation:** converting a sequence of musical symbols (MIDI) and/or control parameters into a format required by a model, then (2) **Model Inference:** passing the prepared input through the model for inference, and finally (3) **Playback Preparation:** reformatting the model output back into MIDI to be played back by the host. These three stages happen sequentially, however, they do not necessarily occur at the same rate (for example, a model may generate a given output for every number of input MIDI messages received). Moreover, the computational resources required for each stage can vary significantly. As a result, ideally these three stages should be deployed in separate threads and communicate information between one another only when needed. While the processes in these threads are model-specific (hence, responsibility of the user), the setup and initialization of the threads as well as the inter-thread communication pipelines are independent of the model, and as a result, NeuralMidiFx should be responsible for them.

Figure 1 illustrates the division of tasks between the researcher and the wrapper.

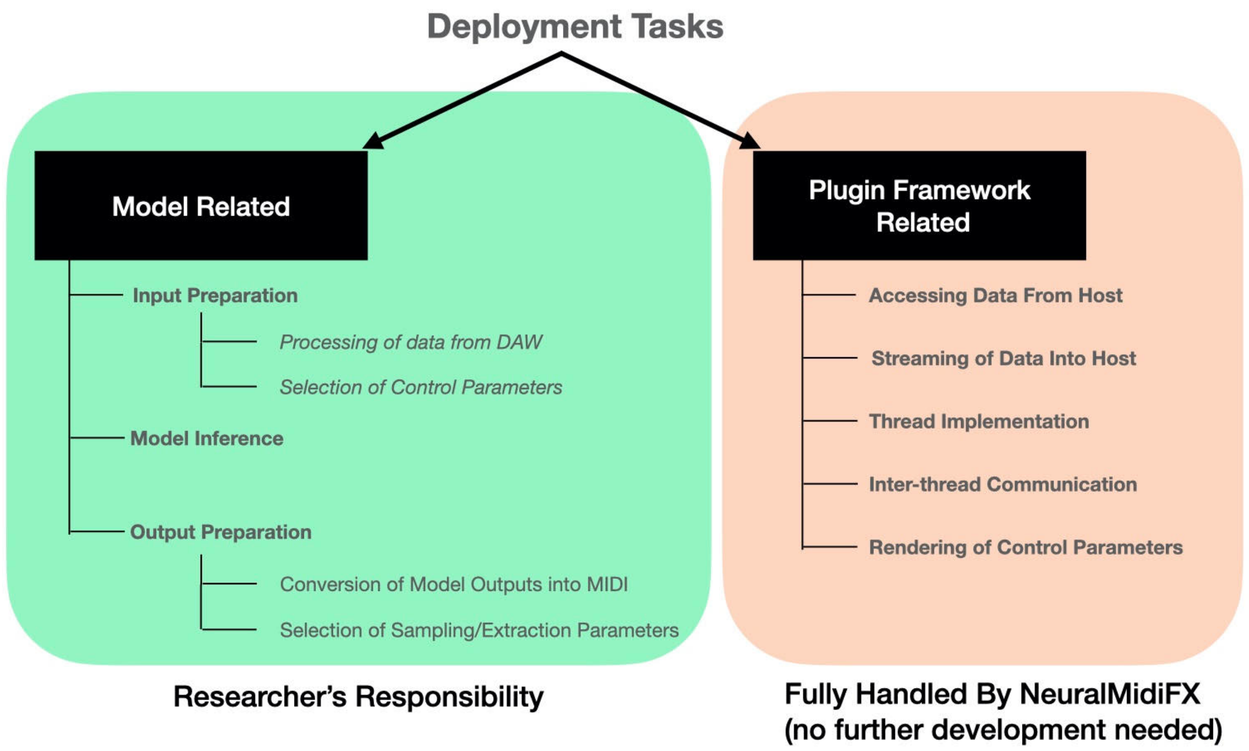
Deployment Tasks
Model Related
Input Preparation
Processing of data from DAW
Selection of Control Parameters
Model Inference
Output Preparation
Conversion of Model Outputs into MIDI
Selection of Sampling/Extraction Parameters
Plugin Framework Related
Accessing Data From Host
Streaming of Data Into Host
Thread Implementation
Inter-thread Communication
Rendering of Control Parameters
Researcher's Responsibility
Fully Handled By NeuralMidiFX (no further development needed)

