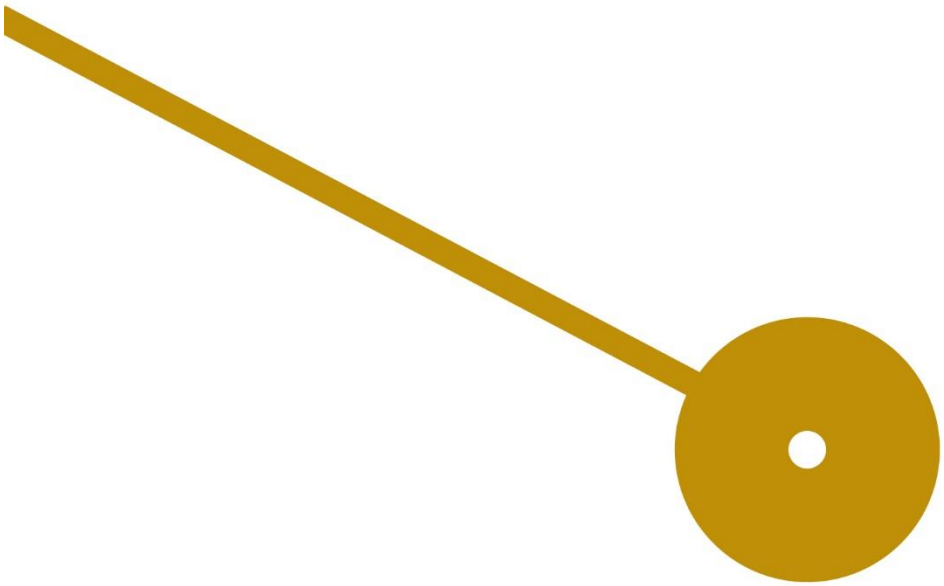
ESCOLA
SUPERIOR
DE MÚSICA
E ARTES
DO ESPETÁCULO
POLITÉCNICO
DO PORTO

MESTRADO
COMPOSIÇÃO E TEORIA MUSICAL

Interacção humano-máquina no processo composicional

Nuno André Pinheiro Trocado da Costa

10/**2017**



ESMAE
ESCOLA SUPERIOR DE MÚSICA E ARTES DO ESPETÁCULO
POLITÉCNICO DO PORTO

P.PORTO

M MESTRADO
COMPOSIÇÃO E TEORIA MUSICAL

Interacção humano-máquina no processo composicional

Nuno André Pinheiro Trocado da Costa

Dissertação apresentada à Escola Superior de Música e Artes do Espectáculo como requisito parcial para obtenção do grau de Mestre em Composição e Teoria Musical

Professor Orientador
Prof. Doutor Carlos Azevedo

Professor Co-orientador
Prof. Doutor Dimitris Andrikopoulos

10/**2017**

Agradecimentos

Aos Profs. Carlos Azevedo, Dimitris Andrikopoulos e Eugénio Amorim, respectivamente orientador, co-orientador e director do curso de mestrado, pelo apoio, ensinamentos e inspiração.

Ao Coreto Porta-Jazz, ao Sérgio Tavares e à Elisabete Magalhães pela disponibilidade.

À minha fantástica família.

Resumo

Com a problemática da interacção humano-máquina no processo composicional como pano de fundo, a presente dissertação prossegue três vectores simultâneos: (a) um *portfolio* de alguma da música que escrevi nos últimos dois anos, na qual fiz uso de (b) um conjunto de ferramentas de composição algorítmica assistida por computador que desenvolvi sincronicamente no mesmo período de tempo, por sua vez fruto de (c) uma reflexão sobre os conceitos subjacentes, aqui apresentada monograficamente. De entre as ferramentas desenvolvidas destaca-se uma biblioteca em Common Lisp integrando funções relativas a operações combinatórias simples com vastas colecções de alturas e à exploração de propriedades geométricas de ciclos rítmicos e tonais, bem como um programa de manipulação de áudio com cadeias de Markov. A música escrita que integra este trabalho assume-se simultaneamente como ponto de partida e de chegada do percurso de investigação.

Palavras-chave: composição algorítmica assistida por computador, common lisp, openmusic, pwgl, excentricidade e uniformidade rítmica e tonal, cadeias de Markov

Abstract

With the issues of human-machine interaction in the compositional process as a background, the present dissertation follows three simultaneous vectors: (a) a portfolio of some of the music that I wrote in the past two years, in which I applied (b) a set of computer-assisted algorithmic composition tools that I developed synchronously in that period of time, following (c) a reflection on the underlying concepts, presented here monographically. A library in Common Lisp stands out among the tools, including functions to perform simple combinatorics with large pitch collections and to the exploration of geometric properties observed in rhythmic and tonal cycles, as well as a program to manipulate audio using Markov chains. The written music—which is an integral part of this work—is simultaneously the point of departure and the point of arrival for the research path that I followed.

Keywords: computer-assisted algorithmic composition, common lisp, openmusic, pwgl, rhythmic oddity, evenness, markov chains

Índice

Lista de Figuras

— 1 —

Introdução e Motivação

Tenho cumprido grande parte da minha actividade musical próximo do universo a que, passe as insuficiências dos rótulos, podemos chamar Jazz. É um universo que problematiza em primeira linha a relação entre escrita e improvisação. A música (bem) improvisada possui uma energia muito particular. Brilha nela a cada passo e a cada momento a centelha da imaginação dos músicos, e coloca no plano artístico uma forma de colaboração sem paralelo no contexto das relações inter-subjectivas. Porém, também é certo que há todo um tipo de construções sonoras absolutamente impossíveis de desenvolver *ex tempore*, por necessitarem da ponderação, da conjugação, da organização sistemática, etc., que só se alcançam por via da (pré-)composição. Neste domínio, a rigorosa disciplina da formalização musical com os seus puros e maravilhosos algoritmos encontra-se nos antípodas da espontaneidade instintiva e caótica da improvisação. Por isso, pretendendo maximizar tanto o que de Apolo como o que de Dionísio há na música, interessei-me facilmente pelo campo da composição algorítmica.

A presente dissertação prossegue três vectores simultâneos: (a) um *portfolio* de alguma da música que escrevi nos últimos dois anos, na qual fiz uso de (b) um conjunto de ferramentas de composição algorítmica assistida por computador que desenvolvi sincronicamente no mesmo período de tempo, por sua vez fruto de (c) uma reflexão sobre os conceitos subjacentes, aqui apresentada monograficamente.

Dada a relativa brevidade exigida para este texto, não será possível analisar todos os aspectos composicionais presentes nas obras, nem todas as ferramentas de composição algorítmica desenvolvidas. Faremos uma selecção, ilustrando apenas algumas das técnicas utilizadas com exemplos retirados das peças e remissões para trechos onde essa utilização seja mais clara.

É de salientar que as obras que integram este estudo funcionam como *prius* metodológico para a investigação, e simultaneamente como resultado dela.

O CD que acompanha esta tese contém ainda o código e os *patches* mencionados no texto, bem como os materiais relativos à secção 3.4.

— 2 —

Composição Algorítmica Assistida por Computador (CAAC)—Um Breve Olhar

2.1 Terminologia

Numa época em que a informática inunda todos os aspectos da nossa existência, falar de composição assistida por computador[1] é uma verdadeira tautologia. O computador *assiste* o compositor na notação e preparação de partituras, na síntese e processamento de som, na gravação e edição, na audição de trechos sequenciados, no acesso a publicações teóricas ou a obras editadas, na utilização de instrumentos virtuais, na análise de partituras ou fonogramas, na comunicação com outros compositores ou com outros profissionais, na organização e planeamento do trabalho... até o mais simples metrónomo electrónico pode ter nas suas entranhas um microcontrolador.

Uma designação alternativa para aquilo de que pretendemos tratar é *composição algorítmica*. Porém, também essa designação é problemática.

Intuitivamente, associamos a palavra algoritmo a qualquer coisa que envolve uma sequência ordenada de passos, a realização de cálculos ou outras operações, a solução de um problema. O dicionário define-a assim:

> 1. (aritmética, obsoleto) sistema de numeração decimal assimilado dos árabes 2. (matemática) sequência finita de regras, raciocínios ou operações que, aplicada a um número finito de dados, permite solucionar classes semelhantes de problemas (p. ex.: para a extracção de uma raiz cúbica) 2.1. processo de cálculo; encadeamento das acções necessárias ao cumprimento de uma tarefa; processo efectivo, que produz uma solução para um problema num número finito de etapas *(o algoritmo que permite obter o seno de x com uma certa precisão)* 3. (por extensão) [...] mecanismo que utiliza representações análogas para resolver problemas ou atingir um fim, noutros campos do raciocínio e da lógica *(pode-se considerar a gramática como um a. na construção das frases)* 4. (informática) conjunto de regras e pro-

[1] *Computer-assisted composition* na designação em língua inglesa; por vezes também surge na literatura como *computer-aided composition*.

> cedimentos lógicos perfeitamente definidos que levam à solução de um problema num número finito de etapas (Houaiss, 2001).

Isto relativamente à utilização quotidiana da palavra. A definição formal de algoritmo, por seu turno, é um assunto que continua a ocupar seriamente os investigadores (Vardi, 2012). Dois dos problemas conceptuais são: (a) a questão da identidade—quando é que dois ou mais programas que calculam uma função o fazem através do mesmo algoritmo ou, ao invés, através de algoritmos diferentes; e, relacionado com isto, (b) qual é a relação básica entre um algoritmo e as suas implementações (Moschovakis, 2001, pp. 17-18). Hill (2016), analisando a questão tanto do ponto de vista da informática como da filosofia, propõe a seguinte definição: "um *algoritmo* é uma estrutura de controle finita, abstracta, efectiva e composta, dada imperativamente, cumprindo uma função determinada através de disposições determinadas"[2] (p. 24). Afirma ainda que não se subsumem à noção de algoritmo coisas como: receitas de cozinha (envolvem juízos de adequação e assunções não expressas—uma série de instruções que pode ser seguida *melhor* ou *pior* não é um algoritmo), o jogo das "Damas" (é interactivo e conceptualmente incerto), o "Jogo da Vida" de Conway (não possui uma cláusula de terminação), a definição recursiva de factorial (não é imperativa; um algoritmo tem de *fazer*, e não de *ser*).

