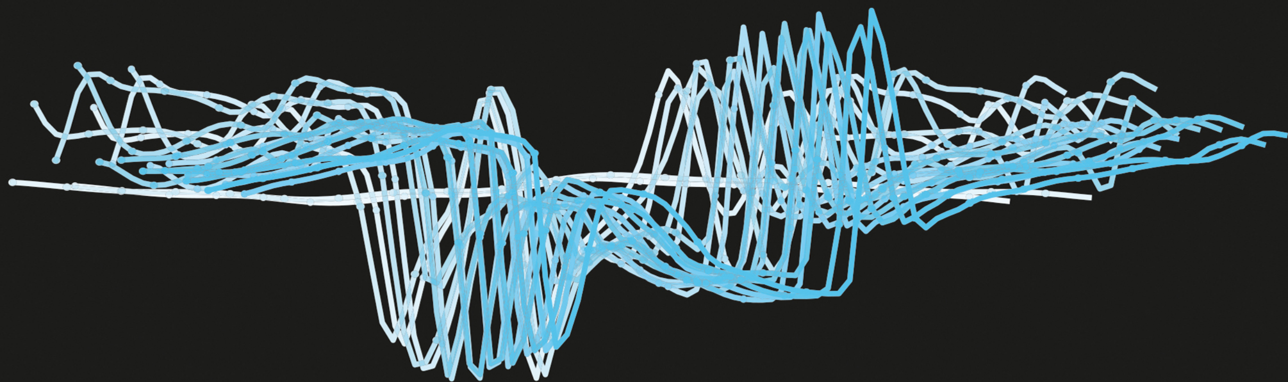
A Focal Press Book

BUILD AI-ENHANCED AUDIO PLUGINS WITH C++



MATTHEW JOHN YEE-KING

ROUTLEDGE

Build AI-Enhanced Audio Plugins with C++

Build AI-Enhanced Audio Plugins with C++ explains how to embed artificial intelligence technology inside tools that can be used by audio and music professionals, through worked examples using Python, C++ and audio APIs which demonstrate how to combine technologies to produce professional, AI-enhanced creative tools.

Alongside a freely accessible source code repository created by the author that accompanies the book for readers to reference, each chapter is supported by complete example applications and projects, including an autonomous music improviser, a neural network-based synthesizer meta-programmer and a neural audio effects processor. Detailed instructions on how to build each example are also provided, including source code extracts, diagrams and background theory.

This is an essential guide for software developers and programmers of all levels looking to integrate AI into their systems, as well as educators and students of audio programming, machine learning and software development.

Matthew John Yee-King is a professor in the department of computing at Goldsmiths, University of London. He is an experienced educator as well as the programme director for the University of London's online BSc Computer Science degree.

"This book is long overdue. With the explosion of activity in the field of AI-assisted music creation, the need for mastering all the chain of software from ideas to actual plugins is stronger than ever. Matthew has a direct, hands-on approach that not only will be of great help to people wanting to contribute to the field, but will also encourage others to experiment and share their code. Matthew's experience in teaching shows and definitely contributes to making the book easy to read and to-the-point."

François Pachet, *Research Director*

Build AI-Enhanced Audio Plugins with C++

Matthew John Yee-King

LONDON AND NEW YORK

Designed cover image: Matthew John Yee-King

First published 2024
by Routledge
4 Park Square, Milton Park, Abingdon, Oxon OX14 4RN

and by Routledge
605 Third Avenue, New York, NY 10017

Routledge is an imprint of the Taylor & Francis Group, an informa business

© 2024 Matthew John Yee-King

The right of Matthew John Yee-King to be identified as author of this work has been asserted in accordance with sections 77 and 78 of the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this book may be reprinted or reproduced or utilised in any form or by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying and recording, or in any information storage or retrieval system, without permission in writing from the publishers.

Trademark notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

ISBN: 978-1-032-43046-1 (hbk)
ISBN: 978-1-032-43042-3 (pbk)
ISBN: 978-1-003-36549-5 (ebk)

DOI: 10.4324/9781003365495

Typeset in Computer Modern by Matthew John Yee-King

Access the Support Material: www.yeeking.net/book

Publisher's Note
This book has been prepared from camera-ready copy provided by the author.

For Sakie, Otoné, and my family. And of course, Asuka the beagle.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Contents

Foreword x

List of figures xi

I Getting started 1

1 Introduction to the book 2

2 Setting up your development environment 11

3 Installing JUCE 23

4 Installing and using CMake 32

5 Set up libtorch 44

6 Python setup instructions 53

7 Common development environment setup problems 60

8 Basic plugin development 62

9 FM synthesizer plugin 72

II ML-powered plugin control: the meta-controller 80

10 Using regression for synthesizer control 81

11 Experiment with regression and libtorch 87

12 The meta-controller 98

13 Linear interpolating superknob **103**

14 Untrained torchknob **107**

15 Training the torchknob **119**

16 Plugin meta-controller **129**

17 Placing plugins in an AudioProcessGraph structure **135**

18 Show a plugin's user interface **143**

19 From plugin host to meta-controller **151**

III The autonomous music improviser **157**

20 Background: all about sequencers **158**

21 Programming with Markov models **169**

22 Starting the Improviser plugin **174**

23 Modelling note onset times **187**

24 Modelling note duration **194**

25 Polyphonic Markov model **200**

IV Neural audio effects **209**

26 Welcome to neural effects **210**

27 Finite Impulse Responses, signals and systems **214**

28 Convolution **220**

29 Infinite Impulse Response filters **231**

30 Waveshapers **241**

31 Introduction to neural guitar amplifier emulation **254**

32 Neural FX: LSTM network **261**

33 JUCE LSTM plugin **274**

34 Training the amp emulator: dataset **287**

35 Data shapes, LSTM models and loss functions **296**

36 The LSTM training loop **309**

37 Operationalising the model in a plugin **315**

38 Faster LSTM using RTNeural **320**

39 Guide to the projects in the repository **328**

Bibliography **335**

Index **340**

Foreword

I am delighted to present my book on building AI-enhanced audio software. My name is Matthew Yee-King, and I work as an educator, musician, and computer music researcher at Goldsmiths, University of London. I have written this book because I started making AI-powered audio plugins myself and found that musicians were much happier using plugins than other forms of software as they integrate with their existing tools. But there were no detailed instructions on how to integrate machine learning with audio in C++ for plugin development... until now! I hope you enjoy the book and find the techniques helpful for your work with AI-enhanced audio software. Here's my bio:

As an educator, I have taught undergraduate courses in digital signal processing, creative audio programming, software engineering, and artificial intelligence using languages such as Java, JavaScript, C++, and Python on-campus and online. I am the academic director for the first undergrad programme on the Coursera platform, the University of London's BSc Computer Science programme. In 2023, the course has seen over 8,000 students from more than 120 countries.