Figure 1. Division of tasks in the deployment process.
The proposed wrapper, NeuralMidiFx, fully automates the deployment processes that are specific to plugin development frameworks. However, tasks related to the model are not fully automated as they are highly dependent on the specific task at hand. Nonetheless, knowing that the stages involved in the inference process can be distinctly divided into three separate procedures (Input Preparation, Model Inference, and Output Preparation/Post-Processing), NeuralMidiFx provides dedicated threads for each of these procedures, with specific access points made available to the researcher. This allows for greater control and customization of the model-related tasks, while still providing an easy and streamlined process for plugin development.

In the next section, we discuss how the architecture of NeuralMidiFx accommodates the design decisions made.

3. Architecture

NeuralMidiFx consists of six main threads. Three of these threads do not require any modification (section 3.1) while the developer preparing the plugin is responsible for adapting the other three threads (called **Deployment Threads**) for their intended application (section 3.2). Lastly, to generate a graphical interface, the developer only needs to provide a list of parameters with their corresponding required attributes (section 3.3). The wrapper can automatically render the user interface for these parameters while also providing access to their values across all major threads. Figure 2 illustrates an overview of the architecture of NeuralMidiFx.

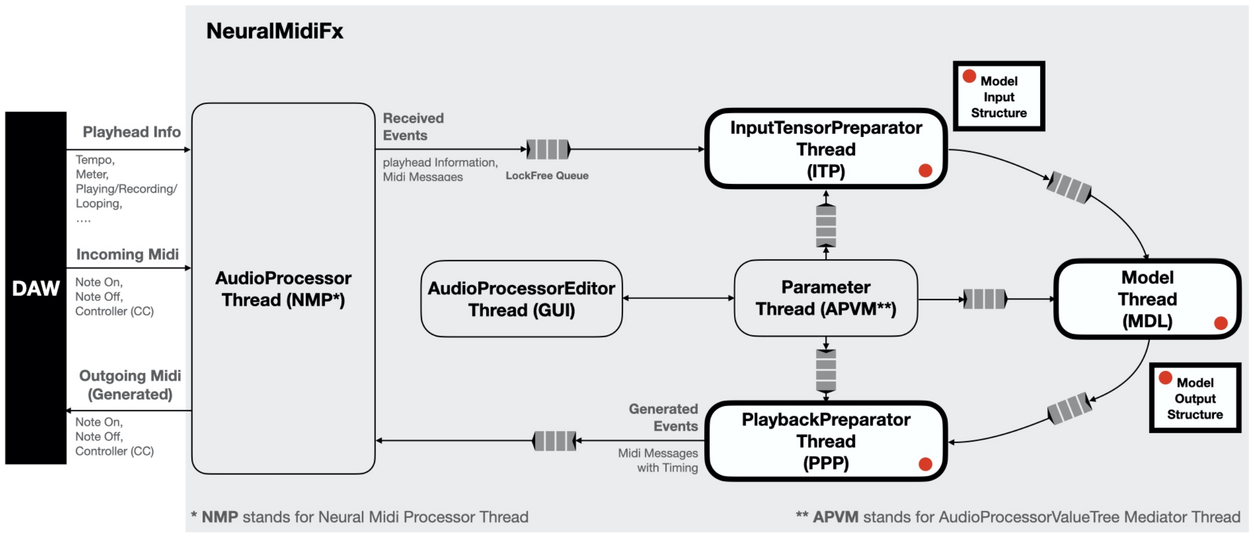
NeuralMidiFx
DAW
Playhead Info
Tempo, Meter, Playing/Recording/ Looping, ...
Incoming Midi
Note On, Note Off, Controller (CC)
Outgoing Midi (Generated)
Note On, Note Off, Controller (CC)
AudioProcessor Thread (NMP*)
Received Events
playhead Information, Midi Messages
LockFree Queue
InputTensorPreparator Thread (ITP)
Model Input Structure
AudioProcessorEditor Thread (GUI)
Parameter Thread (APVM**)
Model Thread (MDL)
Model Output Structure
PlaybackPreparator Thread (PPP)
Generated Events
Midi Messages with Timing
* NMP stands for Neural Midi Processor Thread
** APVM stands for AudioProcessorValueTree Mediator Thread

Figure 2. Architecture of the plugin - white boxes tagged with a red dot require additional implementation by the developer while all other threads/communication means are readily available and don't need any interaction from the developer.