A utilização de algoritmos na composição é intemporal e absolutamente prévia à existência de computadores. Não sendo aqui oportuno desenvolver uma história dos algoritmos na música—Nierhaus (2009, pp. 7-66) integra-a no contexto da evolução da filosofia, da matemática e da tecnologia, *vd.* também, entre outros, Simoni (2003)—podem-se citar exemplos que vão desde o séc. XI e do sistema inventado por Guido d'Arezzo para a geração automática de melodias a partir de um texto (Palisca, 1980), até aos jogos musicais com dados (*Musikalische Würfelspiele*), que circulavam na Europa do séc. XVIII (Hedges, 1978), pelo menos um dos quais pode atribuir-se com segurança a Mozart (Noguchi, 1990), e mais tarde às operações associadas ao serialismo, à música estocástica (Xenakis, 1971), etc.

Certo é que durante séculos se mobilizaram processos algorítmicos sem recurso a computadores, tecnologia que é relativamente recente. Em teoria, muitos algoritmos utilizados hoje na composição poderiam ser seguidos "à mão", muito embora tal constitua uma tarefa infinitamente morosa e fastidiosa, e por isso impraticável, a partir de um certo nível de complexidade. Outros algoritmos mais simples são de facto seguidos quotidianamente sem intervenção da informática. Pense-se, por exemplo, na transposição de um motivo para outro grau da escala, na retrogradação, na aumentação rítmica, no cumprimento de regras estritas para a condução das vozes, etc. Por este motivo, como se pretende reduzir o âmbito do termo aos processos que exigem a utilização de computadores, em razão da respectiva complexidade, ou ao menos em que essa utilização seria claramente vantajosa, a designação "composição algorítmica" tem como óbice apresentar-se como demasiado ampla.

[2] An *algorithm* is a finite, abstract, effective, compound control structure, imperatively given, accomplishing a give purpose under given provisions. [tradução minha, como o são, neste texto, todas as ulteriores]

Ariza (2005, pp. 4-5), recolhendo os variados termos com que os autores foram baptizando projectos em áreas ao menos parcialmente coincidentes, acaba por introduzir um conceito híbrido: CAAC, composição algorítmica ajudada pelo computador (*computer-aided algorithmic composition*). Concede assim a impossibilidade de uma escolha binária—a aglutinação do "algoritmo" com o "computador" obvia à ambiguidade de cada um dos conceitos isolados.

Porém, os computadores são de tal forma ubíquos no mundo actual que, paradoxalmente, se tornam invisíveis. É previsível que, num futuro próximo, deixará de se falar de actividades humanas "assistidas por computador", porque o computador se vai encontrar, de algum modo, omnipresente em praticamente todas essas actividades. Entretanto, a Wikipedia ainda mantém uma entrada para *Computer-aided*, com ligações para os mais diversos campos, desde a engenharia (*computer-aided manufacturing*), à pedagogia (*computer-assisted instruction*), à medicina (*computer-assisted sperm analysis*), etc. ("Computer aided", s.d.). Seja como for, não há grande inconveniente em desde já prescindirmos da referência ao computador, e falarmos tão só em "composição algorítmica", subentendendo-se que esta, naquilo que é particularmente relevante, dificilmente se fará hoje com recurso a aparos e papiros. Contudo, a referência ao computador justifica-se quando se pretende realçar especialmente esse facto—mais dos que os algoritmos, aludir à específica tecnologia utilizada na sua implementação. É, de certa forma, o caso do presente trabalho, pelo que vamos utilizar a expressão que dá título a este capítulo, bem como a sigla "CAAC".

Por vezes também se fala de *composição automática* ou *autónoma*. Esses termos, no entanto, são melhor deixados para os casos em que a máquina é programada para gerar todos ou praticamente todos os elementos de uma obra completa ou de um fragmento, com mínima intervenção do compositor. A diferença, porém, não é só de grau de intervenção. Casos há em que, após experimentação, os algoritmos acabam por ser a tal ponto especializados, e os respectivos parâmetros invariavelmente determinados, que ao computador só resta encontrar e oferecer, já sem intervenção humana, o resultado único que é a obra completa, assim descrita num estado de formalização integral. Na composição automática, pelo contrário, existe uma margem de "livre arbítrio" artificial. O computador é programado para responder a *inputs* que podem ter magnitudes variáveis, ou existe uma certa dose de aleatoriedade, ou se recorre extensivamente a algoritmos de tomada de decisão computacional, tudo decorrendo com um aparente corte do cordão umbilical entre o programador e a máquina, que assim parece produzir a música *sponte sua*. Como notam Anders e Miranda (2009, p. 134), mesmo nestes casos, uma atitude comum consiste em seleccionar manualmente, de entre vários resultados previamente obtidos, aquele ao qual se atribui maior qualidade musical (*cherry-picking*). Ainda segundo os mesmos autores, a composição automática é especialmente interessante para aplicações em que os compositores não podem intervir em simultâneo com a geração da música. Dão exemplos relativos à música interactiva e à música para vídeo-jogos. Para além destes exemplos, é de mencionar ainda o amplo domínio dos sistemas de improvisação automática.

2.2 Papel dos Algoritmos no Processo Composicional

Os algoritmos e o computador situam-se, à partida, no âmbito do processo composicional ou pré-composicional (para quem admite esta última como categoria autónoma). Não integram a obra-resultado, que em si não se confunde com os caminhos da criação musical que a ela conduziram, com a pessoa do autor e com o concurso das suas inteligência e intuição, com os rascunhos descartados e as edições necessárias, com o contexto social e o *Zeitgeist*, e bem assim com as específicas ferramentas utilizadas pelo compositor para gerar o material musical que constitui a composição. A obra pode não ser perfeitamente opaca relativamente a estes factores, revelando-os ao analista de forma mais ou menos expressa, mas em si não se identifica com os meios utilizados na sua feitura.

Isto não significa que não haja aspectos estéticos na ideação de um algoritmo. Quando, aliás, são a poética e a estética do código que se assumem como expressão central da subjectividade artística do criador–programador, estamos perante uma forma de arte conceptual que se pode designar como *software art*, definida como arte em que o material é código de instruções formal e/ou que remete para conceitos culturais relacionados com o *software* (Cramer, 2002).

Por outro lado, a unidade e consistência formais são geralmente apreciadas por teóricos e compositores. O que porém é mais uma preocupação atinente ao *critério* estético de valoração, não propriamente ao *objecto* da obra musical.

Por outro lado ainda, o compositor pode limitar-se à mera explicitação ou definição de um algoritmo. É o que faz J. S. Bach com vários cânones da *Oferenda Musical*, em que o algoritmo canónico (retrógrado, inversão, aumentação, em espelho, *per tonos*, etc.) não se encontra realizado na partitura—a realização é deixada como um *puzzle* para o intérprete. A notação musical pode incluir vários outros tipos de indicações deixadas pelo compositor, muitas vezes por uma simples questão de brevidade, que se não são rigorosamente algoritmos possuem análoga natureza, por envolverem sequências ordenadas de passos, cumprindo uma função determinada.

Na pós-modernidade a prática musical encontra-se de tal forma atomizada que é impossível de descrever e caracterizar em termos gerais. Quaisquer considerações hão-de fazer-se, tendencialmente, no âmbito de um radical individualismo. A pergunta "Qual o papel dos algoritmos na música?" deve ser reformulada para "Qual o papel dos algoritmos na música de Fulano, ou na de Sicrano?", ou mesmo "naquela peça, ou naquele aspecto concreto?". Vimos já que, por vezes, a composição algorítmica é levada ao ponto de o algoritmo descrever integralmente a obra musical. Noutros casos porém, funcionam apenas como catalisadores da criatividade do compositor, que depois prossegue alheado deles. Outro aspecto em que a prática pode ser muito diversa é o seguinte. Face à insatisfação com o resultado produzido automaticamente, há duas atitudes ideais: ou se procede descomprometidamente às correcções necessárias para alinhar o produto mal-nascido com a sensibilidade do artista, ou se vê esse acto como sacrílego, considerando então que a única coisa decente de se fazer é reconfigurar o algoritmo para que no futuro origine melhores resultados. No primeiro caso exerce-se a prerrogativa máxima do compositor: editar.

No segundo priveligia-se a pureza formal do algoritmo, plasmando essa pureza na obra. Outro aspecto ainda: não será errado dizer-se que os algoritmos assumem relevância suplementar na música electrónica, mas o que dizer da música interactiva, de instalações, das artes ditas "digitais", e dos vários modos de expressão humana que hodiernamente dilatam e colocam em crise as noções de arte, de obra, de música…Como identificar abstractamente a função dos processos algorítmicos nesses contextos?

Só mesmo uma observação casuística é idónea a caracterizar a importância e posicionamento de concretos algoritmos no processo de criação da obra. Para além disto, a convocação de meios computacionais assume-se como camaleónica, há uma grande flexibilidade que possibilita os mais diversos modos de diálogo relacional com o compositor, e os sistemas adaptam-se às preferências deste (Brown, 2000).

Na prática correspondente à escrita algorítmica das obras em causa nesta dissertação, identifico as seguintes ideias-chave:

- A prática é *exploratória*. Mais do que alcançar determinado resultado antevisto ou desejado, o objectivo é perscrutar um caminho teórico, sondar da sua valia para a criação musical, investigar de que forma pode ser executado, transfigurado, combinado, a que outras ideias conduz, e como enriquece a imaginação que se desenvolve no muito próprio mundo dos sons. O material musical produzido algoritmicamente pode influenciar a escrita sem sequer transitar directamente para a partitura.

- A prática é *experimental*. Desenvolve-se num ciclo em que a ideia de um algoritmo conduz à sua implementação em *software*, cuja operação é posta à prova, avaliando-se os resultados segundo critérios estéticos e práticos, o que tem como consequência a reformulação do algoritmo—e regressando assim ao início do ciclo. Este é um ciclo ideal, já que na realidade as coisas decorrem de forma bem mais caótica.

- A prática é *exógena*. O algoritmo é executado por uma máquina, segundo os princípios da CAAC, se bem que o processo é interactivo, decorrendo num diálogo muito próximo entre o ser humano e o computador.