As a musician, I have released electronic music on Aphex Twin's Rephlex Records, Warp Records, and others. I have collaborated with many artists, including Tom Jenkinson (Squarepusher), Tom Skinner (Smile Band), Matthew Herbert, Finn Peters, Alex McLean, and Max de Wardener. My main live instrument is the drum kit, acoustic or electronic, but I have also worked as a sound designer, SuperCollider programmer, and creator of musical AI systems.

As a computer music researcher, I have developed music software and published papers on autonomous musical agents, automatic sound synthesiser programming, and systems supporting music education. I have worked on research projects as a research engineer, a post-doc, a co-I, and a PI. I have collaborated with individuals and research groups around the world, including Mark d'Inverno (my fantastic mentor), Andrea Fiorucci, Francois Pachet, Jon McCormack, Mick Grierson, Nick Collins, Rebecca Fiebrink, the Sony Computer Science Laboratory Paris, the Artificial Intelligence Research Institute, CSIC Barcelona, the Department of Human Centred Computing, Monash University Australia, and Politecnico di Milano, Italy.

I would like to thank all the amazingly talented musicians (especially Gaz, Domenico, Finn, Alex, and Max) and researchers I have worked with for inspiring me and, indeed, the work on which some of the projects in the book are based.

List of figures

1.1 Mountainous landscape depicting musical competencies, with the water of AI rising to consume them. 2

2.1 Many component parts are needed to build AI-enhanced audio software. 11
2.2 The components involved in AI-enhanced audio application development. 12
2.3 Setting up your development environment involves complex machinery and lots of steps. 18
2.4 Creating, building and running a C++ console program in Visual Studio. 19
2.5 C++-related packages that you should install. 19
2.6 Creating, building and running a C++ console program in Xcode. 20
2.7 My development setup showing an M1 Mac running macOS hidden on the left, a ThinkPad running Ubuntu 22.04 in front of the monitor and a Gigabyte Aero running Windows 11 on the right. . 22

3.1 The available application types for Projucer. 24
3.2 Projucer project view. The exporter panel is exposed on the left, module configuration panel is on the right. Note that my modules are all set to Global. 25
3.3 Building the Standalone solution in Visual Studio Community 2022. 27
3.4 Enabling console output for a JUCE project in Visual Studio/ Windows. At the top: redirect text output to the immediate window and open the immediate window. At the bottom: a program running with DBG output showing in the immediate window. 28
3.5 Running a JUCE plugin project in Xcode – make sure you select Standalone. 29
3.6 The JUCE AudioPluginHost application, which comes with the JUCE distribution. One of its built–in plugins, a sine synth, is wired to the MIDI input and the audio output. 30

3.7 The list of available plugins in the JUCE AudioPluginHost app. I have clicked the options menu which is showing its 'scan for new or updated ...' function. 31

4.1 CMake running in the Windows Powershell. 33
4.2 A CMake project viewed in VSCode. 36
4.3 A CMake project viewed in Visual Studio Community 2022. . . . 38
4.4 A CMake project viewed in Xcode. 39

6.1 A Jupyter notebook in action. There are cells containing Python code which you can execute. If you trigger a plot command, the plot will be embedded in the worksheet. 57

8.1 dphase depends on the sample rate (the space between the samples) and the frequency (how fast you need to get through the sine wave). 64
8.2 A synthesizer plugin loads data into the incoming blocks. 65
8.3 Printing descriptions of MIDI messages coming into a plugin in Standalone mode from a USB controller keyboard (left) and MIDI coming from an on-screen piano keyboard in AudioPluginHost (right). 70

9.1 Simple FM plugin with sliders for frequency, modulation index and modulation depth. 75
9.2 Showing plugin parameters for the Surge XT synthesiser using AudioPluginHost. 76
9.3 Showing plugin parameters for the FM plugin using AudioPluginHost. 77
9.4 Showing the custom UI for the FM plugin (right), the auto-generated parameter UI (middle) and AudioPluginHost (left). . . . 78

10.1 The meta-controller uses regression to control other plugins. 81
10.2 Linear regression finds the straight line that best fits some data. Important features of the line are the point at which it intercepts the y-axis and the slope gradient. 82
10.3 Linear regression with two lines, allowing the estimation of two parameters given a single 'meta-controller' input control. The x-axis represents the control input and the y-axis shows the settings for the two parameters the control is mapped to. 84
10.4 A neural network applies a function to its input. 85
10.5 A neural network scales by a weight and adds a bias. 86

11.1 Simple single layer network with one input and one output. 91

11.2 More complex single layer with more inputs and outputs. Now we apply a weight to each input as it goes to each output. We then sum the weighted inputs and apply a bias to each output. 93
11.3 The optimiser adjusts the network weights. 95

12.1 The meta-controller uses a neural network for new methods of synthesizer sound exploration. 98
12.2 The Wekinator workflow: data collection, training, inference then back to data collection. 101

13.1 User interface for the simple two-parameter FM synthesizer. The toggle switch switches between drone and envelope mode, the two sliders control modulation depth and index and the piano keyboard allows you to play notes on the synthesizer. 103
13.2 Superknob UI on the left. On the right is a closer view of a range slider. Small triangles above and below the line allow the user to constrain the range of the main slider control. 104