3.1 Communication with the Plugin Host

As discussed in section 2.1, the AudioProcessorThread for NeuralMidiFx (called NMP) is in charge of receiving information from the host, and if required, also provide playback data in return (on a per-frame basis).

3.1.1 Incoming Data

In the context envisioned for NeuralMidiFx, the host may provide two types of data for each buffer:

1. **Play-head Metadata**: Information about the buffer location and attributes such start time, tempo, meter, sample rate, buffer size, playing/recording/looping status, and loop points of the host,
2. **MIDI Messages**: Note On, Note Off and/or Controller messages within the buffer

While the JUCE framework provides all necessary utilities to access this information from the host (if available[4]), depending on the task intended, some of this data requires further processing.[5] Moreover, there are different timing conventions that can be used for timed information: (1) seconds, (2) quarter notes, and (3) audio samples.

The NMP thread internally conducts all necessary processing of the Play-head Metadata and MIDI Messages provided by the host, and wraps each one in a custom datatype (called **Event**), and finally, sends them to the next thread to be used for preparation of the model's input(s). Figure 3 demonstrates the different situations for which Events can be generated.

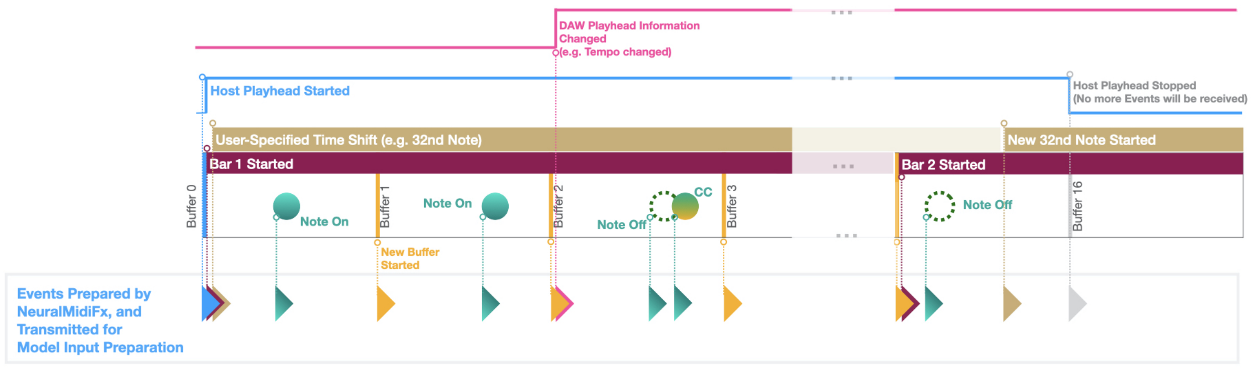
DAW Playhead Information Changed (e.g. Tempo changed)
Host Playhead Started
Host Playhead Stopped (No more Events will be received)
User-Specified Time Shift (e.g. 32nd Note)
New 32nd Note Started
Bar 1 Started
Bar 2 Started
Buffer 0
Buffer 1
Buffer 2
Buffer 3
Buffer 16
Note On
Note On
Note Off
CC
Note Off
New Buffer Started
Events Prepared by NeuralMidiFx, and Transmitted for Model Input Preparation

Figure 3. NeuralMidiFx pre-processes and sequentially transmits all necessary **Events** received from the host. The messages are transmitted sequentially in the order of occurrence, and are time-stamped in different units (seconds, quarter notes, and samples). Users can specify which Events should be transmitted, for example, if Bar locations or Tempo information are not used for tokenization, there is no need to be notified about these Events).

3.1.2 Outgoing Data

Any audio/MIDI data to be played back by the host is also provided on a per-buffer basis within the same thread from which the incoming data is received. Moreover, the plugin can only provide MIDI messages for playback if the timing of messages overlap with the buffer. As a result, to playback a stream of previously generated MIDI messages, the NMP thread must constantly find valid messages overlapping with a given buffer, and subsequently, release them to the host using an accepted timing convention.[6] Complying with all these requirements, NeuralMidiFx automatically manages the streaming of any generated MIDI messages with any timing convention preferred by the developer.