- O algoritmo encontra-se tendencialmente no *exórdio* do processo composicional. É suposto ocorrer uma posterior combinação com a intuição e imaginação humanas.

No que ficou agora dito, decerto que não nos afastamos muito de outras perspectivas convergentes colhidas da literatura. Só para fornecer alguns exemplos, veja-se, *v.g.*, Laske (1981):

> Podemos ver a interacção compositor–programa ao longo de uma trajectória conduzindo desde um controle puramente manual para um controle exercido por algum algoritmo composicional (máquina de composição). A zona de maior interesse para a teoria da composição é a zona do meio da trajectória, pois permite uma aproximação mais flexível. Os poderes da intuição e da computação podem ser combinados.

> Quanto mais cedo no processo composicional os algoritmos são usados, maior é o ónus que recai sobre os compositores como intérpretes dos resultados da computação. Claramente, isto não altera o facto de que cada compositor é em última análise responsável pela forma final da composição.[3] (p. 54)

E de acordo com Anders e Miranda (2009):

> Quando compondo com o auxílio do computador, os compositores aplicam ou até desenvolvem certos meios técnicos, mas estes meios técnicos não são o verdadeiro propósito; o principal resultado do seu trabalho é a criação artística. Os compositores decidem quais as partes ou aspectos da música a gerar pelo computador, e quais as partes compostas manualmente.[4] (p. 134)

Afirma ainda Gato (2016):

> Obter materiais musicais completos e definitivos—que seriam simplesmente copiados para a partitura, dispensando alterações suplementares ou a adição de outros materiais—foi raramente o objectivo …. No centro da composição assistida por computador (CAC) está o equilíbrio entre métodos manuais e automatizados e, conforme explorava cada um deles, aprendi a seleccionar os aspectos que julguei artisticamente mais úteis.[5] (p. 45)

Sandred (2017) relaciona a utilização do computador com a gestão de estruturas numa composição, em particular na invenção de sistemas para a criação de estruturas que sejam musicalmente úteis. Refere ainda como a utilização do computador no processo composicional altera o *workflow* em comparação com outros métodos:

> O computador pode oferecer um *feedback* instantâneo relativamente à forma como uma estrutura encaixa com um conceito musical, tanto pela produção de uma partitura como fornecendo *feedback* auditivo. Assim, o compositor pode, cedo no processo composicional, adaptar as estruturas para melhor corresponderem às suas

[3] We may view composer-program interaction along a trajectory leading from purely manual control to control exercised by some compositional algorithm (composing machine). The zone of greatest interest for composition theory is the middle zone of the trajectory, since it allows for a great flexibility of approach. The powers of intuition and machine computation may be combined. The earlier in the composition process algorithms are used, the greater is the burden on composers as interpreters of the computed results. This does not, of course, alter the fact that each composer is ultimately responsible for the composition's final form.

[4] When doing computer-aided composition, composers apply or even develop certain technical means, but these technical means are not the actual purpose; the main result of their work is the artistic output. Composers decide which compositional parts or aspects of the music the computer generates, and which parts are composed manually.

[5] To get complete and definite music materials—that would simply be copied to the score, needing no further change nor any addition of other materials—was seldom the aim …At the core of computer-assisted composition (CAC) is the balance between manual and automated methods and, as I explored each of them, I learned the aspects I found artistically most useful.

> intenções. O método torna-se interactivo entre o compositor e o computador. Enquanto um computador por ser usado para gerir conceitos já estabelecidos na estruturação da música, também há conceitos novos que seriam praticamente impossíveis de alcançar sem o poder de processamento de um computador.[6] (p. 1)

2.3 O Compositor "Centauro"

No centro da CAAC está assim uma interacção entre procedimentos algorítmicos e a decisão artística humana. Nesta interacção o computador é mais do que tão-só instrumental. Jones, Brown e d'Inverno (2012) caracterizam-no como uma *meta-ferramenta*—estabelecendo a parceria ser humano–computador a partir dos vectores *feedback*, exploração, intimidade, interactividade, introspecção, tempo, autoria e valor, os sistemas algorítmicos generativos apresentam-se como uma forma de "incrementar a novidade e a produtividade, com o distinto potencial de transformar comportamentos criativos mesmo depois de a nossa interacção com um tal sistema ter terminado"[7] (p. 27).

Claramente, a utilização do computador pode colocar-se nos termos de uma espécie de *colaboração*, com reflexo nas obras assim produzidas. Desmistificando os limites das noções tradicionais de autoria e autenticidade, Durkin (2014, p. 29) distingue entre *colaboração directa*, em que há um envolvimento óbvio de um indivíduo numa obra já existente, através da edição, tradução, co-autoria (*cowriting*) ou outros tipos de interacção intencional, e *colaboração contextual*, que diz respeito ao contexto ou enquadramento, envolvendo entidades que tradicionalmente se assume serem passivas, ou mesmo removidas fisicamente do acto criativo. Nesta segunda categoria inserem-se os avanços tecnológicos, que no fio dos séculos foram moldando indelevelmente a música. Pense-se, desde logo, nos próprios instrumentos musicais, ou, mais recentemente, nas técnicas de gravação do som. Assim, na CAAC, por estarem em causa uma máquina e uma organização abstracta de informação, é manifesta uma oportunidade de colaboração contextual: com cientistas, engenheiros, etc. Mas, à parte a dificuldade de (re)ver no computador um sujeito com autonomia ética (dificuldade que me parece inultrapassável), será concebível uma espécie de colaboração directa com um sistema computacional, sobretudo no caso de algoritmos mais complexos e sofisticados?

O problema coloca-se no campo da *inteligência artificial*, ou mais especificamente no da *criatividade artificial*. Sucede que, não obstante os fantásticos avanços que aí se têm verificado, e mesmo sem cair num cepticismo extremo, saber se o computador é realmente criativo é uma pergunta à qual—actualmente e segundo os especialistas—é impossível dar uma resposta,

[6] The computer can give an instant feedback on how a structure fits with a musical concept, either by producing a music score or by giving aural feedback. The composer can in this way already early in the composition process adapt his structures to better fit his intentions.

[7] …to increase novelty and productivity, with the distinct potential to transform creative behaviours even after our interaction with such a system has ended.

porque nela estão pressupostas várias questões prévias de cariz filosófico, para as quais falta solução minimamente clara e consensual:

> Estas [questões] incluem a natureza do significado ou da intencionalidade; se uma teoria científica da psicologia, ou da consciência, é possível em princípio; e se um computador pode alguma vez ser aceite como parte da comunidade moral dos seres humanos. De facto, até podemos ignorar aqui a *criatividade*, já que muitos filósofos defendem que não é possível qualquer explicação naturalística para *qualquer* das nossas capacidades psicológicas, nem mesmo uma explicação com base na neurociência.[8] (Boden, 2009, p. 33)

Em todo o caso, e não obstante a relevância, são questões que extravasam já largamente este estudo, como também o extravasam problemas relacionados com a fenomenologia da interacção tecnológica, a modelação da criatividade e do mecanismo de tomada de decisão, ou a caracterização rigorosa dos processos cognitivos em jogo na composição musical.

Diga-se apenas que a máquina vem tornar possível novos modos de agir na composição, o que acontece com particular acuidade em técnicas que exigem cálculos rápidos e precisos, e a manipulação de quantidades muito grandes de informação.

Uma forma *naïve* mas elucidativa de ver a composição consiste, não na "invenção"' *ex nihilo* da música, mas sim na sua "descoberta", singularizando a obra de entre um conjunto de todas as possibilidades combinatórias que em abstracto representam todas as obras musicais já existentes ou existentes em potência. No conto *A Biblioteca de Babel*, de Borges (1944), uma imensa biblioteca reúne todos os livros que em 410 páginas possuem todas as ordenações possíveis das letras do alfabeto, o ponto final, a vírgula e o espaço. A grande maioria dos livros mais não contém do que sequências ilegíveis de caracteres; porém, algures entre eles, também estão todos os romances até hoje escritos, ou que pudessem ser escritos. Imagine-se agora todas as combinações possíveis de representação simbólica da música: todas as combinações possíveis de notas, de figuras rítmicas, de dinâmica, de orquestração, etc., que pudessem ser humanamente percepcionadas (*v.g.*, notas não demasiado agudas ou graves, figuras rítmicas não demasiado curtas), encadeando-se até determinada duração total. A quantidade de obras assim representáveis desafia a nossa capacidade de compreensão: aí estará uma mera repetição de cem colcheias na nota dó, bem como a Nona Sinfonia, e ainda a Nona Sinfonia mas em que apenas uma nota é diferente. O número de combinações, se astronómico, é contudo finito. Já no séc. XVIII o teórico e violinista italiano Francesco Galeazzi calculava que as oito notas da escala, distribuídas em três figuras rítmicas (por exemplo, mínimas, semínimas e colcheias), podem formar 620.448.401.733.239.439.360.000 melodias diferentes (Galeazzi, 1796/2012, p. 317). Imagine-se agora, no domínio digital, o número de representações possíveis numa duração de 5 minutos,

[8]These include the nature of meaning, or intentionality; whether a scientific theory of psychology, or consciousness, is in principle possible; and whether a computer could ever be accepted as part of the human moral community. Indeed, you can ignore creativity here, for many philosophers argue that no naturalistic explanation of any of our psychological capacities is possible, not even an explanation based in neuroscience.

pressupondo uma amostragem *standard* de 44,1 kHz e 16 bits. Aí estão todos os sons que podem existir com essa duração e forma de representação. O número de possibilidades, porém, é:

$$(2^{16})^{44100\times60\times5}.$$

Trata-se de um número com 63.722.030 dígitos!

É irrecusável qualquer ajuda de que possamos ser beneficiários, e que reduza estes números avassaladores para algo mais dimensionado à nossa modesta condição.

A composição involve sempre um acto de individualizar, entre o infindável cosmos da música hipotética, a instância que melhor cumpre um certo desígnio artístico. A isto chama Roads (2015) o *princípio da economia da selecção*, que significa "escolher uma ou algumas hipóteses óptimas ou mais salientes de entre um vasto deserto de possibilidades insignificantes"[9] (p. 14). Acrescentando ainda que "[f]azer a escolha inspirada e intuitiva de entre uma miríade de possibilidades permanece no domínio exclusivo do talento humano"[10] (p. 15). Ora, se isto é inegável, sobretudo no que diz respeito a uma escolha "inspirada" e "intuitiva", certo é que o computador pode reduzir o âmbito das possibilidades, auxiliando-nos na dura tarefa de escolher. Esta ideia servirá de mote a muitas das técnicas que veremos no capítulo seguinte.