14.1 The torchknob system architecture. 108
14.2 Basic architecture where a linear layer passes into a softmax layer. The numbers in the brackets indicate input and output shape. The linear layer input (2,1) goes from 1 value to 2 nodes, then output (2,2) goes from 2 nodes to 2 outputs. 115

15.1 Interactive machine learning provides more intuitive training for neural networks. 119
15.2 User interface mockup for trainable superknob system (left). We have an additional knob to specify training input without triggering the movement of the sliders. Actual user interface prototype (right). 120
15.3 Example of an experiment you can carry out. First, set the training slider to its lowest value, the same for the modulation controls. Add a training point. Then move to the middle positions, and add a training point. Finally, move to the highest positions, and add a training point. 125
15.4 The learner.js/ Wekinator regression architecture (top). The simpler architecture we used previously (bottom). 125

16.1 Hosting plugins allows you to control more advanced synthesizers. 129

17.1 Wiring plugins together with a processor graph. 135
17.2 The graph you will create. 139
17.3 User interface for the basic host with the load plugin button added. 141

18.1 Class hierarchy for AudioProcessor and its descendants. 143
18.2 User interface for the host with the Surge XT plugin user interface showing in a separate window. 144
18.3 User interface for the host with a show UI button. 145

19.1 User interface for the Dexed DX-7 emulator. It has 155 parameters. 151
19.2 Time for a more complex neural network. 153

20.1 How far are modern sequencers from steam-powered pianos? 158
20.2 My own experience interacting with AI improvisers. Left panel: playing with Alex McLean in Canute, with an AI improviser adding even more percussion. Right panel: livecoding an AI improviser in a performance with musician Finn Peters. 159
20.3 Visualisation of a two-state model on the left and the state transition probability table on the right. 164
20.4 Visualisation of a variable order Markov model containing first and second order states. 166

21.1 Example of the Markov model generated by some simple code. . . 170

22.1 Overview of the autonomous improviser plugin. Yes, a keytar. . . . 175
22.2 The user interface for the basic JUCE MIDI processing plugin. . . 176
22.3 The MIDI Markov plugin running in AudioPluginHost. Note how it receives MIDI and then passes it on to the Dexed synthesiser. . . 181
22.4 Using AudioPluginHost's MIDI Logger plugin to observe the MIDI coming out of the Markov plugin. 182

23.1 Note duration is the length the note plays for. Inter-onset interval is the time that elapses between the start of consecutive notes. . . 187
23.2 Measuring inter-onset-intervals. The IOI is the number of samples between the start and end sample. elapsedSamples is the absolute number of elapsed samples since the program started and is updated every time processBlock is called; message.getTimestamp() is the offset of the message in samples within the current block. 188

24.1 Measuring note duration has to cope with notes that fall across multiple calls to processBlock. 194
24.2 Testing the getTimestamp function on note–on messages – the timestamp is always between zero and the block size of 2048. . . . 196

25.1 If notes start close enough in time, they are chords. If the start times fall outside a threshold, they are single notes. This allows for human playing where notes in chords do not happen all at the same time. 202

26.1 Tape manipulation was an early form of audio effect. 210

27.1 The impulse signal and the impulse responses of a one-pole, two-pole and three-pole system. 216

28.1 Original drum loop spectrum on the left, filtered version on the right. High frequencies have been attenuated in the filtered spectrum. 224

29.1 Infinite impulse responses are powerful! 231
29.2 Pole for pole, IIR filters generate much richer impulse responses. The left panel shows a two-pole, FIR. The right pane shows a two-pole IIR. 232
29.3 Comparison of two pole FIR filter (left) and IIR filter (right). The IIR filter has a more drastic response. 233
29.4 Two types of IIR filter and their frequency responses. IIR filter design is a compromise. 237

30.1 Digital signal processing makes waveshaping much easier than it used to be. 241
30.2 The effect of different waveshaper transfer functions on a sinusoidal signal. Top row: transfer functions, middle row: sine wave signal after waveshaping, bottom row: spectrum of waveshaped sine wave. 242
30.3 Automatically generated generic UI for the waveshaper plugin. . . 248
30.4 A sine wave passing through a series of blocks that emulate in a simplified way the processing done by a guitar amplifier. 249
30.5 Capture an impulse response for the convolutional cabinet simulator. 253

31.1 Capturing training data from a guitar amplifier. 256
31.2 Four stages to train a neural network. 1: send the test input through the device (e.g. amp) you want to model, 2: send the test input through the neural network, 3: compute the error between the output of the network and amp, 4: update network parameters to reduce error using back-propagation. Back to stage 2. 257

32.1 What does our simple, random LSTM do to a sine wave? It changes the shape of the wave and introduces extra frequencies. 266

32.2 The steps taken to process a WAV file with a neural network through various shapes and data formats. 270
32.3 Time taken to process 44,100 samples. Anything below the 1000ms line can potentially run in real-time. Linux seems very fast with low hidden units, but Windows and macOS catch up at 128 units. . . 272

33.1 Block-based processing leads to unwanted artefacts in the audio. The left panel shows the output of the network if the complete signal is processed in one block. The right panel shows what happens if the signal is passed through the network in several blocks. The solution is to retain the state of the LSTM between blocks. 277
33.2 Breakdown of the data type used to store LSTM state. 280
33.3 The LSTM plugin running in the AudioPlugHost test environment, with an oscilloscope showing a sine wave test tone before and after LSTM processing. 284

34.1 How fast can LSTMs process audio? 287
34.2 Four stages to train a neural network. 1: send the test input through the device (e.g. amp) you want to model, 2: send the test input through the neural network, 3: compute the error between the output of the network and amp, 4: update network parameters to reduce error using back-propagation. Back to stage 2. 288
34.3 Tensorboard is a web-based machine learning dashboard. Here, you can see a list of training runs (1) and graphs showing training progress in terms of training (2) and validation (3) errors on two separate runs. 291
34.4 Spectrogram of the Atkins training signal 'v2_0_0.wav'. You can see the signal is quite varied and dynamic. 292
34.5 Capturing training data from a guitar amplifier is similar to 're-amping'. 292
34.6 Clean signal (top) and re-amped signal (bottom) in Reaper. . . . 294

