Composition using Python and Abjad/LilyPond: Life Beyond Notation Software

Release v1.0.1

George K. Thiruvathukal

Apr 04, 2026

TECHNICAL REPORT

> **Warning**
>
> This site is about work in progress. Although these are compositions, they are related to my work in an ongoing composition class and should be viewed as sketches toward a bigger work.

This documentation tree is the working home for the technical report. The target format is an ArXiv-style paper, but the material is organized here first as Sphinx documents so the report can evolve in smaller pieces. For now, distribution of this report is via Figshare at doi.org/10.6084/m9.figshare.31827391

If this repository or the accompanying report is useful in your own work, please consider citing it.

```
@article{Thiruvathukal2026,
  author = "George K. Thiruvathukal",
  title = "{Composition using Python and Abjad/LilyPond: Life Beyond Notation Software}",
  year = "2026",
  month = "3",
  url = "https://figshare.com/articles/online_resource/Composition_using_Python_and_
↪Abjad_LilyPond_Life_Beyond_Notation_Software/31827391",
  doi = "10.6084/m9.figshare.31827391.v1"
}
```

CHAPTER

ONE

ABSTRACT

This report describes a programmatic composition environment built in Python with Abjad and LilyPond. The project grew out of a personal return to formal music study after many years centered on academic life in computer science. I have been on the faculty at Loyola University Chicago since 2001, and during my current service as department chairperson in computer science I also began pursuing postbaccalaureate study in music there. The immediate motivation for this repository was MUSC 246, a composition course taught by Dr. Dongryul Lee (Assistant Professor of Music and Composer). After struggling with WYSIWYG notation software, including Dorico, I turned to LilyPond and then to Python and Abjad as a better fit for the way I think and work. The repository now combines finished scores, reusable compositional utilities, and exploratory algorithmic studies in one reproducible workflow. Each score can be generated from source, engraved to PDF, exported to MIDI, and rendered to WAV with SoundFonts that provide a more realistic sense of performance than notation output alone. The same work can be built locally and in GitHub Actions, which makes the pipeline easier to test and publish. The project also places strong emphasis on sound software engineering: clear package structure, repeatable builds, documented configuration, automated releases, and technical documentation that explains how the system is put together. The main point of the project is not only to produce scores, but also to document a practical path from composition ideas to plain-text notation, code-driven generation, and reproducible outputs.

CHAPTER

TWO

INTRODUCTION

This repository began from a personal shift as much as a technical one. I am a professor of computer science at Loyola University Chicago and have been on the faculty there since 2001. After many years focused on teaching, research, and administration, including the intense work of serving (in my present life) as department chairperson, I returned to music study in a more serious way and began pursuing postbaccalaureate work in music at the same institution. This repository is one result of that return.

The immediate context was MUSC 246, a composition course at Loyola taught by Dongryul Lee, Assistant Professor of Music and Coordinator of Theory and Composition. The course covered a wide range of compositional styles and methods. That breadth was exciting, but it also raised a practical question: how should I write music in a way that fits both my musical goals and my habits as a computer scientist?

My first attempts used standard WYSIWYG notation software. Dorico is a strong system, and it clearly improves on Finale in many ways, but I still found myself fighting the interface. I wanted a workflow closer to plain text, version control, and reproducible builds. That led me first to LilyPond, which has a long history as a GNU project, and then to Python with Abjad as a way to generate LilyPond programmatically.

Most music software assumes that composition begins in a notation editor or a digital audio workstation. This project starts from a different assumption. It treats code as a first-class compositional medium. That choice changes both the workflow and the kinds of musical questions that can be asked. When a score is built in code, the same source can define pitch material, rhythm, instrumentation, notation details, rendering choices, and release artifacts. It also becomes much easier to reproduce a result, rerun it with controlled changes, or compare two closely related systems.

This repository was built around that idea. It uses Python, Abjad, and LilyPond to move from compositional logic to engraved score. It also uses shell scripts and GitHub Actions so that the same scores can be built locally and in continuous integration. The result is a repository that functions as both a set of compositions and a software system.

The artistic motivation shows most clearly in the quartet work. `Algo Rhythms Quartet No. 1` is a proof of concept for a larger planned system. `Algo Rhythms Quartet No. 2` begins from the same base but is allowed to change more freely so that new musical ideas can be tested without rewriting the original piece. This makes the repository useful in two ways at once. It preserves completed work, and it gives a controlled place for technical and musical experiments.

The goal of this report is to describe that combined system in a form closer to a technical paper than to a project README. The report focuses on architecture, generation methods, rendering, and release workflow. It also discusses each current score in enough detail to show how the musical ideas and the code design fit together.

2.1 About the Author

George K. Thiruvathukal, PhD is *Professor and Chairperson* in the Department of Computer Science at Loyola University Chicago. He is also a *Visiting Computer Scientist* at Argonne National Laboratory in the Leadership Computing Facility (ALCF). As of January 2026, he is a *Postbaccalaureate Student* and pianist in Music/Jazz at Loyola at Loyola University Chicago, hoping against all odds to become a serious jazz musician and composer. Wish him (me) luck!

For more information, please see gkt.sh.

CHAPTER

THREE

BACKGROUND AND TOOLING

The project uses Abjad[2] as the Python[1] layer for score construction. Abjad does not try to be a notation editor. Instead, it gives direct control over score objects such as staves, voices, measures, articulations, dynamics, tempo marks, and other notation features. That makes it well suited for work where the score is built from algorithms or data structures rather than from mouse-driven editing.

LilyPond[3] is used as the engraving backend. Abjad emits LilyPond source, and LilyPond then produces the engraved PDF and MIDI output. This separation is important. Python handles the compositional and structural side of the work, while LilyPond handles the engraving layer. That keeps the implementation clean and makes it easy to inspect the generated `.ly` files when debugging or refining notation.

Audio rendering is handled separately from score engraving. For the solo piano work, MIDI can be rendered with FluidSynth and the Salamander Grand Piano SoundFont[4]. For the quartet work, the render path is more involved because the best available piano and string sounds come from different SoundFonts. The system uses Salamander Grand Piano for the piano part and Aegean Symphonic Orchestra[5] for violin, viola, and cello. The quartet system therefore renders piano and strings separately and combines the results with `ffmpeg`. This gives better output than forcing all instruments through a single SoundFont, and it makes the generated WAV output sound closer to a plausible performance.

The broader tooling choice is pragmatic as opposed to academic:

- Python is the control layer.
- Abjad is the score API.
- LilyPond is the engraver.
- FluidSynth and `ffmpeg` fill in the audio path.
- The Unix/Linux shell and GitHub Actions provide local and CI automation.

The value comes from using them together in one repeatable composition workflow. This workflow may or may not work for you but works well for me as my present course expects music notation as part of the composition process.

[2] Abjad Project, Abjad documentation.
[1] Python Software Foundation, Python.
[3] LilyPond Developers, LilyPond music engraving.
[4] SFZ Instruments, Salamander Grand Piano.
[5] HED Sounds, Aegean Symphonic Orchestra.

CHAPTER

FOUR

SYSTEM ARCHITECTURE

The repository is organized as a multi-package Python project under `../src`. Each major score path is implemented as its own package with a CLI entry point. This is a better fit than a single monolithic script. It keeps each score family readable, allows each package to evolve at its own pace, and makes the build process consistent across the repository.

There are five active package paths. `modus_operandi_abjad` contains a fixed three-movement piano work. `jazz_rhythm` contains reusable rhythmic studies. `algorithmic_piano_quartet_no1` contains the first config-driven quartet generator. `algorithmic_piano_quartet_no2` contains the forked and more experimental second quartet generator. `algorithmic` is a placeholder package for future work. The package entry points are defined in `../pyproject.toml` and exposed both as `python -m ...` module invocations and as installed console scripts.

Within each score package, the code usually separates into three layers. The CLI layer handles arguments, output selection, and compilation. The generation layer defines musical material or event logic. The score layer turns that material into Abjad objects and then into LilyPond source. Quartet packages add a fourth layer for configuration loading and a fifth layer for SoundFont handling.

The architecture also separates stable work from exploratory work. `Algo Rhythms Quartet No. 1` stays fixed as a proof-of-concept score. `Algo Rhythms Quartet No. 2` is allowed to diverge. This is a software architecture decision, but it is also a compositional one. It lets the repository preserve finished work without blocking further musical experiments.

One small but important detail is the use of entry points instead of one-off scripts:

Listing 1: Console script entry points defined in the project metadata.

```
[project.scripts]
modus-operandi-abjad = "modus_operandi_abjad.cli:main"
jazz-rhythms = "jazz_rhythm.cli:main"
algorithmic-piano-quartet-no1 = "algorithmic_piano_quartet_no1.cli:main"
algorithmic-piano-quartet-no2 = "algorithmic_piano_quartet_no2.cli:main"
algorithmic = "algorithmic.cli:main"
```

CHAPTER

FIVE

COMMON PIPELINE

The common pipeline in this repository has five steps. First, a package defines musical content either directly in code or through a mix of code and configuration. Second, that material is assembled into Abjad objects. Third, the Abjad representation is serialized into LilyPond. Fourth, LilyPond compiles the score into PDF and MIDI. Fifth, optional audio rendering converts MIDI into WAV. This last step is not required for score production, but it is part of the normal build path for the finished works and the quartet studies.

The fixed-score packages and the generative packages share the same general output pipeline. They differ mainly in how the musical material is created. `modus_operandi_abjad` and parts of `jazz_rhythm` are more direct. The quartet packages are more explicitly generative. They load configuration, turn it into generation rules, create event streams on a quantized grid, and then convert those events into notation.

Reproducibility matters more in the quartet packages than in the fixed-score packages. The quartet generators expose a random seed through configuration. That means a result can be regenerated as long as the code and config stay the same. Output filenames can also include measures, tempo, seed, and timestamp, which makes it easy to track experimental runs.

The quartet audio path is a good example of the pipeline becoming more specialized when the musical needs require it. A single piano SoundFont is not enough for a quartet, and a single orchestral SoundFont does not give the best piano result. The system therefore renders the piano layer and the string layer separately and combines them afterward. The CLI is still simple from the outside, but the internal render path is more careful.

The end of the main quartet CLI shows this in two steps. First, it decides which outputs LilyPond needs to compile:

Listing 1: Output compilation decisions in the first quartet CLI.

```
    formats_to_compile = set()
    if args.pdf:
        formats_to_compile.add("pdf")
    if args.midi:
        formats_to_compile.add("midi")
    if args.wav:
        formats_to_compile.add("midi")

    if formats_to_compile:
        compile_lilypond(ly_path, args.output_dir, stem, formats_to_compile)
    elif args.ly:
        print("Done (--ly only, skipping LilyPond compilation).")
```

Then it decides how WAV rendering should happen once the MIDI file exists:

Listing 2: WAV rendering choices in the first quartet CLI.

```
    if args.wav:
        midi_path = os.path.join(args.output_dir, f"{stem}.midi")
        wav_path = os.path.join(args.output_dir, f"{stem}.wav")
        if args.soundfont:
            render_wav(
                midi_path=midi_path,
                wav_path=wav_path,
                soundfont_path=args.soundfont,
                sample_rate=config.render.sample_rate,
            )
        elif config.render.piano_soundfont and config.render.strings_soundfont:
            render_layered_wav(
                midi_path=midi_path,
                wav_path=wav_path,
                piano_soundfont_path=config.render.piano_soundfont,
                strings_soundfont_path=config.render.strings_soundfont,
                sample_rate=config.render.sample_rate,
            )
        elif config.render.soundfont:
            render_wav(
                midi_path=midi_path,
                wav_path=wav_path,
                soundfont_path=config.render.soundfont,
                sample_rate=config.render.sample_rate,
            )
        else:
            raise ValueError(
                "WAV rendering requires --soundfont, [render].soundfont, "
                "or both [render].piano_soundfont and [render].strings_soundfont."
            )
```

CHAPTER

SIX

CASE STUDY I: MODUS OPERANDI FOR PIANO

`Modus Operandi for Piano` is the most fixed composition in the repository. It is a three-movement solo piano work organized around three modes on F: Dorian, Phrygian, and Lydian. Each movement has a clear tempo, meter, and registral profile. The right hand carries the main line, and the left hand provides a slower accompaniment. Even though the score is produced with Abjad and LilyPond, the work itself is not built as an open-ended generator. It is closer to a score written in code than to a random composition system.