3.2 Deployment Threads

NeuralMidiFx dedicates three threads respectively for preparing the inputs of a model (section 3.2.1), running inference (section 3.2.2), and reformatting the generations into MIDI messages for playback via the host (section 3.2.3). In these threads, the wrapper provides a set of utilities for easily receiving and sending information from/to the threads via the previously implemented inter-thread communication pipelines (section 3.2.4).

3.2.1 InputTensorPreparator Thread (ITP)

Prior to running the inference on a given model, all or some of the host's play-head information, incoming MIDI messages, and possibly parameters controlled via a graphical interface must be reformatted according to a given model's input requirements. In this thread, all necessary information is sequentially provided to carry out the tokenization (or representation) of the relevant symbolic information into a tensor-like format.

The method of tokenization can vary greatly, with each task having different priorities and contextual requirements. For example, a model focused on composing melodies with structural awareness such as the Jazz

Transformer [20] requires encapsulation of chord and phrase events, whereas a drum generation model such as GrooVAE [21] needs information on rhythmic micro-timings. There are several popular tokenization formats, such as REMI [4], Compound Word [22], and Octuple [23], which are typically re-used with minor modifications for subsequent research, but there is no single universal standard.[7]

Whilst it is not possible - nor efficient - to provide support for every type of tokenization in a single VST plugin, most methods rely upon a similar subset of message types: note (note on/off, pitch, velocity), timing (duration, time-shift, bar position), and host information (time signature, tempo, playhead status). As shown in Figure 3, NeuralMidiFX provides all of these messages in an easily-accessible format, enabling researchers to quickly access relevant bytes of data during inference and process them accordingly (see Algorithm 1).

Algorithm 1: Main Processing in ITP Thread

```
1  while thread should run do
2      while input queue (from NMP) is not empty or
         control params changed do
3          if control params changed then
4              get latest values
5          end
6          if input queue (from NMP) is not empty then
7              pop new Event if Event notifying about
                 state of host then
8                  Update tokenized Sequence
                     accordingly
9              end
10             if Event notifying about Midi Message
                 then
11                 Update tokenized Sequence
                     accordingly
12             end
13         end
14         if should send new tokenized sequence then
15             wrap data in ModelInput structure
16             push ModelInput to output queue
17         end
18     end
19 end
```

3.2.2 Model Thread (MDL)

Whenever a new input sequence is to be passed through a model for inference, the ITP thread sends the prepared input sequence to the model thread via a pre-implemented pipeline between the two threads. Upon arrival of the new input sequence, the received sequence is passed through the model to come up with a generation. Subsequently, the generations are passed on to the next thread responsible for extracting the note events from the generations (see Algorithm 2).

Algorithm 2: Main Processing in MDL Thread

```
1 while thread should run do
2     while input queue is not empty do
3         pop new ModelInput
4         pass to model for inference
5         wrap outputs in ModelOutput structure
6         push to output queue
7     end
8 end
```

3.2.3 PlaybackPreparator Thread (PPP)

Once a new generation is received from the model thread, the PPP thread extracts note on/off or controller events from the generations. Using the implementations provided by the developer, the corresponding timing of each of the extracted events should then be calculated, and subsequently, the extracted events with their associated timings (in any unit) should be sent over to the NMP thread for playback.

To play generations properly, before receiving a sequence of generations, the wrapper requires the user to specify the playback policy that should be assumed. Playback policies refer to (1) whether new generations overwrite previous ones, and (2) whether the timing of generations are relative to absolute zero, relative to playback position when host started, or relative to the time the policy specification is received by the host (see Algorithm 3).

Algorithm 3: Main Processing in PPP Thread

```
1  while thread should run do
2      if input queue is not empty then
3          pop new ModelOutput from input queue
4          if not a continuation of older gens then
5              Prepare a playback policy
6              push policy to output queue
7          end
8      end
9      if ModelOutput is not fully processed then
10         extract generated events
11         sequentially push to output queue
12     end
13 end
```

3.2.4 Thread-Safe Data Communication

One important feature of NeuralMidiFx is that all inter-thread communication pipelines are already developed and are ready to use. Moreover, the implementation is such that we ensure that the data is safely provided sequentially from one thread to next via lock-free queues. The datatype communicated between most of the threads is already defined and does not require any modification from the developer. However, given that the input/output interface of a model is only known to the developer, the input/output structures communicated both between ITP and MDL threads and MDL and PPP threads need to be adapted to include all required parameters.