35.1 Sequence length and batch size. 298
35.2 An LSTM network with a four hidden unit LSTM layer and a densely connected unit which 'mixes down' the signal to a single channel. 303
35.3 What does loss mean? The top two plots show extracts from the target output. The middle two plots show the output of an untrained (left) and trained network (right). The right-hand side is much closer to the target. The bottom plots show a simple error between each point in the two plots above. The sum of these values could be a simple loss function. 305

36.1 The training loop. Data is processed in batches with updates to the network parameters between batches. Between epochs, checks are done on whether to save the model and exit. 310
36.2 Comparison of training runs with different sized LSTM networks. At the top you can see the input signal and the target output signal recorded from Blackstar HT-1 valve guitar amplifier. The descending graphs on the left show the validation loss over time for three LSTM network sizes. The waveforms show outputs from the networks before and after training. 314

38.1 A sinusoidal test signal passing through an RTNeural LSTM distortion effect in AudioPluginHost. The sinusoidal wave is the original signal, the clipped out wave is the LSTM-processed signal. 325

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Part I

Getting started

1

Introduction to the book

Welcome to **'Build AI-Enhanced Audio Plugins with C++'**! You are about to embark on a journey into a world of advanced technology, which will be an essential part of the next generation of audio software. In this chapter, I will introduce the general area of AI-music technology and then set out the book's main aims. I will identify different types of people: audio developers, student programmers, machine learning engineers, educators and so on and explain how each group can get the best out of the book. I will explain that you can use any of the large amounts of code I have written for the book however you like. I will also explain the dual licensing model used by the JUCE library. I will finish with a straightforward, working definition of artificial intelligence.

1.1 Exciting times for artificial intelligence

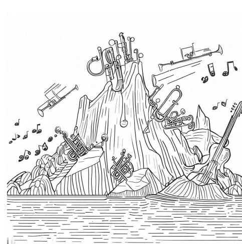


FIGURE 1.1
Mountainous landscape depicting musical competencies, with the water of AI rising to consume them.

As I write this book, we are in exciting times for the progress of artificial intelligence (AI). AI theorist Hans Moravec presents a compelling view of the progress of AI wherein he places human competencies such as picking up a cup, composing a symphony and so forth within a mountainous landscape he calls a "landscape of human competencies"[29]. Complex or highly valued competencies are situated on higher peaks, although people may disagree on what should go where.

In Moravec's landscape of human competencies, artificial intelligence is a sea washing around and rising up the peaks. If the water rises above the position of a human competency, AI has that competency. Right now, there is no doubt that the sea is rising, especially given recent advances in deep neural networks. Game playing is a popular area for AI researchers, and it can provide some perspective on the position of the water level. Chess is underwater[20],

Go is underwater[38], and so is Heads-up no-limit Texas hold'em Poker[5]. But picking up cups is still hard at the time of writing. This leads us to another insight from Moravec known as Moravec's Paradox. Things that people find hard are often easy for AI, and things that people find easy can be very hard for AI.

What about music and sound engineering competencies? Well-known baroque composer Bach is swimming in the sea of AI after Hadjeres's DeepBach created Bach chorales that could fool experts[14]. AI can learn how to program the notoriously difficult to program Yamaha DX-7 synthesizer to make a given sound[49], and AI can automatically mix multi-track audio recordings[28]. AI has even gained competencies which humans do not have. Timbre transfer allows us to use the pitch and dynamics of one instrumental performance to play using the tone of another instrument[12], and sound separation allows us to un-mix a track back to its separate stems[41]. Recent advancements in voice processing technology have allowed 'deep fakes' wherein the voices of dead musicians can be re-animated as if in a kind of Lovecraftian fantasy and made to sing cover versions[1].

Many of those examples are from the research literature about AI and music but commercial music software companies are increasingly using AI in their products. A notable example is Waves and their neural network-powered noise reduction technology Clarity Vx. Another example is Steinberg's GANDrum system, which uses generative adversarial neural networks to synthesize unique drum sounds. There are also an emerging range of neural network powered guitar effects such as Neural DSP's Quad Cortex guitar amplifier emulator.

So right now is an exciting time to be working as a music technology developer because a revolution is underway which is likely to be at least as transformative as the desktop Digital Audio Workstation revolution from the late 90s, which put advanced recording and audio processing capabilities in the hands of anyone who owned a desktop computer. In fact, AI is even more exciting than DAW technology because it is transforming the way we produce and create music in ways that were previously unimaginable. The possibilities are profound, and I am excited to see what the future holds.

1.2 The aims for this book

The first aim of this book is to show you how to build AI-enhanced music software. But I will take a different approach to many other AI and signal processing books. I have carefully developed this approach to solve specific problems and to address particular challenges you will face as a developer of AI-enhanced music software.

[1] https://github.com/svc-develop-team/so-vits-svc

One problem with many AI-music systems I mentioned is that they only exist as descriptions in research papers. The research papers aim to describe systems and their performance to other AI-music experts using a very limited number of pages. They are not tutorials and do not necessarily explain the nuts and bolts needed to build a complete working system.

Sometimes the researchers who write these research papers provide source code repositories but my experience has been that it can be challenging to operationalise the software from these source code repositories. Making the code run likely involves having a particular combination of particular versions of other components installed and often a particular operating system. The challenge is even more significant if there is no source code repository. I have watched excellent PhD students labour for weeks to re-implement systems described in research papers, only to discover there are vital details that should have been included in the paper or other technical issues.

These problems mean that AI-enhanced music systems are not easily accessible to musicians wishing to use them and probably not to programmers wanting to integrate them into innovative music software. This is where this book comes in. This book will show you several examples of how to build complete working AI-music systems. All source code written by me is provided in a repository and is covered with a permissive open-source license, allowing you to re-use it how you see fit. The book also uses a consistent technical setup, allowing you to easily access and "wrench on" the examples. So my first aim is really to make AI-enhanced music technology available and transparent for you.