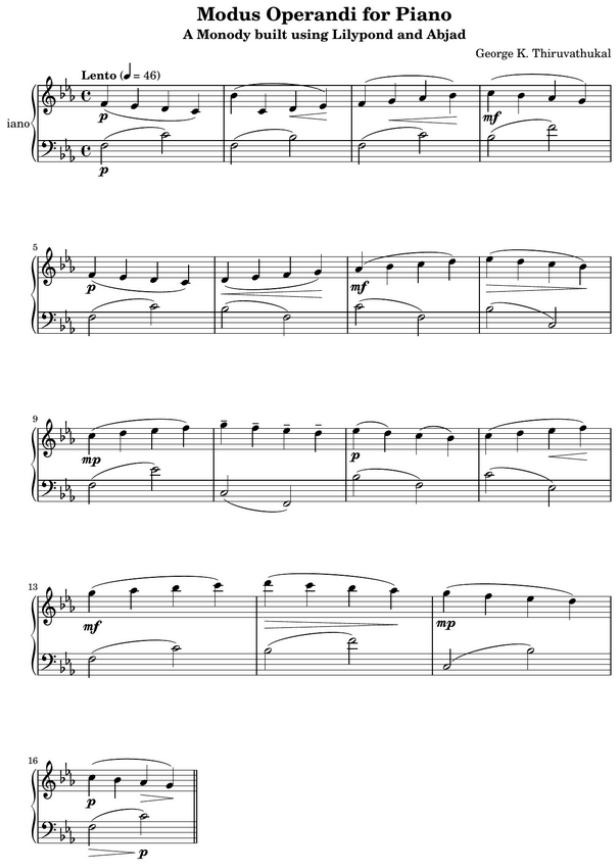
This package is important because it shows that the repository is not limited to generative experiments. It can also support a more traditional compositional process in which the musical material is already decided, but the score is still generated and rendered programmatically. That gives the project a wider scope. It is not only about algorithmic output. It is also about representing finished musical decisions cleanly in code.

The implementation reflects that role. The CLI writes a LilyPond file, compiles it, and then merges the per-movement MIDI files that LilyPond emits. This extra step is necessary because the piece is built from multiple score blocks. The WAV render is then handled through a separate script that downloads and caches a piano SoundFont on first use.

6.1 Download

Format	Link
PDF	modus-operandi-abjad.pdf
WAV	modus-operandi-abjad.wav

6.2 Score Preview



The movement-building code is direct enough to read almost like score notation. It is still too long to reproduce in full in the body of the paper, so only the opening of the movement is shown here:

Listing 1: Opening construction pattern for the first movement of Modus Operandi.

```
# ============================================================
# Movement I - F Dorian, 4/4, Lento
# ============================================================


def make_movement_i():
    # ---- Right Hand ----
    rh = abjad.Voice(name="RH_Voice")

    # Key, time, tempo, clef
    rh_notes = abjad.Container()

    # Bar 1: f4\p( ees d c)
    bar = abjad.Container("f'4 ef'4 d'4 c'4")
    abjad.attach(abjad.Clef("treble"), bar[0])
    abjad.attach(
```

(continues on next page)

(continued from previous page)

```
        abjad.KeySignature(abjad.NamedPitchClass("f"), abjad.Mode("dorian")), bar[0]
    )
    abjad.attach(abjad.TimeSignature((4, 4)), bar[0])
    abjad.attach(
        abjad.MetronomeMark(
            reference_duration=abjad.Duration(1, 4),
            units_per_minute=46,
            textual_indication="Lento",
        ),
        bar[0],
    )
    abjad.attach(abjad.Dynamic("p"), bar[0])
    abjad.attach(abjad.StartSlur(), bar[0])
    abjad.attach(abjad.StopSlur(), bar[3])
    rh_notes.append(bar)

    # Bar 2: bes4( c d\< ees\!)
    bar = abjad.Container("bf'4 c'4 d'4 ef'4")
    abjad.attach(abjad.StartSlur(), bar[0])
    abjad.attach(abjad.StartHairpin("<"), bar[2])
    abjad.attach(abjad.StopHairpin(), bar[3])
    abjad.attach(abjad.StopSlur(), bar[3])
    rh_notes.append(bar)

    # Bar 3: f4( g\< aes bes\!)
    bar = abjad.Container("f'4 g'4 af'4 bf'4")
    abjad.attach(abjad.StartSlur(), bar[0])
    abjad.attach(abjad.StartHairpin("<"), bar[1])
    abjad.attach(abjad.StopHairpin(), bar[3])
    abjad.attach(abjad.StopSlur(), bar[3])
    rh_notes.append(bar)
```

That directness is one of the useful contrasts in the report. `Modus Operandi` shows what the system looks like when the composition is fixed. The quartet packages show what the same repository looks like when the composition logic is open to controlled variation.

CHAPTER

SEVEN

CASE STUDY II: JAZZ RHYTHMIC PATTERNS

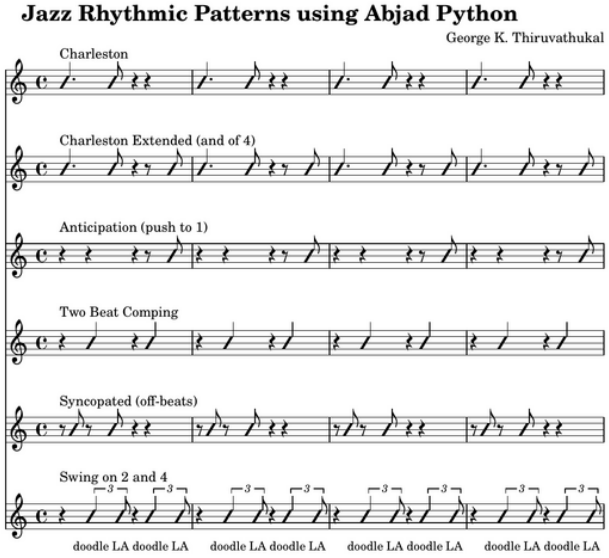
`Jazz Rhythmic Patterns` is structurally smaller than the other projects, but it is useful for a technical report because it shows a different kind of musical abstraction. Instead of generating a large score from many interlocking parameters, it defines a small library of one-measure rhythmic cells. Those cells can be repeated, rendered, and reused in other contexts.

This package works as a compositional vocabulary source. It also works as a simple example of how musical ideas can be encoded as reusable Python functions. Each pattern returns a short list of notes and rests. The score builder then places several staves one under another so the patterns can be compared visually. The notation is now closer to what a jazz player would expect in a rhythm chart: a normal five-line staff, slash noteheads, and hits placed on the middle line. That makes the page read less like abstract rhythmic study and more like practical ensemble notation.

7.1 Download

Format	Link
PDF	jazz-rhythms.pdf
WAV	jazz-rhythms.wav

7.2 Score Preview



The pattern functions are intentionally small:

Listing 1: One of the reusable rhythm cells in the jazz rhythm package.

```
def charleston():
    """Return a one-measure Charleston rhythm."""
    return [
        abjad.Note("b'4."),
        abjad.Note("b'8"),
        abjad.Rest("r4"),
        abjad.Rest("r4"),
    ]
```

The score layer is equally direct:

Listing 2: Assembly of the jazz rhythm comparison score.

```
def build_lilypond_file():
    """Build the LilyPond file for the jazz rhythm score."""
    score = abjad.Score([], name="Score")
    score.append(_make_staff("Charleston Staff", "Charleston", rhythms.charleston))
    score.append(
        _make_staff(
            "Charleston Extended Staff",
            "Charleston Extended (and of 4)",
            rhythms.charleston_extended,
        )
    )
    score.append(
        _make_staff(
            "Anticipation Staff",
            "Anticipation (push to 1)",
            rhythms.anticipation,
        )
    )
    score.append(_make_staff("Two Beat Staff", "Two Beat Comping", rhythms.two_beat))
    score.append(
        _make_staff(
            "Syncopated Staff",
            "Syncopated (off-beats)",
            rhythms.syncopated,
        )
    )
    score.append(
        _make_lyric_staff(
            "Swing Two Four Staff",
            "Swing on 2 and 4",
            rhythms.swing_two_four,
            "doodle LA doodle LA",
        )
    )

    header_block = abjad.Block(name="header")
    header_block.items.append(rf'title = "{TITLE}"')
    header_block.items.append(rf'composer = "{COMPOSER}"')
```

(continues on next page)

(continued from previous page)

```
    header_block.items.append(r"tagline = ##f")

    layout_block = abjad.Block(name="layout")
    layout_block.items.append(r"indent = 2.0\cm")
    layout_block.items.append(
        r"""
        \context {
            \Score
            \override VerticalAxisGroup.default-staff-staff-spacing.basic-distance = #14
            \override VerticalAxisGroup.default-staff-staff-spacing.minimum-distance =
↪#10
            \override VerticalAxisGroup.default-staff-staff-spacing.padding = #3
        }
        """
    )

    midi_block = abjad.Block(name="midi")

    score_block = abjad.Block(name="score")
    score_block.items.append(score)
    score_block.items.append(layout_block)
    score_block.items.append(midi_block)

    return abjad.LilyPondFile(items=[header_block, score_block])
```

This package shows that the repository does not only support finished scores and large generators. It also supports smaller compositional tools. It also suggests that presentation changes how the material is understood. The same rhythms read differently once they are shown in a chart-like notation style instead of a generic rhythm-only staff. The audio path now follows the same idea. Instead of leaving the rhythms silent, the package can render them as a simple clap track so the patterns can also be heard. In a technical report, that helps show the full range of the codebase.

CHAPTER

EIGHT

CASE STUDY III: ALGO RHYTHMS QUARTET NO. 1

`Algo Rhythms Quartet No. 1` is the first substantial config-driven score in the repository. Its role is clear in the project documentation: it is a proof of concept for a broader future composition system. The long-term direction is more general and eventually more post-tonal, but No. 1 is deliberately tonal. That choice simplifies the first experiment. It lets the project test rhythm, range handling, density control, engraving, and audio rendering before the harmonic language becomes more ambitious.

The score is defined through a TOML file and a generator package. The TOML file controls title, file naming, render settings, core generation constraints, pitch-class material, and instrumentation. The generator then creates quantized event streams for violin, viola, cello, and piano. The system enforces a small set of global rules: pitch choice must remain within a supplied pitch-class collection, melodic leaps are bounded, note and rest lengths are bounded, and the total sounding density is capped.

The quartet code relies on a small set of data classes. `PartConfig` describes one instrument entry from the TOML file. It carries the identifiers, instrument names, MIDI settings, and playable range. `GenerationConfig` holds the global generation rules such as measure count, durations, leap limits, and density limits. `ProjectConfig` is the top-level object passed through the package. It bundles title, output settings, pitch material, generation settings, render settings, and the list of parts.