3.3 Parameter Thread and GUI

To declare the parameters, developers only need to provide the wrapper with a custom structure specifying (1) the required parameters (each identifiable via a unique textual label), (2) the operable ranges and default values, and (3) specifying how the parameters should be visually rendered (i.e. what interactive elements should be used for each parameter and how each subset of parameters should be visually grouped together in separate tab). Below is an example of one such tuple.

```
{tab_tuple{ "Model",
    slider_list {
        slider_tuple {"Tension", 0, 1, 0.2},
        slider_tuple {"Velocity", 0, 25, 18.02}},
    rotary_list {
        rotary_tuple {"Density", 0, 1, 0.59},
        rotary_tuple {"Timing", 0, 4, 1.06},
        rotary_tuple {"Range", 0, 1, 0.43}},
    button_list {
        button_tuple {"Quantize", true},
        button_tuple {"Mute", false}} },
tab_tuple{ "Settings",
    button_list {button_tuple {"Record", true}}}}
```

Figure 4 shows the rendered interface for the parameters specified above. Note that all rendered interfaces are manually re-sizable in run-time to allow proper display on different screen sizes.

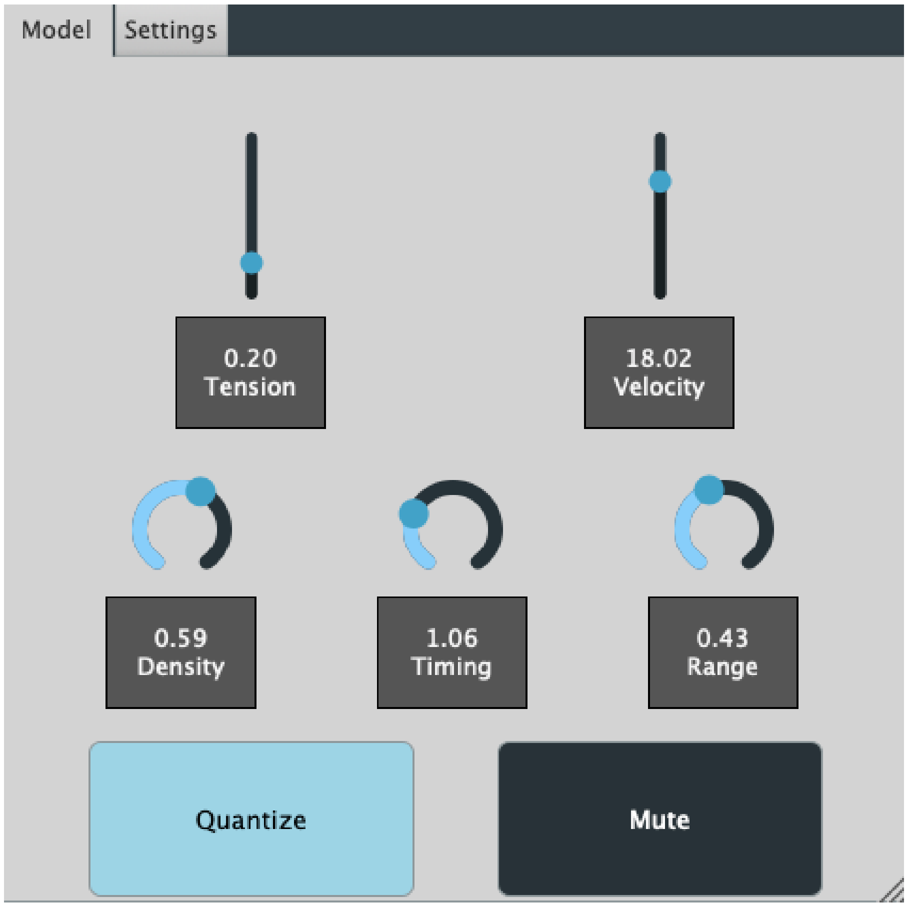

Figure 4. An example of automatically generated plugin interface with two tabs. Figure is showing the first tab of the interface

In addition to the interface generation, NeuralMidiFx ensures that all these parameters are automatable from the host, and also the values of the parameters are always accessible in the deployment threads using only their textual identifiers. Automation and communication of the value of these parameters is carried out using an abstracted mechanism implemented in a dedicated thread called Parameter Thread (see Figure 2). This thread takes advantage of *JUCE*'s *AudioProcessorValueTreeState* utilities, to access the parameters associated with each graphical element and distribute them to all necessary threads using implemented safe inter-thread communication pipelines.