Munidos do poder dos sistemas computacionais, tornamo-nos sobre-humanos. Aproximamo-nos da figura mitológica do centauro. Conjugamos, por um lado, a intuição, criatividade, empatia e experiência humanas, e por outro, a força bruta exibida pelo computador em memorizar, processar e calcular, celeramente, quantidades brutais de combinações numéricas.

2.4 *Do It Yourself*

É insofismável que somos moldados pelas ferramentas que utilizamos, e não tanto o inverso. A selecção das ferramentas, mais do que orientada pela mera adaptação prática às preferências e contingências do labor individual do compositor–artífice, constitui um factor determinante do próprio produto criativo. Colocada assim a questão, a invenção de novas ferramentas personalizadas, em substituição ou complemento das que se encontram universalmente disponíveis, há-de ser forçosamente equacionada, desde que se veja a originalidade artística como valiosa. Isto porque, ao utilizarem-se ferramentas originais, é natural que se alcancem soluções também originais.

Conforme foi já observado (Jones et al., 2012, p. 199), "a tendência de alguns ambientes de produção afunilarem os utilizadores para certos modos de empenho (*modes of engagement*) é frequentemente desconsiderada como uma força activa na criação musical"[11].

Abundando enormemente o *software* direccionado ao acto de fazer música, alguns sistemas são mais genéricos e abertos, aproximando-se da máquina no estado de *tabula rasa*; outros são

[9] …choosing one or a few aesthetically optimal or salient choices from a vast desert of unremarkable possibilities.

[10] Making the inspired, intuitive choice from myriad possibilities remains the exclusive domain of human talent.

[11] The tendency of certain production environments to funnel their users into certain modes of engagement is frequently overlooked as an active force within musical creation.

tão específicos que colocam a dúvida sobre qual é o contributo mais relevante para o resultado final, se o do utilizador ou se o da pré-programação.

Para além disto, é sabido que muitos compositores desenham o seu próprio sistema ou os seus próprios sistemas. Ora, a concepção e aplicação desses sistemas pode necessitar ou beneficiar de uma implementação tecnológia, que a automatize ou, em casos que exijam poder de computação, que pura e simplesmente a tornem exequível. É a tecnologia que tem de se adaptar ao universo estético e aos específicos fins musicais pretendidos pelo compositor, não o contrário. O que também pende a favor de soluções personalizadas.

A contrapartida, claro está, consiste no dispêndio de tempo e energia para desenvolver essas soluções personalizadas, que não raro exigem um extenso *know-how* técnico. Esse tempo e energia pode bem ser mais proveitosamente aplicado alhures. Por outro lado, são de considerar as vantagens da divisão social do trabalho e da especialização, que atribuem a outrem, que não o músico, a responsabilidade de desenvolver os sistemas informáticos em causa, idealmente em colaboração com aquele. Estas considerações são ainda mais prementes quando se trata de empreendimentos qualificáveis como "reinventar a roda"—criar soluções que se encontram já desenvolvidas e disponíveis.

Não obstante tudo isto, para mim tem-se revelado proveitoso o desenvolvimento de sistemas personalizados, mesmo quando se trata nitidamente de "reinventar a roda". É que, mesmo nesses casos, há desde logo uma vantagem didática na reprodução de soluções pré-existentes. Mas, mais importante, são muitas as oportunidades para a reinventar de forma ligeiramente diferente da forma universal, o que, amiúde, e ainda por cima, só se torna patente no decurso do processo.

Ocorrendo o desenvolvimento de ferramentas em sincronia com a composição musical, é natural que pequenas originalidades técnicas confiram à obra a frescura correspondente ao emprego de um vocabulário individual. Na verdade, mais uma vantagem de formalizar e automatizar processos compositivos consiste em que as ferramentas assim desenvolvidas passam a integrar o vocabulário pessoal e ficam disponíveis para utilização regular no trabalho do compositor (Andrikopoulos, 2013, p. 5).

Pessoalmente, retiro ainda uma particular satisfação da programação com fins musicais.

A criação das próprias ferramentes tem limites, que dependem da prática musical, dos interesses, da disponibilidade e da personalidade de cada um. Para dar um exemplo, não me ocorreria fabricar os meus próprios microfones, é-me perfeitamente razoável servir-me daqueles que estão disponíveis no mercado.

Pelo que foi dito, priveligio em princípio aplicações de CAAC que sejam o mais genéricas e de utilidade geral possível, o que implica recorrer a linguagens de programação não erigidas especialmente para o domínio da música.

2.5 Ambientes de Desenvolvimento de CAAC

Apresenta enorme vivacidade o campo de desenvolvimento de *software* para possibilitar e auxiliar a CAAC. Há ferramentas direccionadas para automatizar tarefas composicionais muito específicas, e outras há que são de uso mais geral, permitindo um número ilimitado de aplicações. A CAAC pode ainda desenvolver-se com o auxílio de programas que não foram pensados para a música, bem como pode desenvolver-se em *software* musical mas não vocacionado para a composição algorítmica.

Dois sistemas hoje amplamente utilizados por compositores são o OpenMusic (Assayag, Rueda, Laurson, Agon & Delerue, 1999) e o PWGL (Laurson, Kuuskankare & Norilo, 2009). Ambos utilizam[12] a linguagem de programação Common Lisp, e foram construídos com base no sistema anterior PatchWork, desenvolvido no instituto parisiense IRCAM[13]. Como tal apresentam numerosas semelhanças entre si. Infelizmente, tanto num caso como noutro, e não obstante os esforços muito louváveis dos respectivos autores, o desenvolvimento nos últimos anos não tem sido tão activo como um utilizador entusiasta desejaria…

Em todo o caso, a principal vantagem desses sistemas consiste em oferecerem uma *interface* gráfica de programação visual de alto nível, em que objectos são dispostos no ecrã pelo utilizador e conectados entre si num denominado *patch*, assim estabelecendo as respectivas relações lógicas. O nível de conhecimentos técnicos necessários e a correspondente barreira de entrada à CAAC são assim sensivelmente reduzidos. Porém, isto não se faz sem compromissos. Pessoalmente (e esta é uma matéria em que a preferência pessoal tem um peso relevante), é-me mais fácil, prático e rápido programar em texto. Por outro lado, os ambientes de desenvolvimento integrados tradicionais oferecem facilidades (por exemplo, de *debugging*, ou de análise da *performance* do código) que não encontramos nos referidos ambientes gráficos. Acresce ainda que o texto se presta muito melhor ao *version control*, e é mais fácil de formatar decentemente para melhor compreensibilidade, do que "arrumar" caixas e conectores no ecrã, num esforço infrutífero de fugir ao inevitável "esparguete", como jocosamente se usa caracterizar muitos dos *patches*.

Porém, os referidos sistemas possuem *interfaces* relativamente robustas entre o código e a partitura expressa em notação tradicional. Para além disso, como são especialmente vocacionados para a música, permitem obter um imediato *feedback* auditivo dos cálculos efectuados, bem como permitem exportar os resultados em formatos de ficheiros correntemente utilizados na informática musical, para além de oferecerem muitas outras funcionalidades específicas da prática composicional.

Assim, o *workflow* que segui no trabalho a que esta dissertação diz respeito consistiu, geral-

[12] Ou mais correctamente, *estendem* a linguagem de programação Common Lisp—tal como sucede com qualquer programa nessa linguagem—oferecendo um correspondente gráfico para muitas das funções presentes no *standard*, bem como outras funções úteis e ferramentas específicas para o domínio da composição. A *interface* gráfica que consiste na principal vantagem destes sistemas não impede que em ambos os casos também sejam fornecidos meios de integrar directamente código em Common Lisp.

[13] https://www.ircam.fr/

mente e como se verá melhor *infra*, na programação dos conceitos directamente em Common Lisp, cujas funções foram depois importadas para PWGL ou OpenMusic, com vista a relacioná-las com a notação musical e com a *audição* dos resultados.

Mais recentemente, alguns compositores têm começado a integrar ferramentas de CAAC com a biblioteca *bach: automatic composer's helper*[14] (Agostini & Ghisi, 2013; Agostini & Ghisi, 2015; Trapani, 2017). Esta biblioteca funciona integrada no popular ambiente de programação visual Max, assim tirando partido das possibilidades da programação reactiva[15] bem como da combinação com outras ferramentas desenvolvidas para Max, que desde logo com aquelas, amplamente disponíveis, para gerar e processar áudio.

Escusado é dizer que o estado da arte no que diz respeito aos sistemas tecnológicos disponíveis é tremendamente dinâmico, sendo inevitável que a breve trecho surjam programas novos, enquanto que muitos dos actuais se tornarão obsoletos.

[14] http://www.bachproject.net/

[15] Um paradigma de programação em que, muito sumariamente, alterações verificadas a montante são automaticamente propagadas, em tempo real, pelo *data flow*—pelas conexões lógicas que constituem o programa. Para além dos ambientes Max e PureData, também o OpenMusic possui, desde a versão 6.9, um modelo reactivo.

— 3 —

Concepção de Sistemas Personalizados

3.1 Uma Biblioteca de Funções.

Ao longo dos dois últimos anos fui explorando ideias de formalização e composição algorítmica, e em simultâneo recolhi os processos computacionais assim criados numa "biblioteca" de funções, cujo código integra, em anexo, o presente trabalho. Na secção que aqui se introduz vamos passar em revista algumas das ideias principais aí desenvolvidas, bem como a respectiva mobilização especificamente na escrita das obras musicais que constituem o núcleo desta dissertação.

A biblioteca compõe-se de *funções*, no sentido de sub-rotinas (sequências unitárias de instruções) que aceitam um certo número de *inputs* (os argumentos), e devolvem um ou mais *outputs*. Os valores devolvidos dependem apenas dos argumentos passados à função, que assim não produz "efeitos secundários" (*side effects*), isto é, não altera o estado interno do programa que a chama.