The config loader is where these musical and technical settings become one internal project configuration:

Listing 1: Core quartet configuration data classes.

```
@dataclass(frozen=True)
class PartConfig:
    id: str
    name: str
    short_name: str
    family: str
    clef: str
    midi_channel: int
    midi_program: int
    midi_instrument: str
    range_low: int
    range_high: int
    staff_type: str = "single"
    role: str = "melodic"


@dataclass(frozen=True)
class GenerationConfig:
```

(continues on next page)

(continued from previous page)

```
    measures: int
    time_signature: tuple[int, int]
    min_note_quanta: int
    max_note_quanta: int
    min_rest_quanta: int
    max_rest_quanta: int
    max_simultaneous_tones_per_quantum: int
    max_pitch_leap: int
    seed: int
    tempo_bpm: int
    measure_quanta: int


@dataclass(frozen=True)
class RenderConfig:
    soundfont: str | None
    piano_soundfont: str | None
    strings_soundfont: str | None
    sample_rate: int


@dataclass(frozen=True)
class OutputConfig:
    basename: str
    label: str | None
    include_measures: bool
    include_tempo: bool
    include_seed: bool
    include_timestamp: bool
    timestamp_format: str
```

Listing 2: Loading the No. 1 configuration from TOML.

```


def _default_midi_instrument(part_id: str, family: str, staff_type: str) -> str:
    if staff_type == "grand":
        return "acoustic grand"
    if part_id in {"violin", "viola", "cello"}:
        return part_id
    if family == "keyboard":
        return "acoustic grand"
    return "acoustic grand"


def load_config(path: str | Path) -> ProjectConfig:
    config_path = Path(path)
    with config_path.open("rb") as file_pointer:
        data = tomllib.load(file_pointer)

    generation_data = data["generation"]
```

(continues on next page)

(continued from previous page)

```
    time_signature = tuple(generation_data.get("time_signature", [4, 4]))
    base_unit = Fraction(generation_data.get("min_note_duration", "1/16"))
    measure_length = Fraction(time_signature[0], time_signature[1])
    measure_quanta = _parse_duration_to_quanta(str(measure_length), base_unit)

    parts = []
    for entry in data["parts"]:
        pitch_range = entry["range"]
        parts.append(
            PartConfig(
                id=entry["id"],
                name=entry["name"],
                short_name=entry["short_name"],
                family=entry["family"],
                clef=entry.get("clef", "treble"),
                midi_channel=entry.get("midi_channel", 1),
                midi_program=entry.get("midi_program", 1),
                midi_instrument=entry.get(
                    "midi_instrument",
                    _default_midi_instrument(
                        entry["id"],
                        entry["family"],
                        entry.get("staff_type", "single"),
                    ),
                ),
                range_low=_parse_midi_pitch(pitch_range[0]),
                range_high=_parse_midi_pitch(pitch_range[1]),
                staff_type=entry.get("staff_type", "single"),
                role=entry.get("role", "melodic"),
            )
        )

    generation = GenerationConfig(
        measures=generation_data.get("measures", 4),
        time_signature=time_signature,
        min_note_quanta=_parse_duration_to_quanta(
            generation_data.get("min_note_duration", "1/16"),
            base_unit,
        ),
        max_note_quanta=_parse_duration_to_quanta(
            generation_data.get("max_note_duration", str(measure_length)),
            base_unit,
        ),
        min_rest_quanta=_parse_duration_to_quanta(
            generation_data.get("min_rest_duration", "1/16"),
            base_unit,
        ),
        max_rest_quanta=_parse_duration_to_quanta(
            generation_data.get("max_rest_duration", "1/4"),
            base_unit,
        ),
        max_simultaneous_tones_per_quantum=generation_data.get(
```

(continues on next page)

(continued from previous page)

```
            "max_simultaneous_tones_per_quantum", 4
        ),
        max_pitch_leap=generation_data.get("max_pitch_leap", 5),
        seed=generation_data.get("seed", 17),
        tempo_bpm=generation_data.get("tempo_bpm", 72),
        measure_quanta=measure_quanta,
    )

    materials = data.get("materials", {})
    pitch_classes = tuple(materials.get("pitch_classes", [0, 1, 3, 6, 8, 10]))
    render_data = data.get("render", {})

    output_data = data.get("output", {})

    return ProjectConfig(
        title=data.get("title", "Untitled Piano Quartet"),
        composer=data.get("composer", "Unknown Composer"),
        output=OutputConfig(
            basename=output_data.get("basename", "algo-rhythms-quartet-no-1"),
            label=output_data.get("label"),
            include_measures=output_data.get("include_measures", True),
            include_tempo=output_data.get("include_tempo", True),
```

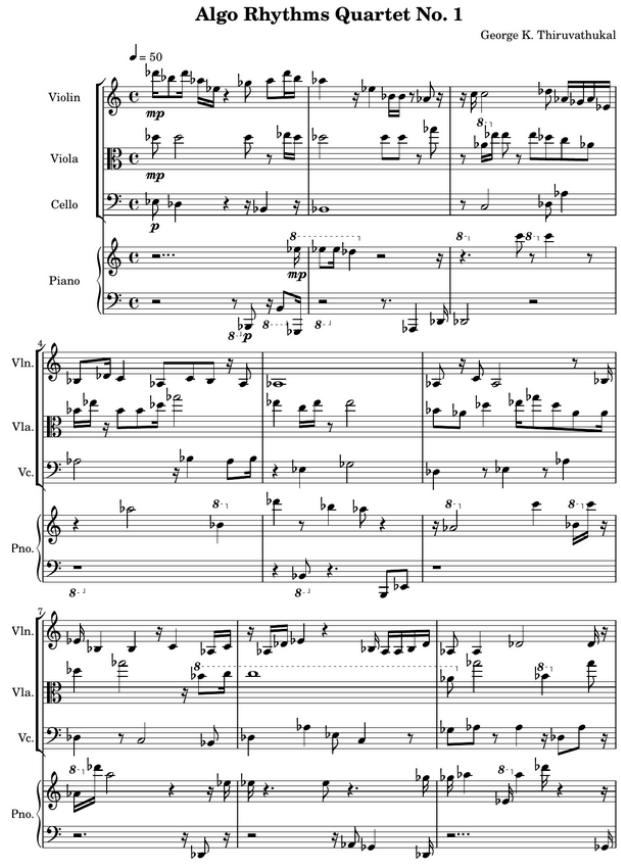
The musical content is not chosen all at once. It is built event by event on a quantized timeline. The generator uses a few core classes here as well. `Event` represents a single timed unit with a start position, duration, and pitch. `VoiceMaterial` is the list of events for one staff. `Piece` is the generated score-level result, including the metadata needed later by the score layer. These classes are simple containers, but they matter because they define the internal shape of the generated material.

The result is not strict counterpoint or formal harmony. It is a constrained event generator designed to produce clear, playable material.

8.1 Download

Format	Link
PDF	algo-rhythms-quartet-no-1.pdf
WAV	algo-rhythms-quartet-no-1.wav

8.2 Score Preview



That event loop sits near the center of the package:

Listing 3: Core event generation for Quartet No. 1.

```
        )
    elif config.render.soundfont:
        lines.append(f"Render: {_soundfont_name(config.render.soundfont)}")

    return tuple(lines)


def _generate_voice(
    staff_id: str,
    part_id: str,
    name: str,
    short_name: str,
    clef: str,
    midi_instrument: str,
    low: int,
    high: int,
    role: str,
    config: ProjectConfig,
    tracker: OccupancyTracker,
```

(continues on next page)

(continued from previous page)

```
    rng: random.Random,
) -> VoiceMaterial:
    total_quanta = config.generation.measures * config.generation.measure_quanta
    note_min = config.generation.min_note_quanta
    note_max = config.generation.max_note_quanta
    rest_min = config.generation.min_rest_quanta
    rest_max = config.generation.max_rest_quanta
    max_pitch_leap = config.generation.max_pitch_leap + (2 if role == "bass" else 0)
    anchor = (low + high) // 2
    last_pitch: int | None = None
    cursor = 0
    events: list[Event] = []

    while cursor < total_quanta:
        measure_offset = cursor % config.generation.measure_quanta
        measure_remaining = config.generation.measure_quanta - measure_offset
        current_density = tracker.count(cursor)
        choose_rest = current_density >= config.generation.max_simultaneous_tones_per_
↪quantum
        if not choose_rest:
            rest_probability = _rest_probability(
                role,
                current_density,
                config.generation.max_simultaneous_tones_per_quantum,
            )
            rest_probability += _ending_rest_boost(
                cursor=cursor,
                total_quanta=total_quanta,
                measure_quanta=config.generation.measure_quanta,
            )
            choose_rest = rng.random() < min(rest_probability, 0.95)

        if choose_rest:
            durations = _duration_candidates(role, rest_min, rest_max, measure_remaining)
            duration = rng.choice(durations)
            events.append(Event(start_quantum=cursor, duration_quanta=duration,␣
↪pitch=None))
            cursor += duration
            continue

        durations = _duration_candidates(role, note_min, note_max, measure_remaining)
        if role == "bass":
            durations = [value for value in durations if value >= 2] or durations
        duration = rng.choice(durations)
        if not tracker.can_place(cursor, duration):
            rest_duration = min(measure_remaining, max(rest_min, 1))
            events.append(Event(start_quantum=cursor, duration_quanta=rest_duration,␣
↪pitch=None))
            cursor += rest_duration
            continue

        candidates = _candidate_pitches(
```

(continues on next page)

(continued from previous page)

```
            low=low,
            high=high,
            pitch_classes=config.pitch_classes,
            anchor=anchor,
            last_pitch=last_pitch,
            max_pitch_leap=max_pitch_leap,
        )
        window = min(6, len(candidates))
        pitch = rng.choice(candidates[:window]) if candidates else anchor
        tracker.occupy(cursor, duration)
        events.append(Event(start_quantum=cursor, duration_quanta=duration, pitch=pitch))
        last_pitch = pitch
        cursor += duration

    return VoiceMaterial(
        staff_id=staff_id,
        part_id=part_id,
```

Once events exist, the score layer turns them into Abjad objects, attaches dynamics and notation details, applies ottava marks, and assembles the final ensemble score. The system also adds a short end note that records the main compositional parameters for the generated run.

No. 1 also matters as a baseline for the second quartet. It shows the original generator design before the later work on chordal piano writing, separate piano occupancy, and hand-specific spacing.

CHAPTER

NINE

CASE STUDY IV: ALGO RHYTHMS QUARTET NO. 2

`Algo Rhythms Quartet No. 2` exists because the repository needed a place to continue experimenting without destabilizing No. 1. From a software point of view, this is a forked package. From a musical point of view, it is a controlled branch for testing new behavior. The decision to fork rather than rewrite No. 1 in place is important. It keeps the first proof-of-concept score intact while opening a second path for larger technical and musical changes.

The first large difference is the piano. No. 1 mainly treats piano lines as single-note event streams. No. 2 allows the piano to generate chords. Those chords are still derived from the configured pitch-class material and the configured hand ranges, but they are shaped with more detail. The left and right hands can have different chord sizes, different preferred spacing, and different span limits.

The second large difference is the occupancy model. In No. 1, the piano competes with the strings under the same density cap. In No. 2, piano has its own occupancy budget. That gives the keyboard more room to act like a harmonic instrument rather than like two more melodic voices squeezed into the same global constraint.