4. Discussion

With NeuralMidiFx wrapper, symbolic-to-symbolic generative systems of music, that are based on neural networks, can be deployed as VST3 plugins. The major benefit of this wrapper is that many stages of the deployment process that deal with VST3 framework-specific limitations/requirements are automated, hence, the wrapper significantly eases the technical and financial barriers that generally demotivate resource-limited researchers from deploying their models in this user-friendly format. In other words, the wrapper allows researchers to spend the majority of the deployment process on tasks with which they have most familiarity.

As discussed previously in section 2.1, deploying neural network based generative models in a plugin format requires a multi-threaded design. Developing multi-threaded applications with safe and reliable means of data communication is a very complicated, hard-to-debug, and time-consuming task. Likewise, implementation of plugin-host communication mechanisms and also designing a graphical interface are other aspects of the deployment process that require a great deal of time and technical expertise. To speed up the deployment process, NeuralMidiFx abstracts these processes from the developer allowing them to focus mainly on the model-related implementations.

As many processes in the template are static (i.e. are abstracted and are not modifiable), NeuralMidiFx provides the users with a limited number of access points to the template, clearly clarifying the role of the developers in the deployment process (as discussed in section 3.2). That said, these limited access points are flexible enough to accommodate symbolic generation tasks ranging from unconditional, uncontrollable and non-real-time generative tasks to conditional, controllable, real-time generative tasks [24].

While we believe that the existing architecture of the wrapper is flexible enough to accommodate any symbolic music generation system, the existing structure is only capable of deploying serialized ***PyTorch*** models [25]. As such, using other model types requires a conversion to this format. Moreover, at this point NeuralMidiFx only supports CPU inference, meaning that computationally intensive models may have poor performance if deployed using the template.

We are currently using this wrapper for deploying our own generative systems as VST3 plugins so as to evaluate them in production/creative settings. As such, we will be actively working on improving the provided template based on our personal experiences with the system. At this point, we have identified a number of features to be implemented in the near future. An overview of these features is provided in the next section.

5. Future Works

There are a number of improvements that can further simplify and optimize the deployment process:

1. provide support for generalized model formats such as **ONNX** [26]
2. provide support for GPU inference
3. provide a stand-alone version of the wrapper
4. provide better debugging tools such as (a) automatic visual piano-roll rendering of generated contents, (b) utilities for importing/exporting MIDI files by drag-in/drag-out interactions, (c) text editor panels for displaying messages

The last feature that can be highly valuable is automatic tokenization of incoming events to any of the familiar formats discussed in section 3.2.1.

6. Conclusions

NeuralMidiFx allows developers to easily and efficiently create VST3 plugins that employ neural network based generative models of symbolic music. The template divides the deployment process into two parts: (1) VST3 framework-related tasks, and (2) model-related tasks. In this template, the developer is only responsible for the tasks related to the model, while all development tasks related to VST3 framework such as implementation of a multi-threaded architecture, inter-thread communication, exchange of information between plugin and host, and graphical interface design and rendering are all automated by the template. This clear division of tasks in NeuralMidiFx allows researchers, even those with minimal knowledge of plugin development, to be able to deploy and share their systems in a user-friendly format. Lastly, the source code and a number of video tutorials[8] for NeuralMidiFx are publicly available at:

https://github.com/behzadhaki/NeuralMidiFXPlugin.git

We hope that this project encourages other researchers in the community to help us improve the design and implementation of the wrapper by either directly collaborating with us or by providing us with suggestions on what modifications to be made in the future.

Acknowledgments

This research was partly funded by the Ministry of Science and Innovation of the Spanish Government. Agencia Estatal de Investigación (AEI). (Reference: PID2019-111403GB-I00).