My second aim is to show you how to construct the technology in a way that makes it accessible to musicians and audio professionals. Knowing how to build and run AI-enhanced music systems on *your* machine is one thing. A quirky Python script hacked together to work on your setup is acceptable for research and experimentation but you will have trouble getting musicians to use it. What is the ideal method for sharing software in a form that musicians and other audio professionals can use it? The next aim of this book is to answer that question and to apply the answer to the design of the examples in the book. For many years I have worked as a researcher/engineer on research projects where one of the aims has been to get new technology into the hands of users so we can evaluate and improve that technology. There are many approaches to achieving this aim: running workshops with pre-configured equipment, making the technology run in the web browser and so on. All of these approaches have their merits and are appropriate in different circumstances. But none of them is quite suitable for our purposes here. Here we are aiming to write software that can be used by musicians, producers and sound engineers with minimal effort on their part.

How can you achieve this aim of having as many audio professionals as possible to be able to use your software, and how can this book help you? Firstly, making software so it integrates with existing creative workflows is crucial. The simplest

way to do that in music technology is via standard plugin frameworks such as VST and Audio Units or standalone, native applications that work with standard protocols such as MIDI and OSC. In this book, we will do precisely this by building software that integrates effectively with existing workflows and technology.

1.3 What is in the book?

I have organised the book into four parts:

1: Getting started. You will set up your system for the development work in the book and build some example plugins and other test programs.

2: ML-powered plugin control: the meta-controller. You will build the first large example in the book, a plugin that hosts and controls another plugin using a neural network.

3: The autonomous music improviser. The second large example is a plugin that can learn in real-time from incoming MIDI data and improvise its own interpretation of what it has learned.

4: Neural audio effects. The third large example is a plugin that models the non-linear signal processing of guitar amplifiers and effects pedals using neural networks.

Each part of the book contains detailed instructions on how to build each example, including source code extracts, diagrams and background theory. I have also included some brief historical and other context for the examples. The examples in parts 2, 3, and 4 are independent, so you can jump to any of those parts once you have completed part 1.

I have created a freely accessible source code repository, currently on Github, which provides each of the examples above in various states of development. As you work through each stage of developing each example, you can pull up a working version for that stage from the repository, in case you get stuck. The repository contains releases of each final product with compiled binaries and installers.

1.4 How to use this book

There are different ways to use the book. The most straightforward approach is to download the releases of the examples and experiment with them in your DAW.

But you will miss out on a lot of learning if you only do that! To fully exploit the content in the book, you will need to start by working through part 1, which explains how to set up your system for the development work described in the book. You will find 'Progress check' sections at the end of each chapter, which clarify what you should have achieved before continuing to the next chapter.

After part 1, there are three detailed example projects, each providing very different functionality. These three projects are independent, so you can choose which order you study them in or only study some and not others.

As you work through the parts of the book, each program you are developing will increase in complexity until it is completely functional at the end of the book part. To work through the examples, you can type in all the code you see in the book and build the complete example by hand, or you can read the code and download the step-by-step versions of the projects from the repository. I find that people sometimes get really stuck working through these larger projects, where they cannot make it compile or work properly. So, I have provided staged versions of the programs in the code repository. If you reach the end of a chapter and cannot figure out why your program does not work, you can just pick up from a working stage in the code repository and continue. Of course, there is much to be gained by spending hours looking for that missing bracket, so do try and debug your problems before grabbing the working version from the repo.

At the end of some chapters and at the end of all parts, I suggest challenges and extensions. These are extra features you can add to the plugins, allowing you to reinforce and increase your understanding of the principles and techniques. If you are using the book for teaching, these challenges and extensions are things you could set students for coursework.

1.5 Who might use this book?

In the following sections I will mention a few different types of people who would be interested in the book and how they can get the best out of it.

1.5.1 Student programmer

If you are a student programmer at the undergraduate or postgraduate level, you will find new knowledge in all areas of the book. You might be assigned all or part of this book as a textbook for a course you are studying, or you might have discovered the book independently. The book will teach you some C++ programming along the way, as well as helping you to understand how to use an IDE and associated tools to develop software. You will probably find the setup section of the book

very helpful as, in my experience, students spend a lot of time struggling to get their software development environment set up and working correctly. The book also provides information about general audio processing techniques and how they can be adapted using AI technology.

1.5.2 Audio programmer

You will find the book helpful if you are an audio programmer wishing to integrate AI technology into your software. The book will show several detailed examples of how you can employ different AI techniques in music software. The code repository contains permissively licensed code which you can use how you see fit in your projects. The book considers cross-platform development and is focused on developing software that integrates with existing music production workflows and technology. You will probably be familiar with the audio side of the development and theory work in this book, and I will try to make connections between this knowledge and the AI domain. For example, you will discover how neural networks are just big fancy signal processors.

1.5.3 Machine learning engineer

If you are a machine learning engineer or AI scientist wishing to apply your skills in the music software domain, the book will help you do that. You will be familiar with the concepts of machine learning models, training and inference, but you will likely need to become more familiar with digital signal processing techniques. You may be unfamiliar with common music technology such as plugins, MIDI data, etc. The book will explain exactly how to work with those technologies. As a machine learning engineer, you likely work primarily in Python or a specialised language. The book covers some Python but is mainly focused on C++ programming. You should find the information about setting up a C++ development toolchain useful.

1.5.4 Educator

If you are an educator planning to use this book as part of your teaching, that is an excellent idea! The most obvious way to use the book in your teaching is to split the content between lectures and lab classes. In the lectures, you can introduce the AI theory. You can go as far as you like with the AI theory, depending on the level and focus of your course. I cover enough in the book to enable the reader to carry out training and inference and to integrate the AI system into a working application. I also explain some characteristics of the particular machine learning techniques used. You can go much further than that, depending on your requirements. For your lab classes, your students can work through the practical implementation of the applications. The book contains detailed instructions on how to build each of

the example applications. I have battle tested and iterated these instructions with my students.

1.5.5 Sound engineer/ sound designer/ musician

You might be a sound designer or musician who wants deeper control and knowledge of AI-enhanced music systems. In that case, you probably need to become more familiar with the world of software development or machine learning and AI, and the book will help you to achieve that. You will have strong domain knowledge covering the musical aspects of the systems developed in the book. You should start by checking out the example software available in the repository. If you are intrigued by those examples, you can use the book to learn more about how they work. That can lead you to customise the software to suit your needs or ideas better.