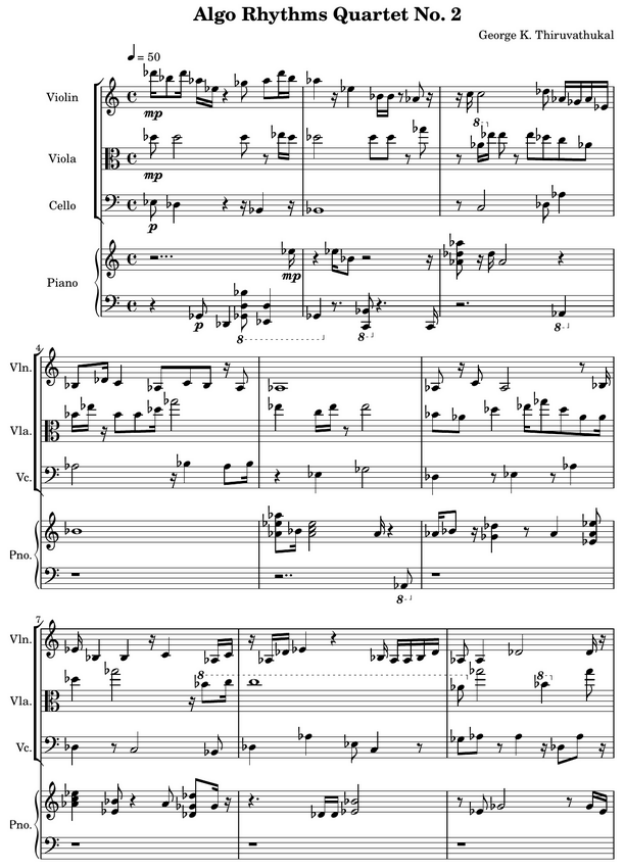
The third difference is the way left-hand spacing is treated. No. 2 now looks at whole chord shapes instead of simply growing a chord one pitch at a time from a seed. This makes it possible to prefer a wider overall left-hand span and to move away from narrow triadic shapes when the configuration asks for something more open.

The second quartet extends the same basic internal model as No. 1, but it expands the generation settings. The added fields define separate piano occupancy, separate left-hand and right-hand chord ranges, and separate preferred spacing rules. In other words, No. 2 does not invent a new architecture. It stretches the old one in a more piano-aware direction.

9.1 Download

Format	Link
PDF	algo-rhythms-quartet-no-2.pdf
WAV	algo-rhythms-quartet-no-2.wav

9.2 Score Preview



The config classes make that change easy to see. `PartConfig` still carries one instrument definition, including name, role, staff type, and range. `RenderConfig` still carries the SoundFont choices and sample rate. The main difference is `GenerationConfig`. In No. 2, this class carries the extra piano controls: separate occupancy for piano, separate chord-size limits for each hand, separate span limits, and separate preferred interval lists. Those fields are the link between the TOML file and the generator.

Listing 1: Core No. 2 configuration data classes.

```
@dataclass(frozen=True)
class PartConfig:
    id: str
    name: str
    short_name: str
    family: str
    clef: str
    midi_channel: int
    midi_program: int
    midi_instrument: str
    range_low: int
    range_high: int
    staff_type: str = "single"
    role: str = "melodic"
```

(continues on next page)

(continued from previous page)

```
@dataclass(frozen=True)
class GenerationConfig:
    measures: int
    time_signature: tuple[int, int]
    min_note_quanta: int
    max_note_quanta: int
    min_rest_quanta: int
    max_rest_quanta: int
    max_simultaneous_tones_per_quantum: int
    piano_max_simultaneous_events: int
    max_pitch_leap: int
    seed: int
    tempo_bpm: int
    measure_quanta: int
    piano_chord_probability: float
    piano_min_chord_tones: int
    piano_max_chord_tones: int
    piano_rh_min_chord_tones: int
    piano_rh_max_chord_tones: int
    piano_lh_min_chord_tones: int
    piano_lh_max_chord_tones: int
    piano_chord_span: int
    piano_rh_chord_span: int
    piano_lh_chord_span: int
    piano_rh_min_total_span: int
    piano_lh_min_total_span: int
    piano_preferred_chord_steps: tuple[int, ...]
    piano_rh_preferred_chord_steps: tuple[int, ...]
    piano_lh_preferred_chord_steps: tuple[int, ...]
    piano_min_chord_separation: int


@dataclass(frozen=True)
class RenderConfig:
    soundfont: str | None
    piano_soundfont: str | None
    strings_soundfont: str | None
    sample_rate: int


@dataclass(frozen=True)
class OutputConfig:
    basename: str
    label: str | None
    include_measures: bool
    include_tempo: bool
    include_seed: bool
    include_timestamp: bool
    timestamp_format: str
```

(continues on next page)

(continued from previous page)

```
@dataclass(frozen=True)
class ProjectConfig:
    title: str
    composer: str
    output: OutputConfig
    pitch_classes: tuple[int, ...]
    generation: GenerationConfig
    render: RenderConfig
    parts: tuple[PartConfig, ...]
```

The chord builder is a good example of that evolution. This function expects a seed pitch, hand range, pitch-class pool, requested chord size, span limits, preferred interval steps, and a minimum separation between adjacent notes. Given those inputs, it searches candidate chord shapes and keeps the result inside the hand range:

Listing 2: Chord construction in Quartet No. 2.

```
def _build_piano_chord(
    seed_pitch: int,
    low: int,
    high: int,
    pitch_classes: tuple[int, ...],
    minimum_tones: int,
    maximum_tones: int,
    max_span: int,
    minimum_total_span: int,
    preferred_steps: tuple[int, ...],
    minimum_separation: int,
    rng: random.Random,
) -> tuple[int, ...]:
    chord_low = max(low, seed_pitch - max_span)
    chord_high = min(high, seed_pitch + max_span)
    candidates = [
        pitch
        for pitch in range(chord_low, chord_high + 1)
        if pitch % 12 in pitch_classes and pitch != seed_pitch
    ]
    if not candidates:
        return (seed_pitch,)

    def interval_quality(pitch: int) -> tuple[int, int]:
        interval = abs(pitch - seed_pitch)
        nearest_preferred = min(abs(interval - step) for step in preferred_steps)
        return (nearest_preferred, interval)

    candidates.sort(key=lambda pitch: (*interval_quality(pitch), pitch))
    candidate_pool = [seed_pitch, *candidates[: min(len(candidates), 12)]]
    available_size = min(maximum_tones, len(candidate_pool))
    if available_size <= 1:
        return (seed_pitch,)

    target_size = rng.randint(minimum_tones, available_size)
```

(continues on next page)

(continued from previous page)

```
    valid_chords: list[tuple[tuple[int, ...], tuple[int, int, int]]] = []

    for combo in combinations(candidate_pool[1:], target_size - 1):
        chord = tuple(sorted((seed_pitch, *combo)))
        if any((upper - lower) < minimum_separation for lower, upper in zip(chord,␣
↪chord[1:])):
            continue
        total_span = chord[-1] - chord[0]
        if total_span > max_span:
            continue
        span_penalty = max(0, minimum_total_span - total_span)
        adjacent_penalty = sum(
            min(abs((upper - lower) - step) for step in preferred_steps)
            for lower, upper in zip(chord, chord[1:])
        )
        seed_penalty = sum(
            min(abs(abs(pitch - seed_pitch) - step) for step in preferred_steps)
            for pitch in chord
            if pitch != seed_pitch
        )
        valid_chords.append((chord, (span_penalty, adjacent_penalty + seed_penalty, -
↪total_span)))

    if valid_chords:
        valid_chords.sort(key=lambda item: item[1])
        best_score = valid_chords[0][1]
        top_chords = [chord for chord, score in valid_chords if score == best_score][:4]
        return rng.choice(top_chords)

    return (seed_pitch,)
```

No. 2 also has a larger configuration surface than No. 1 because it needs to expose these experiments clearly. That is a feature, not a flaw. The second quartet is the branch where new musical controls are tested. If a change proves useful and stable, it can later inform a more general future system.

CHAPTER

TEN

CASE STUDY V: ALGORITHMIC SCAFFOLD

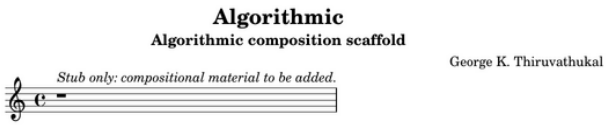
The `algorithmic` package is deliberately incomplete. It exists to hold the place for future work that has not yet become a full score package. In a technical report this may look minor, but it is worth including because it shows a normal pattern in research software. Infrastructure often appears before the final content that will use it.

Right now the package is only a stub. It can generate a placeholder score, run through the same CLI path as the other packages, and produce LilyPond, PDF, and MIDI outputs. That is enough to keep the project metadata, build script, and release workflow ready for future algorithmic material.

10.1 Download

Format	Link
PDF	algorithmic.pdf
WAV	Not published for this score

10.2 Score Preview



The placeholder score code is intentionally small:

Listing 1: Placeholder score generation in the algorithmic scaffold package.

```
def build_lilypond_file():
    """Build a placeholder LilyPond file for future development."""
    staff = abjad.Staff(name="Placeholder Staff")
    staff.append(abjad.Rest("r1"))

    first_leaf = abjad.select.leaf(staff, 0)
    abjad.attach(abjad.Clef("treble"), first_leaf)
    abjad.attach(abjad.TimeSignature((4, 4)), first_leaf)
    abjad.attach(
        abjad.Markup(
            r'\markup \italic "Stub only: compositional material to be added."'
        ),
```

(continues on next page)

(continued from previous page)

```
        first_leaf,
        direction=abjad.UP,
    )

    score = abjad.Score([staff], name="Score")

    header_block = abjad.Block(name="header")
    header_block.items.append(rf'title = "{TITLE}"')
    header_block.items.append(rf'subtitle = "{SUBTITLE}"')
    header_block.items.append(rf'composer = "{COMPOSER}"')
    header_block.items.append(r"tagline = ##f")

    layout_block = abjad.Block(name="layout")
    midi_block = abjad.Block(name="midi")

    score_block = abjad.Block(name="score")
    score_block.items.append(score)
    score_block.items.append(layout_block)
    score_block.items.append(midi_block)

    return abjad.LilyPondFile(items=[header_block, score_block])
```

This section shows that the report is not only about finished compositions. It is also about building a technical environment that can support future composition work cleanly.

CHAPTER

ELEVEN

CASE STUDY VI: BIRD IM-MIGRATION

The `bird_im_migration` package adapts a spectral-composition workflow into the larger `compositions-abjad` repository. It begins from a field recording of birds and a SPEAR partial-tracking analysis of that recording. The compositional goal is to reduce bird-like spectral behavior into performable notation while preserving a visible link to the source analysis.

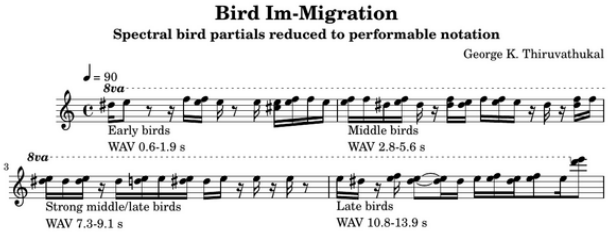
Unlike the more abstract or algorithmic case studies in this repository, this package starts with a fixed audio analysis file and then applies musical reduction on top of it. That makes it a useful counterexample in the book because the score does not begin from a pitch set or a generative procedure. It begins from tracked partials in a real sound recording.

11.1 Download

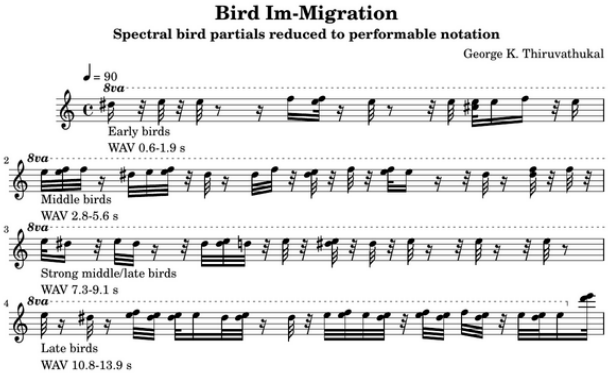
Quantization	Format	Link
q16	PDF	bird-im-migration-q16.pdf
q16	WAV	bird-im-migration-q16.wav
q32	PDF	bird-im-migration-q32.pdf
q32	WAV	bird-im-migration-q32.wav

11.2 Score Preview

11.2.1 q16 - 16th notes



11.2.2 q32 - 32nd notes



11.3 Source Materials

The package keeps its source materials inside the package data directory.