A biblioteca pode ser integrada nos sistemas PWGL e OpenMusic. Ambos oferecem diversas maneiras de correr código em Common Lisp, e utilizá-lo na construção de *patches*, em combinação com as demais ferramentas que os programas já incluem. Outra forma de utilizar a biblioteca é directamente numa implementação da linguagem Common Lisp. Estas utilizam tipicamente um ambiente interactivo denominado REPL (*read-eval-print loop*). O ficheiro com o código é carregado na memória e compilado, e o computador aceita entradas por parte do utilizador (*read*), avalia-as (*eval(uate)*) e mostra os resultados(*print*). Para facilitar este processo existem vários ambientes integrados de desenvolvimento, quer "comerciais" quer gratuitos. A quase totalidade da programação e experimentação com a biblioteca foi realizada assim. Utilizei a implementação SBCL (Steel Bank Common Lisp)[1], em conjugação com o editor de texto Emacs[2] e o modo Slime[3], bem como vários outros *plug-ins* que muito expandem as funcionalidades básicas do editor de texto.

Pode-se dizer que se trata aqui de uma biblioteca *pessoal*. Não obstante o código se encon-

[1] http://www.sbcl.org/
[2] https://www.gnu.org/software/emacs/
[3] https://common-lisp.net/project/slime/

trar minimamente documentado, não me preocupei em fornecer uma *interface* que facilitasse a utilização por terceiros. Não obstante, procurei que as funções fossem definidas com um nível elevado de abstracção. Isto é, que se configurassem de forma não só a resolver ou investigar um problema musical imediato, mas também que pudessem ser utilizadas noutros contextos, por mim ou por outrem, e bem assim integradas modularmente em programas mais complexos. Tanto o PWGL como o OpenMusic permitem a elaboração de bibliotecas segundo um formato próprio, o que tem a vantagem de uma maior integração no sistema de programação visual, definindo-se cada função nos termos do paradigma da programação orientada por objectos. Podemos encontrar ainda um estrato suplementar de abstracção[4] para escrever bibliotecas directamente utilizáveis tanto em Common Lisp "puro" como nos ambientes PWGL e OpenMusic. Porém, não fiz uso de qualquer desses sistemas, já que tal não se afigurou necessário para o *workflow* que segui. No futuro, admito configurar a biblioteca ou parte dela de forma a ser mais facilmente partilhável e aproveitável por outros.

Não obstante tratar-se de uma biblioteca pessoal, encontra-se distribuída publicamente[5] sob o nome Trocadolib, em formato de código aberto, com as permissões correspondentes à licença GNU GPL v3.0.

Os vários temas que vamos abordar em seguida têm em comum algumas ideias principais. Uma é a de partir sempre de algoritmos simples e que possuem a mais ampla expressão na prática musical de praticamente todas as épocas e lugares. Procurei depois fazer uma série crescente de pequenas mutações, que posso classificar como "perversões" ou "ruído algorítmico", para obter diferentes resultados. Outra ideia prende-se com a exploração teórica e especialmente combinatória, usando o poder de computação que está hoje ao nosso dispor. Neste particular, sigo muitas vezes um método *brute force*, tirando partido do crescente poder computacional da tecnologia hoje ao dispor. Gera-se um número muito grande de material musical em potência, verificando caso a caso se se trata de uma das soluções pretendidas ou privilegiadas. Há assim um fluxo em "funil", de um cosmos incaracterístico para uma mais pequena quantidade humanamente apreensível de resultados, tidos como "melhores".

Tendo em consideração a natureza e os objectivos desta dissertação, que não é um manual técnico, não se vão analisar todas as funções presentes na biblioteca. Nem, em cada uma, nos vamos poder ater a explicar pormenorizadamente as funcionalidades oferecidas e os métodos utilizados na programação. Vamos descrever tão-só uma amostra das ferramentas—aquelas com maior expressão na composição das obras musicais em anexo, ou com maior interesse por se relacionarem com estudos teóricos recentes.

[4] Escrito por Kilian Sprotte e disponível em https://github.com/kisp/ompw.
[5] https://github.com/ntrocado/trocadolib

3.2 A Ordem, o Cosmos, as Agulhas, o Palheiro

3.2.1 Rotações.

Se dispusermos os membros de um conjunto ordenado ao longo de uma circunferência e realizarmos um deslocamento circular, obtemos o que em teoria combinatória se denomina *permutação circular*. Uma permutação deste tipo considera-se idêntica à original quando tomadas em conta apenas as posições relativas dos elementos entre si (Brualdi, 2010, pp. 38-39; Chen, Koh & Khee-Meng, 1992, pp. 12-13). Já se nos interessar a posição absoluta de cada um dos elementos, a referida *rotação* produz uma permutação nova, aproximando-se da permutação linear. Dado um conjunto de alturas ordenado de uma primeira nota mais grave para a última mais aguda, a rotação prejudica a ordem original; se a primeira nota se tornar a última, e se pretendermos que essa nota continue a pertencer à mesma classe de alturas, a manutenção da ordem da mais grave para a mais aguda obtém-se transpondo essa nota um número de oitavas tal até que se torne a mais aguda. Funciona aqui o conceito de *equivalência das oitavas*, segundo o qual as alturas separadas por uma ou mais oitavas são comummente percepcionadas como equivalentes (Strauss, 2005, p. 1).

Esta operação é extremamente comum em música, desde logo porque ao colocar como baixo, sucessivamente, cada uma das notas de um acorde, produz o que na literatura relativa à música tonal se designa como *inversões*. Por outro lado, as sucessivas rotações de uma escala produzem os chamados *modos* dessa escala[6].

Como se viu, a rotação faz-se através do intervalo de oitava. Contudo, o algoritmo pode ser "prevertido" se substituirmos o intervalo de oitava por outro intervalo. Assim, a nota mais grave transpõe-se ascendente e sucessivamente por esse outro intervalo até se tornar a mais aguda.

Este processo pode repetir-se iterativamente, pelo menos tantas vezes quantas as notas da colecção. A função `all-rotations` devolve uma lista dessas rotações.

A Figura 3.1 exemplifica com as rotações de uma tríade de dó maior. Em (a) o intervalo é 3, portanto na primeira rotação a nota mais grave é transposta por terceiras menores até se tornar a mais aguda, o que sucede ao fim de três transposições, fixando-se a nota em lá. Depois repete-se o processo com a próxima nota (mi), que também é transposta por terceiras menores até se tornar a mais aguda: si bemol. Em (b) o intervalo é 13 (nonas menores).

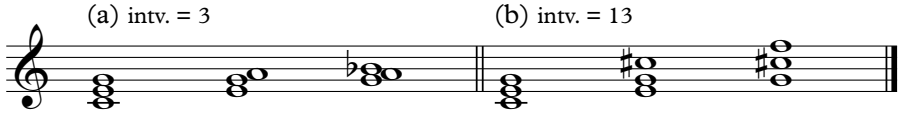


Figura 3.1: Duas rotações de uma tríade de dó maior.

[6]As designações "escala" e "modo" são correntemente utilizadas de forma tudo menos unívoca. No contexto do presente texto, uma escala é uma classe de sucessões ordenadas de alturas, que se instancia num número de modos igual à respectiva cardinalidade, cada qual correspondendo a uma rotação, e em que a primeira nota assume musicalmente maior valor do que as demais.

Pressupondo a repetição iterativa da rotação como a acabamos de a definir, a próxima "perversão" consiste no seguinte: em cada rotação, volta-se a transpor a nota que acabou de se tornar a mais aguda, mas desta feita por um intervalo que se vai alterando conforme a iteração em causa. Aqui parti do princípio de que este segundo intervalo é 1 (segunda menor) na primeira rotação, e vai aumentando meio-tom em cada rotação. Porém, através de um multiplicador, o intervalo pode ser manipulado. Por exemplo, se o multiplicador for 2, o intervalo será 2 (segunda maior) na primeira rotação, 4 (terceira maior) na segunda, 6 (trítono) na terceira, etc. O multiplicador pode ainda ser negativo ou decimal.

A função `many-rotations` devolve uma sequência assim calculada—*vd.* o exemplo da Figura 3.2.

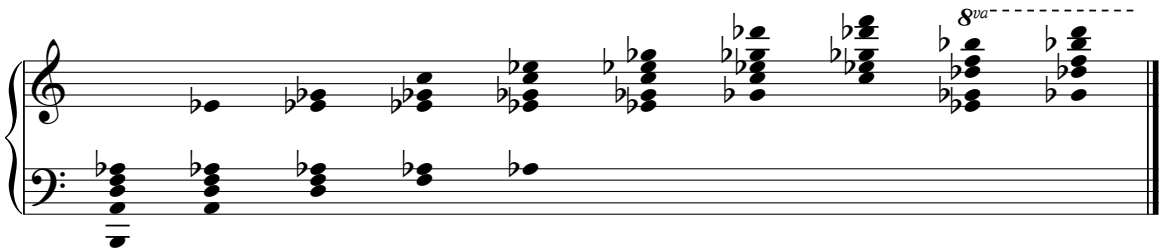


Figura 3.2: Rotações com intervalo = 7 e multiplicador = 0,2, com arredondamento para a escala temperada.

Deste modo, o resultado é uma sequência de colecções. Variando o intervalo ordenador da rotação, o multiplicador e o número de iterações, e reunindo-se todas as sequências resultantes, obtém-se uma enorme quantidade de sequências. A função `many-many-rotations` calcula-as, para todos os multiplicadores ascendendo entre um valor mínimo e máximo, em passos que podem ser decimais. Todas as sequências assim obtidas possuirão uma lógica interna particular, que lhes confere coerência. Mas algumas delas serão mais úteis do ponto de vista musical, outras menos. Impõe-se pois separar o trigo do joio—problema de que se tratará mais à frente.

Outra ideia relativa às rotações é a de transpô-las de forma a manter constante cada uma das notas da colecção original, que funciona como nota *pivot*. Isto é, recolher todas as rotações, transpostas para que a nota mais grave corresponda sempre à nota mais grave da colecção original, depois voltar a recolhê-las, mas transpostas para que a segunda nota mais grave corresponda sempre à segunda nota mais grave da colecção original, e assim sucessivamente. A cada nota equivale assim um conjunto de rotações.