Ethics Statement

The main intention behind this work is to facilitate and encourage researchers in the field to make their works more accessible to the public in a user-friendly format that does not exclude non-technical users. This involvement of the stakeholders early in the design process may lead to more responsible, human-centric and ethical pursuits of research in the field of generative AI. Moreover, this work is also intended for inclusion of researchers with more limited resources in the discussions around AI and music generation, specifically, by allowing the researchers to promote, at a lower cost, their works to a wider audience.

Throughout this work, we were not in any situation that gave rise to a conflict of interest.

Footnotes

1. such as sample rate, buffer size, tempo, time signature, whether host is in play, loop and/or record modes, and so on. See docs.juce.com/master/classAudioPlayHead.html ↩
2. Lifespan here is less than the block size in seconds. e.g., for a 128-sample block size at a sample rate of 44.1kHz, the processBlock is operable for a maximum of 2.8ms ↩

3. Ease-of-use from the perspective of the developer (i.e. researcher in this case). Not to be confused with ease-of-use from the perspective of the plugin user (i.e. musicians) ↩

4. Availability of some of the buffer metadata is not guaranteed, that is, some hosts may not provide some of the metadata fields ↩

5. For instance, the start time of each buffer is provided in absolute time, while the start time of each MIDI event is provided relative to the beginning of the buffer ↩

6. In this case, the messages must be timed in terms of audio samples relative to the start time of a buffer ↩

7. The MidiTok library (https://github.com/Natooz/MidiTok) provides an excellent overview and implementation of several popular methods. ↩

8.

Video Tutorials

Tutorial 1: Introduction to NeuralMidiFx

https://www.youtube.com/watch?v=o_4NsttseDw

Tutorial 2: Parameters and GUI Generation

https://youtu.be/r3oBxg6RQmM

Tutorial 3: InputTensorPreparator Thread (ITP)

https://youtu.be/B2UUSiIU7Y0

Tutorial 4: Model Thread (MDL)

https://youtu.be/VnSaHGR_6JA

Tutorial 5: PlaybackPreparatorThread (PPP)

https://youtu.be/SMbSrPlxubM ↩

References

- Dorsey, B. (2017). *MIDI-RNN*. https://brangerbriz.com/blog/ using-machine-learning-to-create-new-melodies. https://brangerbriz.com/blog/ using-machine-learning-to-create-new-melodies ↩
- Ens, J., & Pasquier, P. (2020). MMM : Exploring Conditional Multi-Track Music Generation with the Transformer. *CoRR*, *abs/2008.06048*. https://arxiv.org/abs/2008.06048 ↩
- Gillick, J., Roberts, A., Engel, J. H., Eck, D., & Bamman, D. (2019). Learning to Groove with Inverse Sequence Transformations. In K. Chaudhuri & R. Salakhutdinov (Eds.), *Proceedings of the 36th*

International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Vol. 97, pp. 2269–2279). PMLR.↩

- Hadjeres, G., & Nielsen:, F. (2020). Anticipation-RNN: enforcing unary constraints in sequence generation, with application to interactive music generation. *Neural Comput. Appl.*, *32*(4), 995–1005. https://doi.org/10.1007/s00521-018-3868-4 ↩
- Hadjeres, G., Pachet, F., & Nielsen, F. (2017). DeepBach: a Steerable Model for Bach Chorales Generation. In D. Precup & Y. W. Teh (Eds.), *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017* (Vol. 70, pp. 1362–1371). PMLR. http://proceedings.mlr.press/v70/hadjeres17a.html ↩
- Haki, B., Nieto, M., Pelinski, T., & Jordà, S. (2022, September). Real-Time Drum Accompaniment Using Transformer Architecture. *Proceedings of the 3rd Conference on AI Music Creativity*. https://doi.org/10.5281/zenodo.7088343 ↩
- Hsiao, W.-Y., Liu, J.-Y., Yeh, Y.-C., & Yang, Y.-H. (2021). Compound Word Transformer: Learning to Compose Full-Song Music over Dynamic Directed Hypergraphs. *Proceedings of the AAAI Conference on Artificial Intelligence*. ↩
- Huang, Y.-S., & Yang, Y.-H. (2020). Pop Music Transformer: Beat-based modeling and generation of expressive Pop piano compositions. *Proceedings of the 28th ACM International Conference on Multimedia*. ↩
- Ji, S., Luo, J., & Yang, X. (2020). A Comprehensive Survey on Deep Music Generation: Multi-level Representations, Algorithms, Evaluations, and Future Directions. *CoRR*, *abs/2011.06801*. https://arxiv.org/abs/2011.06801 ↩
- JUCE. (2023). Jules' utility class extensions (Version 7). In *JUCE: Jules' Utility Class Extensions*. https://juce.com/. https://juce.com/ ↩
- Lattner, S., & Grachten, M. (2019). High-Level Control of Drum Track Generation Using Learned Patterns of Rhythmic Interaction. *CoRR*, *abs/1908.00948*. http://arxiv.org/abs/1908.00948 ↩
- Nuttall, T., Haki, B., & Jorda, S. (2023). Completing Audio Drum Loops with Symbolic Drum Suggestions. *NIME 2023*. ↩
- ONNX. (2022). Open Neural Network Exchange. In *ONNX*. https://onnx.ai/. https://onnx.ai/ ↩
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32* (pp. 8024–8035). Curran Associates, Inc. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf ↩
- Pati, A., & Lerch, A. (2021). Attribute-based regularization of latent spaces for variational auto-encoders. *Neural Comput. Appl.*, *33*(9), 4429–4444. https://doi.org/10.1007/s00521-020-05270-2 ↩
- Payne, C. (2019). *MuseNet*. OpenAI. ↩