- `src/bird_im_migration/data/DL_parkbirds.wav`
- `src/bird_im_migration/data/DL_parkbirds_partials.txt`

The WAV file is the original recording. The partials text file is the SPEAR analysis export that drives the score generation.

11.4 How We Analyze the Spectrum

The analysis code focuses on a bird-like high-frequency band around `4.5-6.0 kHz`. That band is not meant to be a universal description of birdsong. It is a practical filter for isolating the recurring chirp-like material in this particular recording.

The package keeps two layers of interpretation. It can infer approximate candidate regions from the partials automatically. It also preserves a curated set of bird regions that serve as the score timeline.

The curated regions for this recording are:

- `Early birds`: `0.6-1.9 s`
- `Middle birds`: `2.8-5.6 s`
- `Strong middle/late birds`: `7.3-9.1 s`
- `Late birds`: `10.8-13.9 s`

11.5 How We Generate the Music

The package parses the SPEAR text file, filters for bird-like partials, quantizes onset material within each curated region, reduces each rhythmic bin to one or two salient pitches, and groups repeated bins into notated events. Because the underlying sound is spectrally complex, the resulting notation sometimes uses small pitch clusters rather than a single line.

The package currently supports at least two useful quantization settings.

- `q16` for a 16th-note reduction (more suitable for humans)
- `q32` for a 32nd-note reduction (more suitable for computers/synthesizers)

Those quantization choices appear directly in the generated output stems so that both versions can coexist in the build directory:

11.6 Implementation

The analysis pipeline lives in `bird_im_migration.analysis`. The Abjad score builder lives in `bird_im_migration.score`. The command-line entry point lives in `bird_im_migration.cli`.

The analysis module defines the spectral parsing and reduction logic:

Listing 1: Quantizing bird-like partials into notated pitch bins.

```
def quantize_region_pitches(
    partials: list[Partial],
    region: Region,
    *,
    bins_per_measure: int,
    min_frequency: float = 4500.0,
    max_frequency: float = 6000.0,
    min_amplitude: float = 0.02,
    max_notes_per_bin: int = 2,
) -> list[tuple[str, ...]]:
    buckets: list[dict[str, float]] = [dict() for _ in range(bins_per_measure)]
    span = region.end - region.start
    if span <= 0:
        return [tuple() for _ in range(bins_per_measure)]
    for partial in partials:
        if not (region.start <= partial.start <= region.end):
            continue
        if not (min_frequency <= partial.peak_frequency <= max_frequency):
            continue
        if partial.peak_amplitude < min_amplitude:
            continue
        position = (partial.start - region.start) / span
        bucket_index = min(bins_per_measure - 1, int(position * bins_per_measure))
        midi_number = round(69 + 12 * math.log2(partial.peak_frequency / 440.0))
        note_name = midi_to_note_name(midi_number)
        current = buckets[bucket_index].get(note_name, 0.0)
        if partial.peak_amplitude > current:
            buckets[bucket_index][note_name] = partial.peak_amplitude
    result: list[tuple[str, ...]] = []
    for bucket in buckets:
        top = sorted(bucket.items(), key=lambda item: item[1], reverse=True)[:max_notes_
↪per_bin]
        result.append(tuple(note for note, _ in top))
    return result
```

The score module turns those reduced events into an Abjad score with region labels and score metadata:

Listing 2: Building the Bird Im-Migration LilyPond file from the partials analysis.

```
def build_lilypond_file(
    *,
```

(continues on next page)

(continued from previous page)

```
    partials_path: str | Path = DEFAULT_PARTIALS_PATH,
    quantization: int = 32,
    tempo_bpm: int = 90,
    midi_instrument: str = "acoustic grand",
):
    partials = parse_spear_partials(partials_path)
    score = build_score(
        partials,
        quantization=quantization,
        tempo_bpm=tempo_bpm,
        midi_instrument=midi_instrument,
    )

    header_block = abjad.Block(name="header")
    header_block.items.append(rf'title = "{TITLE}"')
    header_block.items.append(rf'subtitle = "{SUBTITLE}"')
    header_block.items.append(rf'composer = "{COMPOSER}"')
    header_block.items.append(r"tagline = ##f")

    layout_block = abjad.Block(name="layout")
    layout_block.items.append(
        r"""
\context {
  \Staff
  ottavation = "8va"
}
""".strip()
    )
    midi_block = abjad.Block(name="midi")

    score_block = abjad.Block(name="score")
    score_block.items.append(score)
    score_block.items.append(layout_block)
    score_block.items.append(midi_block)
    return abjad.LilyPondFile(items=[header_block, score_block])
```

11.7 Build and Use

The repository build script now generates both quantized Bird Im-Migration variants alongside the other case studies. The package can also be invoked directly:

```
python -m bird_im_migration -o build --quantization 16 --pdf --midi
python -m bird_im_migration -o build --quantization 32 --pdf --midi
```

This case study shows that the Abjad approach can also absorb analysis-driven material rather than only symbolic or algorithmically invented content. It expands the technical report from score construction alone into the broader territory of spectral reduction and transcription as compositional method.

CHAPTER

TWELVE

CASE STUDY VII: BIRD IM-MIGRATION ENSEMBLE

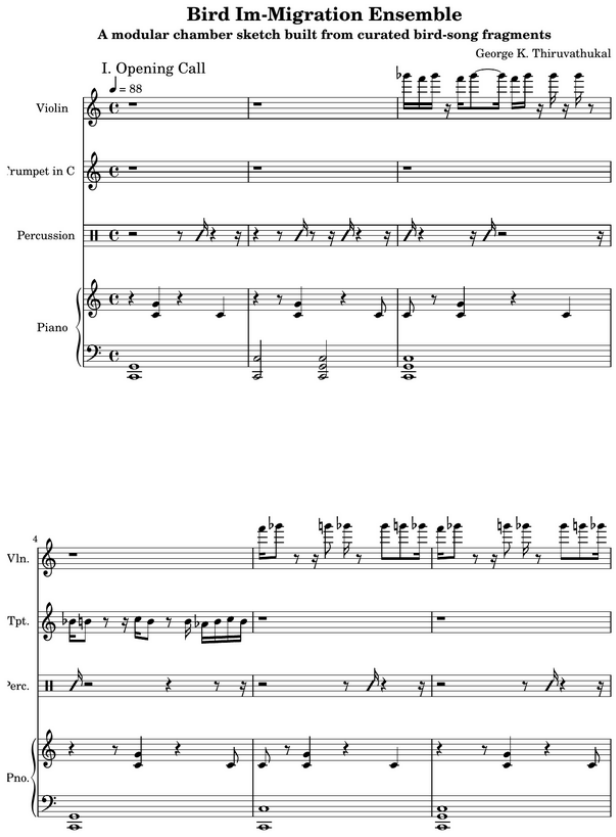
`bird_im_migration_ensemble` takes the analysis-driven material from *Case Study VI: Bird Im-Migration* and turns it into a short chamber piece. The earlier package is a proof of concept for reducing bird-like partial activity into playable notation. The ensemble package keeps that reduction as its source library, but it stops trying to remain purely spectral. Instead, it treats the bird fragments as modular motives and places them inside a more obviously composed environment: alternating treble calls, a low piano drone with patterned variation, and percussion that behaves like an environmental pulse rather than a literal transcription of the recording.

This case study shows a second stage of the same project. The spectral analysis is still there. The curated regions are still there. But the composition is no longer only a reduction of recorded sound. It becomes a score system that can support performance, transformation, substitution of instruments, and movement-level formal planning.

12.1 Download

Format	Link
PDF	bird-im-migration-ensemble.pdf
WAV	bird-im-migration-ensemble.wav
Movement I WAV	bird-im-migration-ensemble-mvt1.wav
Movement II WAV	bird-im-migration-ensemble-mvt2.wav
Movement III WAV	bird-im-migration-ensemble-mvt3.wav

12.2 Score Preview



12.3 From Spectral Reduction to Modular Phrases

The raw material is still the same field recording and the same curated regions introduced in the original Bird Im-Migration chapter. The difference is that the ensemble package does not treat those regions as a finished score. It converts them into a phrase library. Each phrase remembers which sample and which curated region it came from. This keeps later transformations attached to a recognizable source motive rather than turning them into anonymous note sequences.

The core data structures and the default movement plans live together in the generator module:

Listing 1: Phrase and movement data classes plus the default three-movement plan.

```
DEFAULT_SOURCE = BirdSource(
    sample_id="parkbirds",
    partials_path=str(DEFAULT_PARTIALS_PATH),
    curated_regions=CURATED_BIRD_REGIONS,
)


DEFAULT_MOVEMENTS: tuple[MovementConfig, ...] = (
    MovementConfig(
        number=1,
```

(continues on next page)

(continued from previous page)

```
        title="I. Opening Call",
        subtitle="Bird-call fragments introduced in measured exchange",
        time_signature=(4, 4),
        tempo_bpm=88,
        key_literal=r"\key d \dorian",
        center_midi=50,
        total_measures=32,
        phrase_pairs=6,
        intro_measures=2,
        closing_measures=3,
        phrase_measures=1,
        call_transforms=("identity", "repeat"),
        response_transforms=("identity", "retrograde"),
        call_response_probability=0.95,
        percussion_density=0.18,
        percussion_pattern="son-clave",
        piano_pattern="two-beat",
        seed_offset=0,
    ),
    MovementConfig(
        number=2,
        title="II. Echo / Nocturne",
        subtitle="Slower transformation and expanded time",
        time_signature=(3, 4),
        tempo_bpm=57,
        key_literal=r"\key b \minor",
        center_midi=47,
        total_measures=32,
        phrase_pairs=6,
        intro_measures=1,
        closing_measures=3,
        phrase_measures=1,
        call_transforms=("augment", "identity"),
        response_transforms=("retrograde", "augment"),
        call_response_probability=0.9,
        percussion_density=0.12,
        percussion_pattern="habanera-wave",
        piano_pattern="charleston",
        seed_offset=29,
    ),
    MovementConfig(
        number=3,
        title="III. Last Calls",
        subtitle="Layered returns and controlled closure",
        time_signature=(5, 4),
        tempo_bpm=108,
        key_literal=r"\key d \minor",
        center_midi=50,
        total_measures=32,
        phrase_pairs=7,
        intro_measures=2,
        closing_measures=3,
```

(continues on next page)

(continued from previous page)

```
        phrase_measures=1,
        call_transforms=("identity", "retrograde", "repeat"),
        response_transforms=("identity", "retrograde", "augment"),
        call_response_probability=1.0,
        percussion_density=0.28,
        percussion_pattern="cascara-light",
        piano_pattern="syncopated",
        seed_offset=71,
    ),
)


PERCUSSION_PATTERNS: dict[str, tuple[int, ...]] = {
    "son-clave": (0, 3, 6, 10, 12, 19, 22, 26),
    "habanera-wave": (0, 3, 4, 6, 8, 12, 15, 16, 18, 20, 24, 27, 28, 30),
    "cascara-light": (0, 2, 5, 7, 8, 11, 12, 14, 17, 19, 20, 23, 24, 26, 29, 31),
}


PIANO_PATTERNS: dict[str, tuple[int, ...]] = {
    "two-beat": (4, 12),
    "charleston": (0, 6),
    "anticipation": (14,),
    "syncopated": (2, 6),
}
```

In practice this means the composition can be described at the level of phrase behavior rather than note-by-note editing. The movement configuration chooses tempo, meter, pitch center, phrase count, percussion density, and the allowed transformations for calls and responses. That is the main difference between this package and the earlier spectral reduction package. The new piece is still derived from the bird analysis, but it is shaped as a parametric form.