A função `rotation-matrix` gera a matriz assim construída. A Figura 3.3 representa um *patch* PWGL que, partindo de um acorde inicial, nos dá a matriz de rotações desse acorde. Essas rotações serviram de ponto de partida para parte da elaboração harmónica da peça *Dois*. Porém, ao longo do processo composicional alterei a posição dos acordes, adicionei e removi notas, etc. A partitura que se vê na figura está anotada com números que correspondem à ordem pela qual se introduzem os acordes que dão início à peça, num estrato harmónico distribuído inicialmente pelo contrabaixo, trombones, clarinete baixo e por vezes também a flauta.

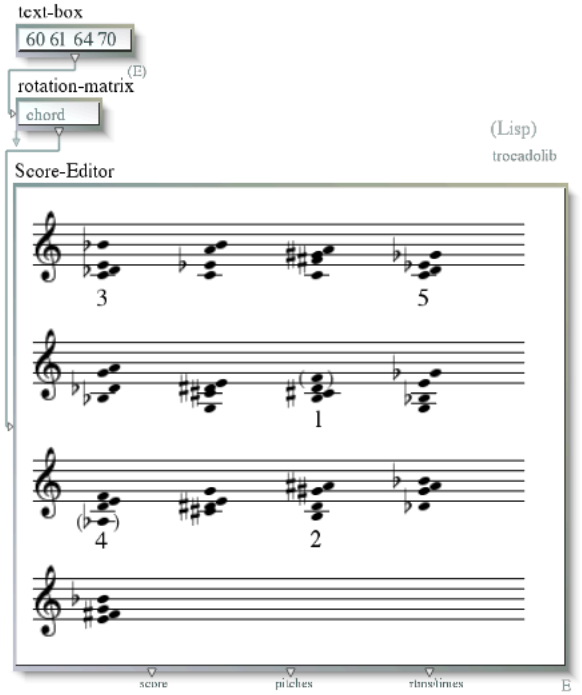


Figura 3.3: *Patch* PWGL para gerar a matriz de rotações de uma colecção.

3.2.2 Expansões.

Por *expansão* designo a transformação de uma colecção ordenada de notas inicial, em que se incrementa consecutivamente o intervalo entre aquelas, meio-tom entre a primeira e a segunda notas, dois meios-tons entre segunda e a terceira, três meios-tons entre a terceira e a quarta, etc. Através de um multiplicador é possível ainda trabalhar com incrementos diferentes do meio-tom—se o multiplicador for, *v.g.*, 3, o intervalo entre a primeira e a segunda notas é incrementado em três meios-tons, seis meios-tons entre a segunda e a terceira, nove meios tons entre a terceira e a quarta, etc.

A expansão pode ser ascendente, descendente, ou simultaneamente ascendente e descendente, com centro numa nota *pivot*. Em qualquer um dos casos, há sempre uma nota que se mantém constante. Essa nota tanto pode transitar, junto com as demais, para o material composicional a utilizar, como também pode ser removida, funcionando, quando *pivot*, como eixo subjacente.

Se aplicarmos este procedimento sucessivas vezes obtemos uma sequência que se vai expandindo cada vez mais, isto é, vai aumentando o intervalo entre as diferentes notas. Para além disto, cada uma das notas iniciais é transposta sempre pelo mesmo intervalo, gerando aquilo a que é habitual designar-se *ciclo intervalar*, estrutural nomeadamente na música de Berg, Bartók e Stravinsky (Perle, 1977), ou na de Charles Ives (Lambert, 1990), de Varèse (Strauss, 2005), ou ainda na de Thomas Adès (Travers, 2004). Como há uma sobreposição de dois ou mais ciclos intervalares, desdobrando-se na mesma direcção, num alinhamento de nota-contra-nota, trata-se

também de um exemplo de *ciclo alinhado* (Stoecker, 2014).

Se a partir do material inicial calcularmos as expansões para vários multiplicadores, por exemplo para todos de 1 a 11, obtemos um conjunto de muitas sequências. Existe ainda a possibilidade de fixar o multiplicador num número decimal. Neste caso, ou se admite os microtonalismos resultantes, ou no final se arredonda os valores para corresponderem à escala temperada de doze sons.

Seja como for, e tal como sucedeu com as rotações, também aqui será necessário identificar, de entre todas as inúmeras sequências em teoria possíveis, quais é que são mais promissoras, em termos de satisfazer determinados objectivos musicais.

Uma primeira abordagem consiste na identificação do multiplicador que gera os "melhores" resultados. Por exemplo: vamos expandir sucessiva e ascendentemente uma tríade de dó maior, repetindo 10 vezes o processo acima descrito (*vd.* Figura 3.4).

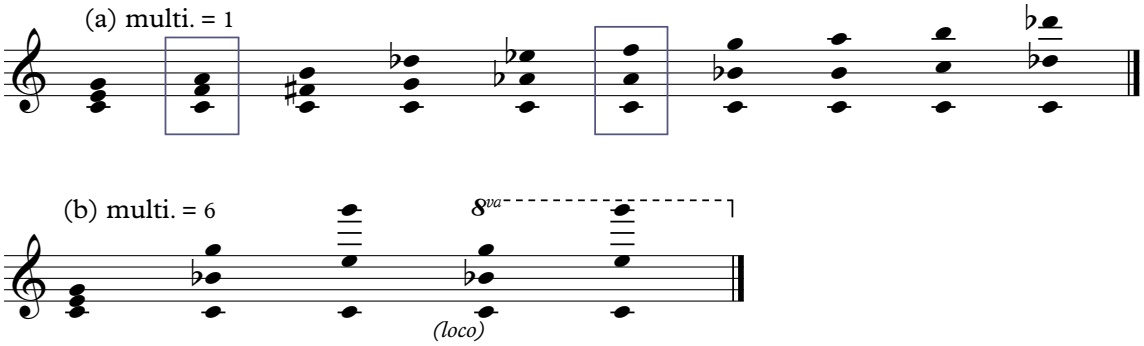


Figura 3.4: Expansões de uma tríade de dó maior, com (a) multiplicador = 1; e (b) multiplicador = 6.

Se o multiplicador for 1, obtemos 10 acordes diferentes, mas o segundo e o sexto acordes possuem exactamente as mesmas classes de alturas: dó, fá e lá—assim, há apenas nove acordes diferentes em *mod12*[7] Já se o multiplicador for 6 verificamos que só há dois acordes diferentes em *mod12*: o original e {dó sol si-bemol}, que se vão alternando. Ora, partindo do princípio que esta mera alternância não nos serve, pois pretendemos privilegiar uma solução em que haja o máximo de diversidade de acordes, isto conduz-nos à questão: qual é o multiplicador que, para um determinado número de iterações, gera o máximo de colecções diferentes?

A função `find-best-expansion` oferece automaticamente a resposta, o que é bastante prático, sobretudo quando se admite um multiplicador com uma ou mais casas decimais, disparando logo o número de sequências a experimentar. A função devolve um triplo resultado: (a) o multiplicador que conduz ao maior número de colecções não ordenadas de classes de alturas diferentes (obtendo-se o máximo de colecções diferentes através de mais do que um multiplicador, a função devolve o valor mais baixo, já que valores mais altos têm a relativa desvantagem de conduzirem rapidamente a extremos de tessitura); (b) o número de colecções diferentes; e (c) o vector de multiplicadores da expansão: um vector de números inteiros, cada qual representando

[7]No espaço cromático de 12 sons com equivalência de oitavas.

o número de colecções diferentes gerados pela expansão, ascendendo da esquerda para a direita para os sucessivos multiplicadores, de 1 até 11, em passos inteiros ou em décimos, centésimos, etc., conforme o número de casas decimais escolhido.

Há outras abordagens para procurar resultados segundo diversas condicionantes, assunto esse que se analisará *infra* (3.2.4).

3.2.3 *Voicing.*

Os processos com os quais nos vimos ocupando produzem sequências de colecções—e essas sequências são tendentes a evoluir na mesma direcção, o que conduz a que logo após um pequeno número de iterações se alcancem notas extremamente agudas ou graves. Impõe-se assim uma reorganização das notas, de forma a constrangi-las a um determinado âmbito pretendido. Muitas vezes isto faz ainda quebrar a unidireccionalidade da sequência, o que altera radicalmente a sonoridade resultante. As técnicas que utilizo para o efeito são perfeitamente quotidianas. As funções `top-limit` e `bottom-limit` oitavam toda a colecção de forma a que não seja ultrapassado um limite superior ou inferior. A função `transpoct` transpõe as notas fora de um âmbito determinado as oitavas necessárias para que fiquem dentro do referido âmbito. A função `closed-position` devolve a colecção em posição fechada.

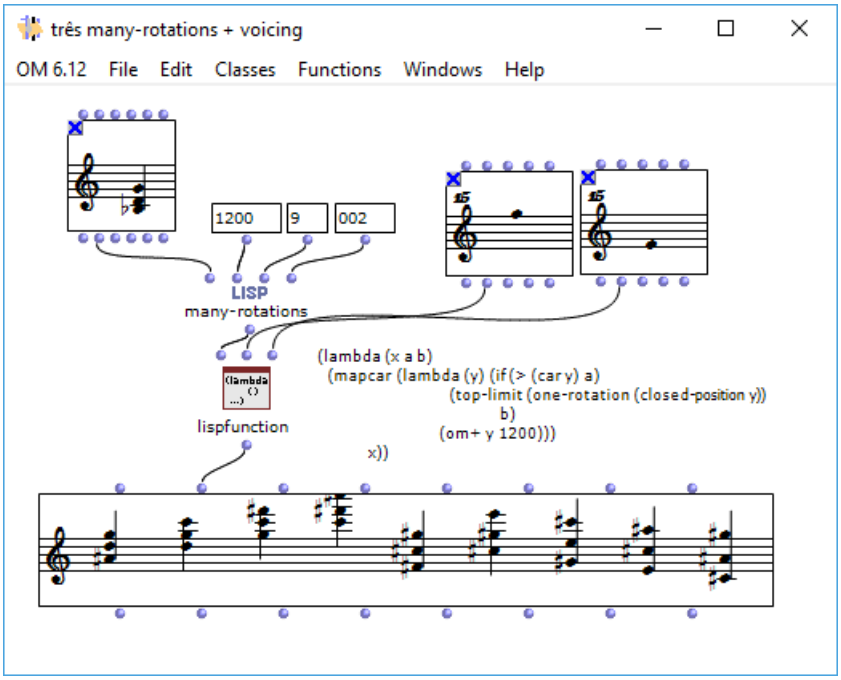


Figura 3.5: *Voicing* das rotações determinado algoritmicamente.