1.6 The source code repository

The book comes with a source code repository on GitHub. The source code repository contains all the code for the projects described in the book and various instructional materials. You should go ahead and install the git tool and clone the repository to your machine right now. Chapter 39 describes each of the projects you will find in the source code repository. I refer to this chapter when I am working through example projects to ensure you can access the correct code for that project directly. This is a book, and books cannot easily be updated, but the software libraries we are working with are regularly updated. This causes tension between the desire to provide the most correct and up-to-date instructions and code in the book and wanting to ensure the instructions remain correct. If you find any bugs in the example code in the repository or in the book, please report them as issues on the GitHub page. I cannot do much about updating the code in the book (pending a new edition), but I will ensure the repository is as bug-free as possible.

1.6.1 About source code licensing

In this section, I will explain the licensing model used for code in this book. When I read these kind of legal details of source code licensing, my eyes glaze over, and I start wishing for the paragraph to end. But please read this section, as it is important to understand how to use the code here. Here is the executive summary: you can use any of the code I have written for this book freely in your commercial

projects but if you want to use the complete examples in a closed-source manner you should ensure you understand the JUCE library's dual licensing model.

Open-source has various definitions. For our purposes, open-source means that the source code for a piece of software is available. Open-source code generally comes with a license that dictates what the code's author wants you to do with that code. Permissive licenses, such as the MIT license, place few limitations on the use of that code. Users are free to use that code as they see fit. Users of MIT-licensed code can adapt the code and even include it in commercial projects without needing to release their adapted code. All the code written by me for this book is MIT licensed.

The GPL licenses take a stronger philosophical position concerning freedom and are designed to encourage further sharing of source code. You can adapt and use GPL'd code in your project, even if it is a commercial project, but you will be required to release your code. You are also obliged to make your source code open-source with a GPL license if your code links to GPL libraries.

Releasing source code with multiple licenses is possible if they are compatible. The code in this book that I have written is released under a dual MIT / GPL license. I will explain why below. If you use my code in your project, the MIT license applies. The GPL license applies if you use my complete examples, which also link to GPL'd code.

The reason I have dual-licensed the code is because of the JUCE library. The JUCE library carries a dual license. If you build against the JUCE library, you can either GPL your whole project or apply for a JUCE license, and then, you do not need to GPL your code.

1.6.2 Example code-use scenarios

Here are some examples to illustrate typical ways you might want to work with the source code from this book.

1. I want to experiment with the examples in this book. I am not planning to release anything commercially. Great – you can do that, no problem.
2. I want to use the code written by the author of this book in my own plugin projects, where I am using my own framework instead of JUCE. So I am not going to build against the JUCE library. Great – the MIT license applies to my code in that scenario, and you are free to release projects which use that MIT code commercially.
3. I have adapted an example in the book into a really cool plugin I want to release commercially. I do not want to release my complete code with a GPL. It uses the JUCE library. Do not worry; you can do that. You need to get yourself an appropriate license from the JUCE web page.

1.7 User-readiness

I will show you how to get the software to a point where it will run on your machine and, with some fiddling, on other people's machines. I do not cover the creation of installers or the process of signing / notarising software. Some great resources online tell you what to do with installers, signing etc., once your amazing AI-powered plugin is ready for the world.

1.8 A working definition of artificial intelligence

To complete the pre-amble here, I would like to provide a working definition of what I mean when I refer to artificial intelligence and how that differs from machine learning and other software development techniques. Here goes:

An artificially intelligent system is an automated system that can carry out a task generally considered to require intelligence were it to be carried out by a human. Machine learning refers to a set of techniques that can be used to build artificially intelligent systems amongst other things. Machine learning techniques involve learning in the sense that the program changes itself or its parameters in a manner that allows it to perform a task more and more effectively. Often machine learning involves learning patterns in data.

2

Setting up your development environment

In the next few chapters, I will explain how you can set up your development environment for audio software development. You need this setup to work on and run the example programs in the book. After working through these chapters, you should be able to build and run a simple C++ program linked to the JUCE audio library and the libtorch machine learning library using an integrated development environment (IDE). The chapters should also familiarise you with the CMake tool which will allow you to create cross-platform projects with which you can create native applications and plugins for Windows, macOS and Linux systems. You can use these setup chapters how you like – read them, scribble on them, etc. but I recommend working through the material with a computer available. Expect to install software on the computer, run commands in its command shell and execute programs.

2.1 Component parts

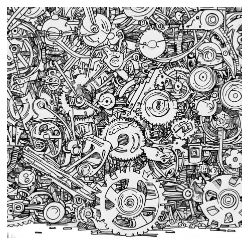


FIGURE 2.1
Many component parts are needed to build AI-enhanced audio software.

Before we set up the development environment, I would like to describe some key components you will encounter in this environment. You are probably familiar with several of these components, but I am describing them here to clarify what I plan to use them for. Several of these components are shown in relation to each other in figure 2.2.

2.1.1 Purpose of different components

Build tool

A build tool helps developers specify how their software should be built. For example, what are the software libraries they are using? Where are the source code files? What is the target platform? Build tools are handy when you want