12.4 Phrase Extraction and Transformation

The phrase library is built directly from the curated spectral regions. Each region is quantized onto a 16th-note grid and stored as one modular phrase object. From there, the package creates variants using a deliberately small set of operations: identity, retrograde, augmentation, and repetition.

Listing 2: Converting curated regions into phrase objects and creating transformed phrase variants.

```
def _make_variant(
    phrase: Phrase,
    transform_name: str,
    *,
    measure_units: int,
    phrase_measures: int,
) -> PhraseVariant:
    if transform_name == "identity":
        span_measures = phrase_measures
        bins = _resample_bins(phrase.note_bins, target_units=measure_units * span_
↪measures)
    elif transform_name == "retrograde":
```

(continues on next page)

(continued from previous page)

```
        span_measures = phrase_measures
        bins = _resample_bins(
            _retrograde_bins(phrase.note_bins),
            target_units=measure_units * span_measures,
        )
    elif transform_name == "augment":
        span_measures = phrase_measures * 2
        bins = _resample_bins(phrase.note_bins, target_units=measure_units * span_
↪measures)
    elif transform_name == "repeat":
        span_measures = phrase_measures * 2
        bins = _repeat_bins(phrase.note_bins, target_units=measure_units * span_measures)
    else:
        raise ValueError(f"Unknown transform: {transform_name}")
    return PhraseVariant(
        phrase=phrase,
        transform_name=transform_name,
        note_bins=bins,
        span_measures=span_measures,
    )


def build_phrase_library(
    *,
    sources: Iterable[BirdSource] = (DEFAULT_SOURCE,),
    bins_per_phrase: int = 16,
) -> tuple[Phrase, ...]:
    phrases: list[Phrase] = []
    for source in sources:
        partials = parse_spear_partials(source.partials_path)
        for region_name, region in source.curated_regions:
            note_bins = tuple(
                quantize_region_pitches(partials, region, bins_per_measure=bins_per_
↪phrase)
            )
            phrase_id = f"{source.sample_id}:{region_name.lower().replace(' ', '-')}"
            phrases.append(
                Phrase(
                    phrase_id=phrase_id,
                    sample_id=source.sample_id,
                    region_name=region_name,
                    note_bins=note_bins,
                )
            )
    return tuple(phrases)
```

This is the central compositional decision in the package. The score does not ask the spectral analysis to solve the whole piece. Instead, the analysis contributes a motive bank. The movement planner then decides whether a phrase should be stated plainly, reversed, stretched, or repeated. That keeps the birdsong recognizable but also lets the piece behave like chamber music rather than a transcription exercise.

12.5 How the Bird Lines Are Created

The violin and trumpet lines are built from the same pool of phrase variants. The first excerpt shows the call-and-response loop itself. The generator chooses a phrase for the call, maps it into the violin register, and then optionally answers it with a transformed phrase in the trumpet register. The mapping is intentionally asymmetric: the higher line prefers the upper note in each bin, while the trumpet response sits lower and answers rather than duplicates.

Listing 3: Call-and-response writing for violin and trumpet.

```
    while total_measures < target_phrase_measures:
        call_phrase = rng.choice(phrases)
        remaining_before_closing = target_phrase_measures - total_measures
        call_transform = rng.choice(config.call_transforms)
        call_variant = _make_variant(
            call_phrase,
            call_transform,
            measure_units=measure_units,
            phrase_measures=config.phrase_measures,
        )
        if call_variant.span_measures > remaining_before_closing:
            call_variant = _make_variant(
                call_phrase,
                "identity",
                measure_units=measure_units,
                phrase_measures=min(config.phrase_measures, remaining_before_closing),
            )
        last_variant = call_variant
        call_bins = [
            _map_bin_to_instrument(
                note_bin,
                center_midi=74,
                low=72,
                high=91,
                max_notes=1,
                prefer_highest=True,
            )
            or None
            for note_bin in call_variant.note_bins
        ]
        whistle_bins.extend(call_bins)
        trumpet_bins.extend([None] * len(call_bins))
        total_measures += call_variant.span_measures

        remaining_before_closing = target_phrase_measures - total_measures
        if remaining_before_closing <= 0:
            break

        if rng.random() <= config.call_response_probability:
            response_phrase = rng.choice(phrases)
            response_variant = _make_variant(
                response_phrase,
                rng.choice(config.response_transforms),
                measure_units=measure_units,
```

(continues on next page)

(continued from previous page)

```
                phrase_measures=config.phrase_measures,
            )
            if response_variant.span_measures > remaining_before_closing:
                response_variant = _make_variant(
                    response_phrase,
                    "identity",
                    measure_units=measure_units,
                    phrase_measures=min(config.phrase_measures, remaining_before_
↪closing),
                )
            last_variant = response_variant
            response_bins = [
                _map_bin_to_instrument(
                    note_bin,
                    center_midi=67,
                    low=55,
                    high=79,
                    max_notes=1,
                    prefer_highest=False,
                )
                or None
                for note_bin in response_variant.note_bins
            ]
            whistle_bins.extend([None] * len(response_bins))
            trumpet_bins.extend(response_bins)
            total_measures += response_variant.span_measures
```

The second excerpt shows how those generated bins become concrete instrument parts at the end of the movement build:

Listing 4: Assembling the movement parts after the bins have been generated.

```
    measures = config.total_measures
    parts = (
        InstrumentPart(
            staff_id="violin",
            name="Violin",
            short_name="Vln.",
            clef="treble",
            midi_instrument="violin",
            events=tuple(_phrase_bins_to_events(whistle_bins)),
        ),
        InstrumentPart(
            staff_id="trumpet",
            name="Trumpet in C",
            short_name="Tpt.",
            clef="treble",
            midi_instrument="trumpet",
            events=tuple(_phrase_bins_to_events(trumpet_bins)),
        ),
        InstrumentPart(
```

(continues on next page)

(continued from previous page)

```
            staff_id="percussion",
            name="Percussion",
            short_name="Perc.",
            clef="percussion",
            midi_instrument="woodblock",
            events=tuple(_phrase_bins_to_events(percussion_bins)),
        ),
        InstrumentPart(
            staff_id="piano_rh",
            name="Piano",
            short_name="Pno.",
            clef="treble",
            midi_instrument="acoustic grand",
            events=tuple(_phrase_bins_to_events(piano_rh_bins)),
        ),
        InstrumentPart(
            staff_id="piano_lh",
            name="Piano",
            short_name="Pno.",
            clef="bass",
            midi_instrument="acoustic grand",
            events=tuple(_phrase_bins_to_events(piano_lh_bins)),
        ),
    )
```

This is where the alternating bird effect comes from. One line states a phrase and the other line responds in the next span. Because calls and responses draw from overlapping but not identical transformation sets, the exchange can sound like imitation, transformation, or recollection. If violin or trumpet are unavailable in performance, the same lines could be reassigned to voice, whistle, or another treble instrument without changing the underlying phrase logic. The system is therefore instrument-specific in engraving, but not conceptually tied to one only possible ensemble.

12.6 Environmental Layers: Piano and Percussion

The environmental effect in this piece is not produced by spectral fidelity alone. It comes from adding non-spectral layers that support the bird material without trying to mimic it exactly. The piano is the clearest example. The left hand provides the low drone and harmonic floor. The right hand adds a lighter, pattern-based comping layer on an eighth-note grid. Movement II is handled differently again, with a two-measure arpeggiated left-hand cycle to make the nocturne character more audible.

Listing 5: Building the piano layers, with sustained drone behavior, right-hand patterns, and the nocturne arpeggiation.

```
def _build_piano_bins(
    config: MovementConfig,
    *,
    measures: int,
    measure_units: int,
) -> tuple[list[tuple[int, ...] | None], list[tuple[int, ...] | None]]:
    root = config.center_midi - 14
    fifth = root + 7
    octave = root + 12
    shimmer_root = config.center_midi + 10
```

(continues on next page)

(continued from previous page)

```
    shimmer_fifth = shimmer_root + 7
    shimmer_ninth = shimmer_root + 14
    shimmer_upper = shimmer_root + 12
    total_units = measures * measure_units
    left_bins: list[tuple[int, ...] | None] = [None] * total_units
    right_bins: list[tuple[int, ...] | None] = [None] * total_units
    eighth_units = 2
    half_measure = max(eighth_units, measure_units // 2)

    if config.piano_pattern == "two-beat":
        right_entries = (
            (max(eighth_units, measure_units // 4), (shimmer_root, shimmer_fifth), 2),
            (max(eighth_units, (measure_units * 3) // 4), (shimmer_root,), 2),
        )
    elif config.piano_pattern == "charleston":
        right_entries = (
            (0, (shimmer_root, shimmer_fifth), 3),
            (max(eighth_units, (measure_units * 3) // 8), (shimmer_root,), 1),
        )
    elif config.piano_pattern == "anticipation":
        right_entries = (
            (max(eighth_units, measure_units - 2), (shimmer_fifth, shimmer_ninth), 2),
        )
    elif config.piano_pattern == "syncopated":
        right_entries = (
            (max(eighth_units, measure_units // 8), (shimmer_root,), 2),
            (max(eighth_units, (measure_units * 3) // 8), (shimmer_fifth, shimmer_ninth),
↪ 2),
        )
    else:
        raise ValueError(f"Unknown piano pattern: {config.piano_pattern}")

    for measure_index in range(measures):
        measure_start = measure_index * measure_units
        in_closing = measure_index >= measures - config.closing_measures
        if in_closing:
            _apply_span(
                left_bins,
                start=measure_start,
                duration=measure_units,
                pitches=(root, fifth, octave),
            )
            _apply_span(
                right_bins,
                start=measure_start,
                duration=half_measure,
                pitches=(shimmer_root, shimmer_fifth),
            )
            _apply_span(
                right_bins,
                start=measure_start + half_measure,
                duration=half_measure,
```

(continues on next page)

(continued from previous page)

```
                pitches=(shimmer_root,),
            )
            continue

        if config.number == 2:
            two_measure_phase = measure_index % 2
            arpeggio_cycle = (
                (0, (root,)),
                (2, (fifth,)),
                (4, (octave,)),
                (6, (fifth,)),
                (8, (root + 12,)),
                (10, (fifth,)),
            )
            shifted_cycle = (
                (0, (root, fifth)),
                (2, (octave,)),
                (4, (fifth,)),
                (6, (root,)),
                (8, (fifth, octave)),
                (10, (fifth,)),
            )
            cycle = arpeggio_cycle if two_measure_phase == 0 else shifted_cycle
            for offset, chord in cycle:
                _apply_span(
                    left_bins,
                    start=measure_start + min(offset, max(0, measure_units - eighth_
↪units)),
                    duration=eighth_units,
                    pitches=chord,
                )
        else:
            if measure_index % 3 == 0:
                left_chord = (root, fifth)
            elif measure_index % 3 == 1:
                left_chord = (root, octave)
            else:
                left_chord = (root, fifth, octave)
            _apply_span(
                left_bins,
                start=measure_start,
                duration=measure_units,
                pitches=left_chord,
            )
            if measure_index % 2 == 1:
                _apply_span(
                    left_bins,
                    start=measure_start + half_measure,
                    duration=half_measure,
                    pitches=(root, fifth),
                )
```

(continues on next page)

(continued from previous page)

```
        shifted_entries = [
            (
                min(measure_units - eighth_units, start + (eighth_units if measure_index
↪% 2 == 1 else 0)),
                chord,
                duration_eighths,
            )
            for start, chord, duration_eighths in right_entries
        ]
        for start, chord, duration_eighths in shifted_entries:
            _apply_span(
                right_bins,
                start=measure_start + start,
                duration=max(eighth_units, duration_eighths * eighth_units),
                pitches=chord,
            )
    return left_bins, right_bins
```

The percussion line is also environmental rather than imitative. It does not try to duplicate every attack in the birdsong. Instead, it uses named rhythmic patterns and then thins or accents them according to what the bird lines are doing. That gives the texture a sense of place: not literal forest noise, but a patterned pulse that can suggest environment, motion, or distance.