- Puckette, M. S., & others. (1997). Pure data. *ICMC*. ↩
- Puckette, M., Zicarelli, D., & others. (1990). Max/msp. *Cycling*, *74*, 1990–2006. https://cycling74.com/products/max ↩
- Roberts, A., Engel, J., Mann, Y., Gillick, J., Kayacik, C., Nørly, S., Dinculescu, M., Radebaugh, C., Hawthorne, C., & Eck, D. (2019). Magenta Studio: Augmenting Creativity with Deep Learning in Ableton Live. *Proceedings of the International Workshop on Musical Metacreation (MUME)*. http://musicalmetacreation.org/buddydrive/file/mume_2019_paper_2/ ↩
- Smilkov, D., Thorat, N., Assogba, Y., Nicholson, C., Kreeger, N., Yu, P., Cai, S., Nielsen, E., Soegel, D., Bileschi, S., & others. (2019). Tensorflow. js: Machine learning for the web and beyond. *Proceedings of Machine Learning and Systems*, *1*, 309–321. ↩
- Steinberg Media Technologies. (2022). Virtual Studio Technology (VST). In *VST 3 SDK* . Steinberg Media Technologies. https://steinbergmedia.github.io/vst3_doc/vstsdk/index.html ↩
- Sturm, B., Santos, J. F., & Korshunova, I. (2015). Folk music style modelling by recurrent neural networks with long short term memory units. *16th International Society for Music Information Retrieval Conference*. ↩
- Tokui, N. (2020). Towards democratizing music production with AI-Design of Variational Autoencoder-based Rhythm Generator as a DAW plugin. *CoRR*, *abs/2004.01525*. https://arxiv.org/abs/2004.01525 ↩
- Wu, S.-L., & Yang, Y.-H. (2020). The Jazz Transformer on the Front Line: Exploring the Shortcomings of AI-composed Music through Quantitative Measures. In J. Cumming, J. H. Lee, B. McFee, M. Schedl, J. Devaney, C. McKay, E. Zangerle, & T. de Reuse (Eds.), *Proceedings of the 21th International Society for Music Information Retrieval Conference, ISMIR 2020, Montreal, Canada, October 11-16, 2020* (pp. 142–149). http://archives.ismir.net/ismir2020/paper/000339.pdf ↩
- Yu, B., Lu, P., Wang, R., Hu, W., Tan, X., Ye, W., Zhang, S., Qin, T., & Liu, T.-Y. (2022). Museformer: Transformer with Fine- and Coarse-Grained Attention for Music Generation. *CoRR*, *abs/2210.10349*. https://doi.org/10.48550/arXiv.2210.10349 ↩
- Zeng, M., Tan, X., Wang, R., Ju, Z., Qin, T., & Liu, T.-Y. (2021). *MusicBERT: Symbolic Music Understanding with Large-Scale Pre-Training*. ↩