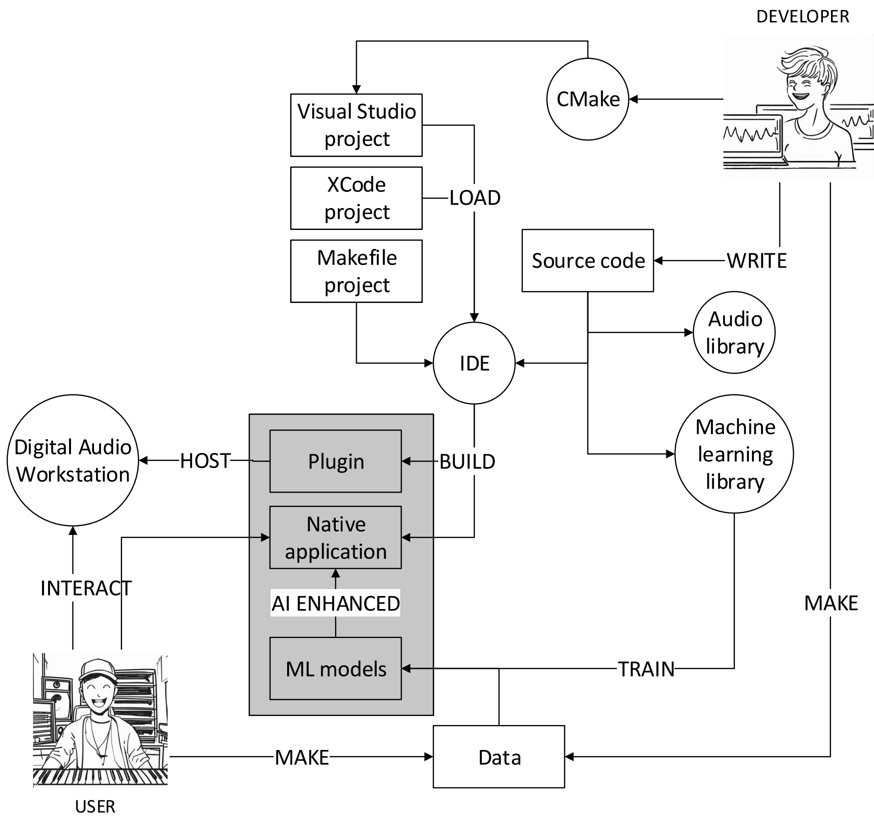


FIGURE 2.2
The components involved in AI-enhanced audio application development.

to be able to build your software for multiple platforms using different Integrated Development Environments (IDEs). We will use the CMake[1] build tool to help us generate projects for various IDEs. This will make building the software for different hardware and OS platforms possible.

CMake

CMake is the build tool we will use in this book. With CMake, you write a single configuration file then you can use it to generate projects for different IDEs. In the configuration file, you can specify different targets for the build, such as a test program and a main program. You can specify associations between your project and external libraries. You can specify actions to be taken, such as copying files. This makes it a valuable tool for audio application developers wishing to support various operating systems, as you can maintain a single CMake configuration and codebase and use it to build for multiple platforms.

Integrated Development Environment (IDE)

An IDE is a set of tools that enable a developer to write, build and debug software. The IDE will be the hub of your software development activity. Using the CMake build tool, you can generate projects for different IDEs from the same codebase. Many IDEs are available, but I will cover Xcode for Apple device development, Visual Studio for Windows and Visual Studio Code or a custom setup for GNU/Linux in this book. In fact, you can also use Visual Studio Code for Windows, Apple and Linux development, which allows for a consistent environment across platforms.

Codebase

The codebase is the set of source code files in a project. The build tool and some handy macros will allow us to have a single codebase for all platforms.

Native program

Some of the programs you encounter in the book will be compiled into native programs in machine code that run on particular CPU hardware. We will write these programs using the C++ language. Native programs are most appropriate when developers wish to integrate directly with plugin and Digital Audio Workstation (DAW) technology. For example, VST3 plugins are native programs. Native programs generally run faster than interpreted programs, and that is important for realtime audio applications.

[1] https://CMake.org/

Interpreted program

Some of the programs we write will be interpreted as opposed to compiled. Interpreted programs are converted to machine code on the fly instead of being converted into machine code before running. We will write interpreted programs in the Python language. Interpreted programs are more suited to the kind of experimentation one needs to do when developing machine learning models. It is common for AI researchers to provide Python code along with their research papers to allow other people to explore their work more easily. It is less common for researchers to provide C++ code, but there has been a trend towards this in AI-music research in the last few years.

Machine learning model

Machine learning models carry out the smart processing associated with artificial intelligence systems. A neural network is an example of a machine learning model. You can think of a machine learning model as a kind of data processing black box that can learn to process data in a way that is useful to us. The structure of the model (inside the black box) and its parameters dictate what kind of processing it does. Depending on the application, we will sometimes develop the models using Python and sometimes C++. Experimenting with model designs and training data in Python is generally more straightforward. C++ allows the models to be integrated into native applications and plugins so they can be accessed by regular users.

Trained model

A trained model is a machine learning model that has learnt something valuable. The model defines the structure of the machine learning component. Training teaches that component to process data in a particular way. The ability to self-configure through learning is the essence of machine learning. Trained models provided by a third party are often called pre-trained models. In case you have heard of openAI's infamous GPT model, the 'P' stands for pre-trained.

Inference

Inference is the process of using a trained model to generate an output. Inference does not change the internal configuration of the machine learning model; it just passes data through it. Once a model is trained, you will use it for inference.

Generative model

A generative model is a machine learning model that can generate something interesting. For example, instead of detecting cats in images, it might generate images of cats. The 'G' in GPT stands for generative, as GPT generates text.

Machine learning library

A machine learning library is a set of components that makes it easier to carry out machine learning tasks. Typical components include different algorithms such as clustering and neural networks, routines to train the models, and data importers. Examples are sci-kit learn, TensorFlow and PyTorch. We will use the PyTorch library along with its C++-compatible library libtorch. This will allow us to work in both Python and C++, taking advantage of the strengths of each language. libtorch is also compatible with the CMake build tool.

Audio library

An audio library is a set of components that makes it easier to construct audio applications. Typical components are audio file readers and writers, audio device management and sound synthesis routines. We will use the JUCE audio library as it allows us to construct cross-platform audio applications and plugins. If you are not keen on using JUCE, you should be able to convert the applications we make to work in other audio libraries. For example, it is possible to create VST plugins directly using Steinberg's library, or you could use IPlug2. There are a few reasons I have chosen to use JUCE: it provides a very consistent experience on different platforms, it includes a set of cross-platform user interface components; it is compatible with the CMake build tool, and it can export plugins in several formats such as VST3, AudioUnit and so on. JUCE also provides components to build applications and plugins that can host other plugins, a capability we will make use of in some of the examples in the book.