Listing 6: Turning named rhythmic patterns into a light environmental percussion line.

```
def _build_percussion_bins_from_pattern(
    *,
    whistle_bins: list[tuple[int, ...] | None],
    trumpet_bins: list[tuple[int, ...] | None],
    measures: int,
    measure_units: int,
    closing_measures: int,
    pattern_name: str,
    density: float,
    rng: random.Random,
) -> list[tuple[int, ...] | None]:
    bins: list[tuple[int, ...] | None] = []
    cycle_units = measure_units * 2
    pattern_hits = _scaled_pattern_hits(pattern_name, cycle_units)
    previous_active = False
    for index, (whistle_bin, trumpet_bin) in enumerate(zip(whistle_bins, trumpet_bins)):
        measure_index = index // measure_units
        unit_index = index % measure_units
        cycle_index = index % cycle_units
        active = whistle_bin is not None or trumpet_bin is not None
        in_closing = measure_index >= measures - closing_measures
        pattern_hit = cycle_index in pattern_hits and unit_index % 2 == 0
        answer_hit = measure_index % 2 == 1 and unit_index == max(2, measure_units // 2)
        entry_accent = active and not previous_active and rng.random() <= min(1.0,␣
↪density + 0.15)
        if in_closing:
```

(continues on next page)

(continued from previous page)

```
            bins.append((60,) if unit_index in (0, max(1, measure_units // 2)) else None)
        elif pattern_hit and rng.random() <= min(1.0, density + 0.25):
            bins.append((60,))
        elif answer_hit and active and rng.random() <= density + 0.1:
            bins.append((60,))
        elif entry_accent:
            bins.append((60,))
        else:
            bins.append(None)
        previous_active = active
    return bins
```

This added material is why the piece is not simply a spectral reduction with accompaniment. The ensemble code accepts that a playable composition can preserve the bird motive while also introducing supporting musical layers that are chosen for form, playability, and atmosphere.

12.7 How the Piece Is Rendered

The package also treats rendering as part of the composition system. Notation is produced in LilyPond and then the audio path separates the ensemble into layers before synthesis. The first excerpt shows the layered render for one movement. It uses Salamander for the two piano hands, Aegean for the melodic ensemble, and a clap-style render for percussion.

Listing 7: Layered WAV rendering for one movement of Bird Im-Migration Ensemble.

```
def render_layered_wav_for_midi(
    midi_path: str,
    wav_path: str,
    piano_soundfont_path: str,
    ensemble_soundfont_path: str,
    sample_rate: int,
) -> None:
    midi = mido.MidiFile(midi_path)
    piano_rh_channels = _channels_for_prefixes(midi, {"piano_rh"})
    piano_lh_channels = _channels_for_prefixes(midi, {"piano_lh"})
    ensemble_channels = _channels_for_prefixes(midi, {"violin", "trumpet"})
    percussion_channels = _channels_for_prefixes(midi, {"percussion"})

    if not piano_rh_channels or not piano_lh_channels or not ensemble_channels:
        raise ValueError("Could not detect both piano and melodic ensemble MIDI channels␣
↪for layered rendering.")

    with tempfile.TemporaryDirectory() as temp_dir:
        piano_rh_midi = Path(temp_dir) / "ensemble-piano-rh.midi"
        piano_lh_midi = Path(temp_dir) / "ensemble-piano-lh.midi"
        ensemble_midi = Path(temp_dir) / "ensemble-other.midi"
        percussion_midi = Path(temp_dir) / "ensemble-percussion.midi"
        piano_rh_wav = Path(temp_dir) / "ensemble-piano-rh.wav"
        piano_lh_wav = Path(temp_dir) / "ensemble-piano-lh.wav"
        ensemble_wav = Path(temp_dir) / "ensemble-other.wav"
        percussion_wav = Path(temp_dir) / "ensemble-percussion.wav"
```

(continues on next page)

(continued from previous page)

```
        _write_filtered_midi(
            midi,
            piano_rh_channels,
            piano_rh_midi,
            channel_map={channel: 0 for channel in piano_rh_channels},
            force_program=0,
        )
        _write_filtered_midi(
            midi,
            piano_lh_channels,
            piano_lh_midi,
            channel_map={channel: 0 for channel in piano_lh_channels},
            force_program=0,
        )
        _write_filtered_midi(midi, ensemble_channels, ensemble_midi)
        if percussion_channels:
            _write_filtered_midi(midi, percussion_channels, percussion_midi)

        render_wav(str(piano_rh_midi), str(piano_rh_wav), piano_soundfont_path, sample_
↪rate)
        render_wav(str(piano_lh_midi), str(piano_lh_wav), piano_soundfont_path, sample_
↪rate)
        render_wav(str(ensemble_midi), str(ensemble_wav), ensemble_soundfont_path,␣
↪sample_rate)
        layers = [str(piano_lh_wav), str(piano_rh_wav), str(ensemble_wav)]
        weights = [1.8, 1.0, 0.9]
        if percussion_channels:
            render_clap_wav(str(percussion_midi), str(percussion_wav), sample_
↪rate=sample_rate)
            layers.append(str(percussion_wav))
            weights.append(0.8)
        mix_wavs(layers, wav_path, weights=weights)
```

The second excerpt shows how the movement MIDI files are ordered and concatenated into the full listening file:

Listing 8: Ordering the movement renders and concatenating them into the full WAV.

```
def render_full_wav(
    *,
    output_dir: str,
    stem: str,
    piano_soundfont_path: str,
    ensemble_soundfont_path: str,
    sample_rate: int,
) -> None:
    midi_paths = _movement_midi_paths(output_dir, stem)
    if not midi_paths:
        raise FileNotFoundError("No MIDI files found to render.")

    movement_wavs: list[str] = []
```

(continues on next page)

(continued from previous page)

```
    for index, midi_path in enumerate(midi_paths, start=1):
        movement_wav = os.path.join(output_dir, f"{stem}-mvt{index}.wav")
        render_layered_wav_for_midi(
            midi_path=midi_path,
            wav_path=movement_wav,
            piano_soundfont_path=piano_soundfont_path,
            ensemble_soundfont_path=ensemble_soundfont_path,
            sample_rate=sample_rate,
        )
        movement_wavs.append(movement_wav)

    with tempfile.TemporaryDirectory() as temp_dir:
        concat_list = os.path.join(temp_dir, "concat.txt")
        Path(concat_list).write_text(
            "".join(f"file '{Path(path).resolve()}'\n" for path in movement_wavs),
            encoding="utf-8",
        )
        output_wav = os.path.join(output_dir, f"{stem}.wav")
        cmd = [
            "ffmpeg",
            "-y",
            "-f",
            "concat",
            "-safe",
            "0",
            "-i",
            concat_list,
            "-c",
            "copy",
            output_wav,
        ]
        print(f"Running: {' '.join(cmd)}")
        result = subprocess.run(cmd, capture_output=True, text=True)
        if result.returncode != 0:
            print(result.stderr, file=sys.stderr)
            sys.exit(result.returncode)
        if result.stderr:
            print(result.stderr, end="")
        print(f"Wrote {output_wav}")
```

This layered render path exists for musical reasons as much as technical ones. It keeps the piano drone audible, gives the treble material its own timbral space, and allows the release process to expose both the full piece and the individual movement WAVs. The build and release workflow therefore records not only the score but also the listening model used during composition.

12.8 What This Case Study Shows

The ensemble package shows a practical way to move from analysis-derived material into a more independent composition. The source recording still plays an important role. The curated regions remain central as well. But the final result is not governed only by the source audio. It is governed by a phrase system, a movement plan, and a set of environmental layers designed to make the piece performable.

This hybrid approach is one way to move from analysis-derived material toward a more independent composition. It

suggests how an analysis-driven motive can remain visible while the surrounding musical world becomes more flexible, more modular, and more explicitly composed.

CHAPTER

THIRTEEN

CONFIGURATION AND PARAMETERIZATION

Configuration is central to the quartet work. The repository uses TOML because it is readable, easy to version, and structured enough to separate different kinds of choices. In the quartet packages, configuration controls three main areas. The first is output and metadata. The second is rendering. The third is musical generation.

The output section controls naming and filename construction. The render section controls SoundFonts and sample rate. The generation section controls measure count, note and rest durations, leap bounds, density, tempo, and seed. The materials section defines the pitch-class pool. The parts section defines instrumentation, ranges, MIDI channel assignments, clefs, and roles.

This design is important for two reasons. First, it makes generated scores reproducible. Second, it turns the quartet packages into research interfaces rather than fixed scripts. The config is the place where an experiment can be defined, rerun, and compared.

No. 2 extends this model rather than replacing it. The second quartet adds hand-specific piano parameters, separate piano occupancy, and more detailed chord controls. This is one of the best examples in the project of how technical design and musical design work together.

The first quartet config is compact enough to show the overall model:

Listing 1: Core configuration for Quartet No. 1.

```toml
title = "Algo Rhythms Quartet No. 1"
composer = "George K. Thiruvathukal"

[output]
basename = "algo-rhythms-quartet-no-1"
label = ""
include_measures = true
include_tempo = true
include_seed = true
include_timestamp = true
timestamp_format = "%Y-%m-%d@%H-%M"

[render]
# Dual-soundfont render setup:
#   piano   -> ~/.soundfonts/SalamanderGrandPiano-V3+20200602.sf2
#   strings -> ~/.soundfonts/AegeanSymphonicOrchestra.sf2
piano_soundfont = "~/.soundfonts/SalamanderGrandPiano-V3+20200602.sf2"
strings_soundfont = "~/.soundfonts/AegeanSymphonicOrchestra.sf2"
sample_rate = 44100

[generation]
measures = 32
```

(continues on next page)

(continued from previous page)

```
time_signature = [4, 4]
min_note_duration = "1/16"
max_note_duration = "1"
min_rest_duration = "1/16"
max_rest_duration = "1/4"
max_simultaneous_tones_per_quantum = 4
max_pitch_leap = 5
tempo_bpm = 50
seed = 42

[materials]
pitch_classes = [0, 1, 3, 6, 8, 10]

[[parts]]
id = "violin"
name = "Violin"
short_name = "Vln."
family = "strings"
clef = "treble"
role = "melodic"
midi_channel = 1
midi_program = 41
midi_instrument = "violin"
range = ["G3", "A7"]

[[parts]]
id = "viola"
name = "Viola"
short_name = "Vla."
family = "strings"
clef = "alto"
role = "melodic"
midi_channel = 2
midi_program = 42
midi_instrument = "viola"
range = ["C3", "E6"]

[[parts]]
id = "cello"
name = "Cello"
short_name = "Vc."
```

The second quartet config shows how the same structure can grow to support more detailed piano behavior:

Listing 2: Extended generation controls in Quartet No. 2.

```
title = "Algo Rhythms Quartet No. 2"
composer = "George K. Thiruvathukal"

[output]
basename = "algo-rhythms-quartet-no-2"
label = ""
```

(continues on next page)

(continued from previous page)

```
include_measures = true
include_tempo = true
include_seed = true
include_timestamp = true
timestamp_format = "%Y-%m-%d@%H-%M"

[render]
# Dual-soundfont render setup:
#   piano   -> ~/.soundfonts/SalamanderGrandPiano-V3+20200602.sf2
#   strings -> ~/.soundfonts/AegeanSymphonicOrchestra.sf2
piano_soundfont = "~/.soundfonts/SalamanderGrandPiano-V3+20200602.sf2"
strings_soundfont = "~/.soundfonts/AegeanSymphonicOrchestra.sf2"
sample_rate = 44100

[generation]
measures = 32
time_signature = [4, 4]
min_note_duration = "1/16"
max_note_duration = "1"
min_rest_duration = "1/16"
max_rest_duration = "1/4"
max_simultaneous_tones_per_quantum = 4
piano_max_simultaneous_events = 6
max_pitch_leap = 5
tempo_bpm = 50
seed = 42
piano_chord_probability = 0.45
piano_min_chord_tones = 2
piano_max_chord_tones = 4
piano_rh_min_chord_tones = 2
piano_rh_max_chord_tones = 3
piano_lh_min_chord_tones = 2
piano_lh_max_chord_tones = 3
piano_chord_span = 12
```

CHAPTER

FOURTEEN

BUILD, RENDER, AND RELEASE ENGINEERING

The repository treats build and release behavior as part of the system, not as an afterthought. Local builds run through `../build.sh`. That script creates or reuses a virtual environment, installs the project in editable mode, checks for required tools, and then builds each score path in turn. When audio tools are available, it also renders WAV output. This gives one repeatable command for the full local workflow.