Na Figura 3.5 encontramos um *patch* elaborado em OpenMusic que produz uma sequência de rotações, no caso com multiplicador 2, depois submetida a um processo *ad hoc*, programado ao fim de várias experiências, que combina as funções `top-limit` e `closed-position`. A colecção inicial é a tríade de sol menor que surge no início da melodia da peça *Três*. A sequência

resultante foi utilizada integralmente na secção "B" dessa peça, e fragmentada em múltiplos outros pontos, sendo o exemplo mais expressivo o dos compassos 149-154.

3.2.4 Análise e classificação de material harmónico.

Como vimos, através dos processos de rotação e expansão obtemos um elevado número de sequências de colecções, pelo que será necessário encontrar, de entre elas, aquelas que cumprem determinadas condições, definidas de acordo com os específicos objectivos musicais que se pretendem alcançar.

Mas para tanto há que abordar uma questão prévia, que é a de estabelecer os critérios pelos quais se há-de classificar o material "em bruto", de forma a dele extrair as soluções subjectivamente tidas como "melhores". Com este desiderato, é importante utilizar uma classificação quantitativa que permita comparar numericamente e ordenar as várias hipóteses. Trata-se então de atribuir um valor—uma *pontuação*—a cada uma das colecções previamente obtidas, sendo depois de esperar que os resultados mais interessantes se irão encontrar entre aqueles que obtiveram pontuação superior.

Conteúdo intervalar.

A função `interval-score` atribui uma pontuação conforme o valor dado a cada um dos intervalos presentes numa colecção. A colecção é ordenada da nota mais grave para a mais aguda, e os intervalos são considerados ascendentemente em *mod12* (as nonas são iguais às segundas, etc.). Por exemplo, se atribuirmos o valor 50 às segundas menores, 10 às terceiras maiores e 100 às sétimas maiores, um acorde maior de sétima no estado fundamental terá como pontuação 120, enquanto que o mesmo acorde na primeira inversão terá como pontuação 60. Pode assim dar-se preferência a determinadas sonoridades em detrimento de outras partindo do conteúdo intervalar, o que corresponde, aliás, a uma estratégia geral bem corrente, já que "a qualidade de uma sonoridade pode ser grossamente sumariada pelo enumerar de todos os intervalos que ela contém" (Strauss, 2005, p. 11).

Diversidade intervalar.

Para além dos específicos intervalos presentes numa colecção, pode interessar também averiguar da respectiva variedade. Amiúde, o resultado musical depende da intensificação de um só ou poucos intervalos; ou, pelo contrário, pretende-se o resultado inverso e privilegiam-se estruturas em que não haja um reduzido número de intervalos repetidos, ou pelo menos que de entre os intervalos repetidos não haja um que sobressaia e domine a sonoridade.

Analogamente, medir quantos tipos diferentes existem num determinado conjunto de dados e quão equilibrada é a sua distribuição é algo que interessa a vários ramos do conhecimento, e em particular à ecologia. Aqui, o objectivo é quantificar a concentração ou diversidade de

indivíduos (animais, plantas) de uma certa população. Para esse efeito, um dos índices ao dispor dos investigadores é o proposto por Simpson (1949), que permite estimar a probabilidade de dois indivíduos escolhidos aleatoria e independentemente pertencerem ao mesmo grupo, assim se medindo a diversidade da população em causa. O *índice de Simpson* obtém-se pela fórmula:

$$D = \frac{\sum n(n-1)}{N(N-1)}$$

em que n é o número total de organismos de uma determinada espécie e N é o número total de organismos de todas as espécies. Assim, é tido em conta o número de espécies presentes, bem como a relativa abundância de cada espécie. A diversidade aumenta quando aumentam a riqueza das variadas espécies e o respectivo equilíbrio mútuo. Para que o índice faça corresponder a diversidade infinita a 1 e a ausência de diversidade a 0, o que é mais intuitivo, pode tomar-se como $1 - D$.

Quando pretendemos medir a diversidade de intervalos numa colecção, trata-se fundamentalmente do mesmo problema. Assim, faz todo o sentido mobilizar o mesmo índice e a respectiva fórmula.

A função `simpsons-index` devolve o índice de Simpson calculado para uma colecção ordenada. Por exemplo, uma série cromática de oito notas consecutivas tem o valor 0,8518519, enquanto que uma escala maior tem o valor 0,9206349, assim se verificando que esta é mais diversa do que aquela. Como são considerados os intervalos formados entre cada uma das notas e as demais, é impossível na prática obter valores inferiores a 0,5. Isto poderia ser facilmente corrigido pela modificação da fórmula ou pela medição da diversidade através de um índice diferente. Porém, a função tal qual se encontra é perfeitamente válida para confrontar colecções e comparar a sua diversidade, e o resultado pode sempre ser posteriormente transposto para diferentes escalas de magnitude.

Coincidência harmónica.

Ideias de consonância e dissonância permanecem hoje uma *vexata quaestio*. No fio dos séculos, esses termos não vêm sendo utilizados de forma constante (Tenney, 1988), mas já para os filósofos da Antiguidade a consonância se identificava com proporções harmónicas formadas pelos números fraccionais mais simples, cristalizando-se depois em Rameau (1722, p. 5) a ideia de se estabelecer a consonância como uma relação entre os vários sons face a um som fundamental, e posteriormente a percepção de consonância foi explicada por von Helmholtz (1877) com base nos batimentos produzidos entre parciais. Hoje coexistem pelo menos seis diferentes explicações para aquilo a que chamamos consonância e dissonância, cada qual com relativos pontos fortes e limitações (Sethares, 2005, p. 85).

Em todo o caso, a percepção de uma sonoridade e o próprio fenómeno acústico têm como aspecto central a série harmónica. Por isso, é interessante classificar colecções conforme o grau de congruência face a um determinado espectro harmónico.

A função `harmonic-coincidence` compara uma colecção com a série harmónica construída a partir de uma dada nota fundamental, devolvendo o recíproco do somatório de todas as distâncias entre cada uma das notas da colecção e o parcial mais próximo, proporcionalmente ao número de notas da colecção. Assim, um valor de 1 corresponde a uma perfeita coincidência e um valor de 0 à ausência de coincidência. Através da *keyword* `compare-spectra` pode especificar-se que a comparação não é só entre dadas alturas e a série harmónica a partir da fundamental, mas entre a série harmónica de cada uma das alturas e a série harmónica a partir da fundamental. Para facilitar operações em que se pretende valorizar a dissonância, a função aceita ainda que se fixe em `t` a *keyword* `inverse`, devolvendo então valores superiores para menores graus de coincidência.

3.2.5 Pesquisa automática e ordenação preferencial de soluções.

Os processos de rotação e expansão, que analisámos *supra*, geram potencialmente uma quantidade de material muito significativa. É na prática impossível o compositor ater-se a cada uma das soluções, seleccionando aquelas que lhe são mais interessantes, como se fossem a proverbial "agulha no palheiro". Por isso, vimos também formas de classificar todas essas soluções segundo diversos critérios. Falta então o passo final, que é ordená-las de forma a que as preferenciais apareçam primeiro.

Para isto serve a função `sort-chords`, que aceita uma lista de funções classificativas e uma lista de ponderações, já que se pode pretender utilizar em simultâneo diferentes critérios de classificação, mas valorizando-se uns mais do que outros. Devolve uma lista (ou *tree*) com a seguinte configuração:

((<pontuação$_1$> (<colecção$_1$>))
...
(<pontuação$_n$> (<colecção$_n$>))),

ordenada descendentemente da maior para a menor pontuação.

Quando em causa estão não apenas colecções mas sim *sequências* de colecções, a função que as classifica e ordena é `sort-sequences`, devolvendo:

((<pontuação$_1$> ((colecção$_{1,1}$>) ... (colecção$_{1,n}$)))
...
(<pontuação$_i$> ((colecção$_{i,1}$>) ... (colecção$_{i,n}$)))),

também ordenada descendentemente da maior para a menor pontuação.

Assim, o sistema aqui explicitado funciona em três fases sucessivas: (a) geração de um cosmos de possibilidades; (b) processamento ou transformação dessas possibilidades; (c) ordenação preferencial segundo critérios estabelecidos.

Uma outra abordagem a este problema seria possível através da chamada *programação por constrangimentos* (*constraint programming*) (Rossi, van Beek & Walsh, 2008, p. 181). Trata-

se de um paradigma no qual, muito resumidamente, os constrangimentos constituem relações lógicas declaradas pelo utilizador, versando sobre um conjunto de variáveis de decisão— incógnitas que variam num determinado domínio. Assim descrito o denominado problema de satisfação por constrangimentos (*Constraint Satisfaction Problem, CSP*) numa notação própria, há um algoritmo solucionador (*solver*) que se encarrega de procurar valores para as variáveis que satisfaçam os constrangimentos. Este paradigma presta-se bem à solução de problemas combinatórios atinentes à composição musical, por permitir a declaração de regras composicionais, segundo as quais será restringido um largo conjunto de potenciais soluções. Por isso, tem sido alvo de intensa atenção por parte dos investigadores.

Podemos encontrar um levantamento dos sistemas musicais desenvolvidos e dos principais aspectos deste domínio em Anders e Miranda (2011) e em Fernández e Vico (2013, pp. 530-534). A programação por constrangimentos é tipicamente integrada numa linguagem de programação de alto nível. Entre muitas outras implementações, são de mencionar OMClouds (Truchet, Assayag & Codognet, 2003), que é distribuída conjuntamente com o OpenMusic, e PWConstraints (Laurson, 1999; Laurson, Kuuskankare & Kuitunen, 2005), que integra o ambiente PWGL.