Application Programming Interface (API)

An API is a set of ready-made components which you can use to develop applications. APIs come in different forms but the JUCE API contains C++ classes representing user interface components, audio file readers and so forth. See also the comments about libraries below.

Dynamic and static linking

When you build native applications, part of the process involves connecting your application somehow to the libraries you have used. There are two options here: dynamic linking and static linking. Static linking means the library (for example,

the audio library) is included inside your program's binary. Dynamic linking means the library exists as a separate file, and your application stores a reference to it. Depending on your platform, dynamically linked libraries are also called DLLs, shared objects and dylibs. When you run an application with dynamic links, those libraries must be located on the computer running the application. Depending on the operating system, the process of locating linked libraries varies. I will explain in more detail how to deal with this when you encounter it in the book.

Digital Audio Workstation (DAW)

A DAW is like an IDE for musicians. It is a collection of tools that work together to allow for the recording and production of music. DAWs support plugins – external software components that can be loaded and used for sound synthesis, effects processing and MIDI processing.

Plugin

A plugin is a software component that works inside a larger program. We will create plugins that work inside DAWs. The plugins we create will include machine learning capabilities.

Plugin host

A plugin host is any software that can load and use external plugins. DAWs are plugin hosts, but we will also use a simpler plugin host in some of our examples to help test our software and to allow our software to host and control plugins.

2.1.2 Computer hardware

Now you have learned about the development environment's main components, it is time for a brief discussion of computer hardware. I have used several computer systems to write this book and develop the cross-platform software within it. These include a Windows 10 Intel i7 machine with a discrete Nvidia Geforce 2070 graphics processor (GPU), an 'Apple Silicon' Mac with an M1 chip, an Intel Mac with an Intel i5 CPU, both running macOS and an Ubuntu Linux Intel machine with integrated Intel GPU. Do not worry if you cannot access as many different machines. You only need one machine to work through the book. But you will need access to those systems if you want to build versions of the programs for Windows, macOS and/ or Linux. You will also need administrative rights to install software on your development machine(s) unless it has already been installed for you.

GPUs, training and inference

If you have attempted to use a machine learning model before, you may have thought about the role of the GPU. For example, it is common for artificial intelligence researchers to publish executable Python notebooks with their research papers, and you may have experimented with one of these. Often notebooks are set up to run on cloud computing services such as Google colab[2]. When you run the notebook in the cloud, you will find various GPU options available in the user interface.

So why do machine learning systems use GPUs? This is an interesting question because using GPUs for machine learning was one of the breakthrough ideas that enabled the deep learning revolution to begin in the late noughties. Geoffrey Hinton, sometimes called the 'AI Godfather', presumably in the spiritual mentor as opposed to crime-boss sense was one of the pioneers in the use of GPUs for deep learning. As deep learning pioneer LeCun puts it, GPUs "were convenient to program and allowed researchers to train networks 10 or 20 times faster"[23]. In the same paper, the researchers mention that speech recognition was one of the first applications. So audio signal processing was a core application for deep learning from its inception.

So machine learning, particularly deep learning, runs faster on GPUs. Why is that, given that GPUs are actually designed to process graphics? The original purpose of a GPU is to compute parts of the graphics pipeline. There are several parts of the graphics pipeline requiring the multiplication of matrices. For example, computing lighting effects, applying textures and updating the positions of moving objects. GPUs have been designed to compute many matrix multiplications in parallel. Unlike CPUs, GPUs have many cores – the Geforce 2070 in my laptop has 2,304 cores, whereas the CPU has eight. GPU cores are simpler and more specialised than CPU cores. Essential parts of machine learning algorithms can be boiled down to many matrix multiplications, and GPUs excel at these.

At this point, you may worry that you need a powerful GPU to work through the practical activities in the book, but that is not necessarily the case. Firstly, not all machine learning algorithms in the book require heavy computation. As well as that, the machine learning algorithms in the book that do require heavy computation only need it in their training phase. Training is when the algorithm learns. Inference is when we use the model to carry out a task. Remember that one aim of the book is to build applications that music and audio professionals can access – well that means we need to make sure the inference part runs fast enough on a CPU. Still, it will be useful to have access to a GPU sometimes – so check out the information later about Google Colabs for cloud-based GPU use. I will also provide pre-trained models where appropriate.

[2] https://colab.google.com/

2.2 Setting up for C++ development

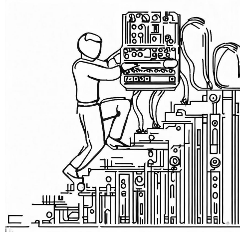


FIGURE 2.3
Setting up your development environment involves complex machinery and lots of steps.

At this point, you should be familiar with the purposes of the critical components you will encounter in your development environment. Now it is time to put that knowledge into practice by setting up the components on your machine(s) and attempting to build and run some programs.

I have taught programming for many years in several languages. My experience has been that getting that first program compiled and running can be quite a frustrating first step. The reason is that the programmer needs to engage with many complex tools and concepts, even though they have not written a line of code yet. This problem is especially true when working in C++ with an IDE. Even worse, we are working with two different libraries, JUCE and libtorch and potentially, cross-platform!

So, I want you, the reader, to know that the next few steps can go wrong and be quite frustrating, but stick with it and be prepared to search online for any error messages you encounter. Eventually, you will have all the tools humming away nicely. At the end of this section, I have created a list of common issues people encounter with the development environment setup process along with possible solutions.

If you already have an IDE installed and can compile and run C++ programs, it is still worth reading through this section, as a few special steps are required for the projects in this book. For example, you might need to become more familiar with JUCE, CMake or libtorch and how they work together.

As for all software setup and configuration instructions in the book, minor variations between operating systems and software versions might cause glitches, and things might break when new software versions are released. If you encounter any glitches, the best thing to do is to post an issue on the GitHub page for the course materials.

We will start with the essential tools that allow us to build software: an IDE and the tools for C++ development. What you do here depends on the operating system(s) and hardware you have available to you. Instructions are here for all three operating systems supported by the book. You can skip the sections for the operating systems you are not using.