Continuous integration mirrors the same idea. GitHub Actions builds the score packages, uploads artifacts, and creates tagged releases. The repository is therefore able to move from source code to a published release without a separate manual packaging process. That helps keep the technical report grounded because the output pipeline is not hypothetical. It is already in regular use.

The quartet render path is the most complex part of this build layer. The system downloads and caches SoundFonts when needed. It renders piano and string layers separately. It uses FluidSynth for synthesis and `ffmpeg` for the final mix. This is a concrete example of a musical requirement driving a more careful software design.

This build-first design also improves debugging. When a score changes, the same automated path can regenerate notation and audio. That shortens the loop between compositional change and technical verification.

CHAPTER

FIFTEEN

DETAILED DISCUSSION OF SCORE BEHAVIOR

This section should be treated as score-first rather than code-first. The earlier sections explain how the system is built. Here the focus should shift to what the generated or encoded scores actually do.

For `Modus Operandi`, the main points are movement shape, modal identity, and the relation between a clear melodic line and slow intervallic support. For `Jazz Rhythmic Patterns`, the main points are rhythmic identity, reuse, comparison across notated examples, and the choice to show those examples in a chart-like jazz hits notation rather than a more abstract rhythmic staff. For the quartets, the main points are density, register, pitch organization, instrumentation, and the contrast between No. 1 and No. 2.

The two quartets deserve the closest reading. No. 1 should be discussed as the first stable proof-of-concept score. No. 2 should be discussed as an evolving branch whose main musical changes are concentrated in piano behavior. This includes chord generation, spacing, occupancy, and the gradual shift away from purely single-line keyboard writing.

In the final paper, this section should rely more on score figures than on code. Small code references are still fine when they explain a specific musical behavior, but the primary evidence here should be the engraved scores and, if useful, short notated examples.

CHAPTER

SIXTEEN

LIMITATIONS

The quartet generator is useful, but it is not finished. The most obvious limitation is that material can still accumulate too easily in the upper voices. Even after changes to piano behavior, the ensemble does not yet distribute thematic attention in a fully balanced way. Violin can still attract more foreground activity than viola or cello.

Motivic transfer is also limited. The present system can generate local material, but it does not yet do enough to move short ideas between instruments in a clear chamber-music way. This weakens the sense of dialogue between parts.

Harmonic planning is another limitation. The current quartet code makes many local decisions well enough to produce playable material, but it does not yet plan longer spans of harmonic motion in a strong way. In No. 2, left-hand chord spacing is more deliberate than before, but the system is still not building a full harmonic strategy over large stretches of the piece.

Finally, the repository still contains a useful but incomplete placeholder in the `algorithmic` package. That is not a flaw in itself, but it does show that the broader planned system is still under construction.

CHAPTER

SEVENTEEN

FUTURE WORK

The next clear step is stronger motivic transfer across instruments. The quartet work would benefit from short ideas that can move from one part to another in a controlled way. This would make the ensemble behave less like several related generators and more like chamber music.

Section-level control is also needed. At present, most decisions are local. Future versions should define larger sections with different density, register, and activity profiles. That would improve formal shape and help the music move through clearer phases.

Harmonic planning should become more explicit, especially for piano writing. No. 2 has already started to separate hand behavior and chord spacing, but the next stage is to connect those local sonorities into larger harmonic motion. Broader instrumentation and more systematic post-tonal methods are also natural next steps for the repository as a whole.

CHAPTER

EIGHTEEN

CONCLUSION

This repository shows that programmatic composition can be treated as both an artistic workflow and a technical system. The project already supports a complete path from source code to engraved score, MIDI, WAV, and published release output. It also shows that finished work and exploratory work can live in the same codebase if the architecture is clear enough.

The main value of the repository is not only that it contains several scores. It is that those scores are connected to explicit generation logic, rendering choices, and automated build paths. That makes the project useful for composition, documentation, and further research.

The quartets are the best example of this combined role. No. 1 preserves a stable proof of concept. No. 2 provides a branch for active musical and technical change. Together they show how a composition system can grow without losing its earlier results.

CHAPTER

NINETEEN

APPENDIX A: REPOSITORY MAP

The repository is organized around a small number of score packages, configuration files, and build helpers.

```
compositions-abjad/
├── configs/
│   ├── algorithmic-piano-quartet-no1.toml
│   └── algorithmic-piano-quartet-no2.toml
├── docs/
├── src/
│   ├── modus_operandi_abjad/
│   ├── jazz_rhythm/
│   ├── algorithmic_piano_quartet_no1/
│   ├── algorithmic_piano_quartet_no2/
│   ├── bird_im_migration/
│   ├── bird_im_migration_ensemble/
│   └── algorithmic/
├── build.sh
├── midi2wav.sh
├── pyproject.toml
└── .github/workflows/build.yml
```

The top level is intentionally small. Score packages live in `src`. Quartet configuration lives in `configs`. Build and release behavior lives at the top level and in the GitHub workflow file. The `docs` directory now holds the technical report source.

CHAPTER

TWENTY

APPENDIX B: SELECTED CONFIGURATIONS

This appendix will collect shorter, annotated excerpts from the quartet configuration files. The goal is to keep the main body of the paper readable while still showing the actual parameter structure used by the generators.

The first quartet config should be used to show the baseline model. The second quartet config should be used to show how that baseline can be extended without changing the overall shape of the configuration file.

CHAPTER

TWENTYONE

APPENDIX C: SELECTED CODE LISTINGS

This appendix is reserved for code listings that are too long for the main body. The main paper should use short excerpts when a small example is enough. Longer listings belong here, where they can support reproducibility without breaking the flow of the argument.

One useful example is the second quartet chord builder, which contains much of the recent experimental work on left-hand and right-hand spacing:

Listing 1: Full chord-construction function for Quartet No. 2.

```
def _build_piano_chord(
    seed_pitch: int,
    low: int,
    high: int,
    pitch_classes: tuple[int, ...],
    minimum_tones: int,
    maximum_tones: int,
    max_span: int,
    minimum_total_span: int,
    preferred_steps: tuple[int, ...],
    minimum_separation: int,
    rng: random.Random,
) -> tuple[int, ...]:
    chord_low = max(low, seed_pitch - max_span)
    chord_high = min(high, seed_pitch + max_span)
    candidates = [
        pitch
        for pitch in range(chord_low, chord_high + 1)
        if pitch % 12 in pitch_classes and pitch != seed_pitch
    ]
    if not candidates:
        return (seed_pitch,)

    def interval_quality(pitch: int) -> tuple[int, int]:
        interval = abs(pitch - seed_pitch)
        nearest_preferred = min(abs(interval - step) for step in preferred_steps)
        return (nearest_preferred, interval)

    candidates.sort(key=lambda pitch: (*interval_quality(pitch), pitch))
    candidate_pool = [seed_pitch, *candidates[: min(len(candidates), 12)]]
    available_size = min(maximum_tones, len(candidate_pool))
    if available_size <= 1:
```

(continues on next page)

(continued from previous page)

```
        return (seed_pitch,)

    target_size = rng.randint(minimum_tones, available_size)
    valid_chords: list[tuple[tuple[int, ...], tuple[int, int, int]]] = []

    for combo in combinations(candidate_pool[1:], target_size - 1):
        chord = tuple(sorted((seed_pitch, *combo)))
        if any((upper - lower) < minimum_separation for lower, upper in zip(chord,␣
↪chord[1:])):
            continue
        total_span = chord[-1] - chord[0]
        if total_span > max_span:
            continue
        span_penalty = max(0, minimum_total_span - total_span)
        adjacent_penalty = sum(
            min(abs((upper - lower) - step) for step in preferred_steps)
            for lower, upper in zip(chord, chord[1:])
        )
        seed_penalty = sum(
            min(abs(abs(pitch - seed_pitch) - step) for step in preferred_steps)
            for pitch in chord
            if pitch != seed_pitch
        )
        valid_chords.append((chord, (span_penalty, adjacent_penalty + seed_penalty, -
↪total_span)))

    if valid_chords:
        valid_chords.sort(key=lambda item: item[1])
        best_score = valid_chords[0][1]
        top_chords = [chord for chord, score in valid_chords if score == best_score][:4]
        return rng.choice(top_chords)

    return (seed_pitch,)
```

Another useful example is the first quartet score assembly layer:

Listing 2: Final score assembly for Quartet No. 1.

```
    if not piece.generation_note_lines:
        return None

    lines = " ".join(
        rf'\line {{ "{_quote_markup_text(line)}" }}'
        for line in piece.generation_note_lines
    )
    return abjad.Markup(rf'\markup \column {{ {lines} }}')


def build_lilypond_file(piece: Piece) -> abjad.LilyPondFile:
    voice_lookup = {voice.staff_id: voice for voice in piece.voices}

    violin_staff = _voice_to_staff(voice_lookup["violin"], piece)
```

(continues on next page)

(continued from previous page)

```
    viola_staff = _voice_to_staff(voice_lookup["viola"], piece)
    cello_staff = _voice_to_staff(voice_lookup["cello"], piece)
    piano_rh_staff = _voice_to_staff(voice_lookup["piano_rh"], piece)
    piano_lh_staff = _voice_to_staff(voice_lookup["piano_lh"], piece)

    piano_staff = abjad.StaffGroup(
        [piano_rh_staff, piano_lh_staff],
        lilypond_type="PianoStaff",
        name="PianoStaff",
    )
    strings_group = abjad.StaffGroup(
        [violin_staff, viola_staff, cello_staff],
        lilypond_type="StaffGroup",
        name="Strings",
    )

    score = abjad.Score([strings_group, piano_staff], name="Score")
    _apply_dynamics(
        violin_staff=violin_staff,
        viola_staff=viola_staff,
        cello_staff=cello_staff,
        piano_rh_staff=piano_rh_staff,
        piano_lh_staff=piano_lh_staff,
        piece=piece,
    )
    _apply_ending(score, piece)

    instrument_names = [
        (violin_staff, "Violin", "Vln."),
        (viola_staff, "Viola", "Vla."),
        (cello_staff, "Cello", "Vc."),
        (piano_staff, "Piano", "Pno."),
    ]
    for component, full_name, short_name in instrument_names:
        abjad.setting(component).instrumentName = rf'\markup "{full_name}"'
        abjad.setting(component).shortInstrumentName = rf'\markup "{short_name}"'

    tempo = abjad.MetronomeMark(
        reference_duration=abjad.Duration(1, 4),
        units_per_minute=piece.tempo_bpm,
    )
    abjad.attach(tempo, abjad.select.leaf(violin_staff, 0))

    header_block = abjad.Block("header")
    header_block.items.append(rf'title = "{piece.title}"')
    header_block.items.append(rf'composer = "{piece.composer}"')
    header_block.items.append(r"tagline = ##f")

    layout_block = abjad.Block("layout")
    layout_block.items.append(r"indent = 2.0\cm")
```