Não obstante, não senti necessidade de lançar mão deste paradigma, que exige a aprendizagem de uma notação particular de forma a exprimir os CSPs. Cada implementação tem a sua notação específica, o que impossibilita a portabilidade de um sistema para outro, e torna o risco da obsolescência do código mais relevante do que a utilização directa de uma linguagem duradoura e com especificação estável como é o Common Lisp. Por outro lado, e como cada implementação possui naturalmente as suas limitações, o método que segui apresenta-se como mais flexível, e adaptado ao concreto problema a solucionar.

Passemos agora a alguns breves exemplos de como as funções se podem combinar.

Como se pode ver no *patch* de OpenMusic representado na Figura 3.6, são geradas várias sequências de expansões com vários multiplicadores e *pivot* na nota mi. De entre elas, o programa prefere aquela cujos membros possuam a valorização de intervalos fornecida à função `interval-score`, numa ponderação de 0,25, e apresentem uma coincidência harmónica com a nota dó3, numa ponderação de 0,75. As notas são depois transpostas de forma a situarem-se num âmbito pretendido. O resultado final foi utilizado, com poucas alterações, na peça *Três*, compassos 193-196. Na mesma peça explorei extensivamente esta ideia, de combinar expansões ou rotações com a pesquisa automática através do conteúdo e diversidade intervalares, bem como através da coincidência harmónica, construindo *patches* como os das Figuras 3.6 e 3.7. Encontramos exemplos disso nas seguintes passagens: compassos 34-35, 38-39, 118-125, 130-133, 201-204.

Na escrita de *Efémero*, construí o *patch* representado na Figura 3.7 para gerar a sequência de acordes que surge na parte inferior da imagem. No canto superior esquerdo está o acorde que funciona como ponto de partida (também é o acorde inicial da peça). É calculado um manancial de sequências de rotações, que seguidamente são processadas pelas funções `transpoct-seq` e `top-limit`, ambas incluídas na caixa "lispfunction" (lado esquerdo). As sequências de acor-

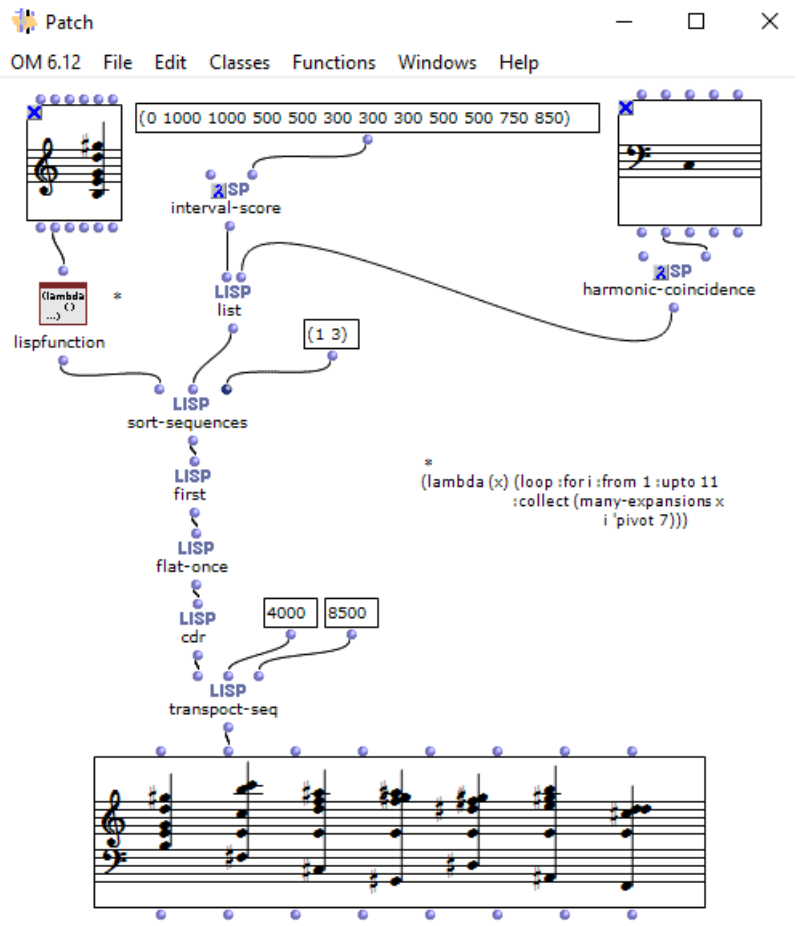


Figura 3.6: *Patch* de pesquisa automática de sequências de expansões, com funções classificadoras ponderadas.

des são depois ordenadas ascendentemente pelo número de classes de alturas repetidas em cada acorde, e os resultados apresentados na partitura. É possível verificar que algumas caixas apresentam um rebordo cor-de-rosa. Significa isso que nelas está activada a programação reactiva, o que torna imediata a apresentação do resultado sempre que se altera o número ordinal (também a cor-de-rosa, do lado esquerdo). Após experimentar vários números, acabei por preferir o 0, como está na imagem, que aliás corresponde à solução inicialmente oferecida pela máquina—aquela que possui menos classes de alturas repetidas em cada acorde. Esta sequência de acordes transitou tal qual para a secção que vai desde o terceiro tempo do compasso 109 até ao compasso 120 da peça.

3.3 Propriedades Geométricas

3.3.1 Excentricidade rítmica.

Na análise de ciclos rítmicos repetitivos, formados por um certo número de ataques que ocorrem distribuídos em certo número de pulsações, há um tipo particular de assimetria que consiste

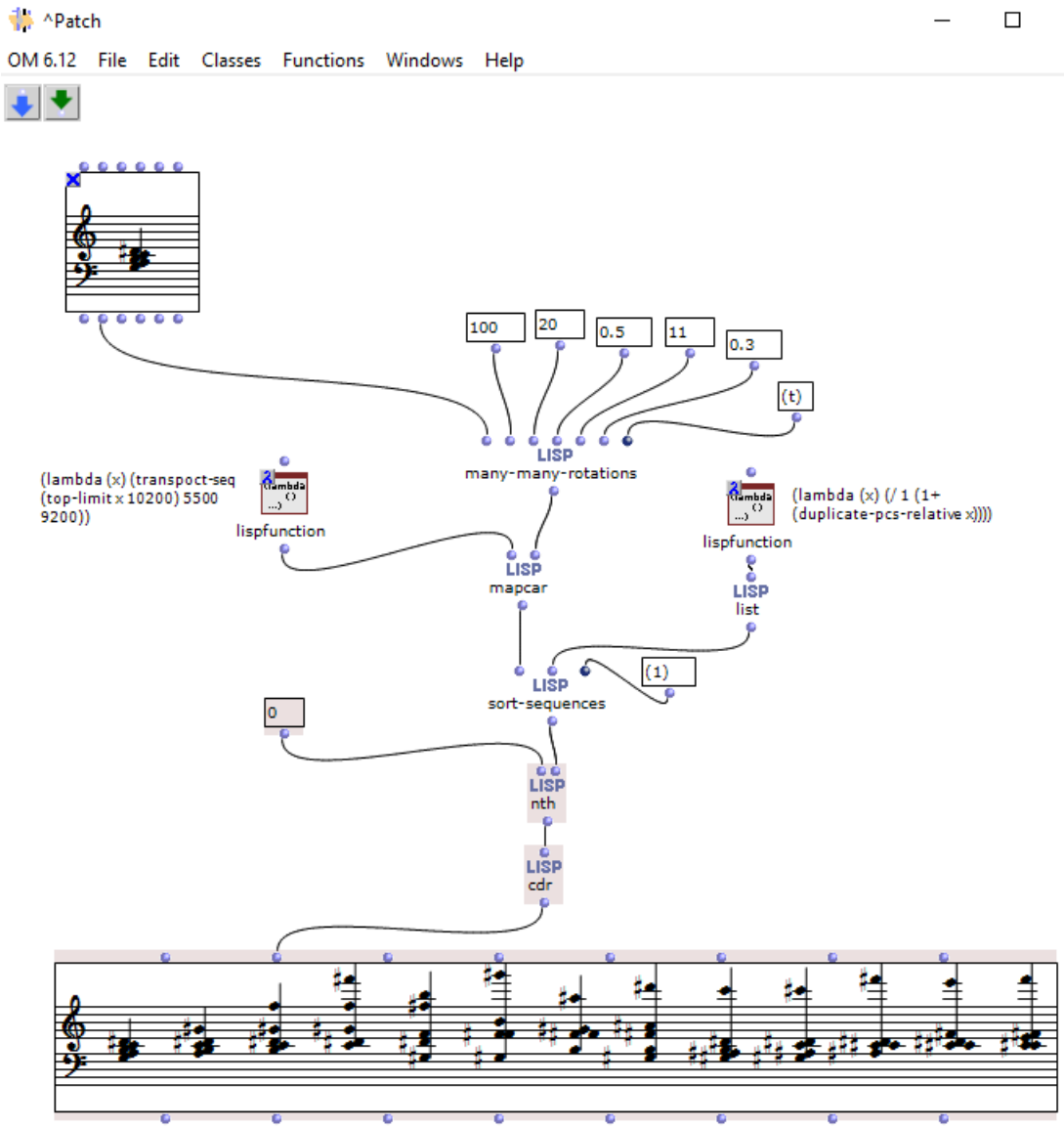


Figura 3.7: *Patch* de pesquisa automática de sequências de rotações.

no seguinte: num número par de pulsações, não há dois ataques que dividam o ciclo em dois segmentos de igual duração (Toussaint, 2013, p. 85). Ou dito de outra forma, se se dispuser as pulsações num círculo e se se tentar quebrar o círculo em duas partes, a partir dos pontos que correspondem aos ataques, não é possível ficar com duas partes iguais (Chemillier, 2002, p. 176). Ou dito de outra forma ainda, um ciclo rítmico com um período de $2n$ pulsações possui este tipo de assimetria se as posições x e $x + n$ não contiverem ambas um ataque (Hall & Klingsberg, 2004).

Como se trata de ritmos que se repetem de forma cíclica, é conveniente (e comum) representá-los num círculo, em cujo perímetro se distribuem as pulsações, marcando-se a negro aquelas que correspondem a ataques. A Figura 3.8 mostra exemplos de ritmos assim representados, que possuem excentricidade rítmica. Já na Figura 3.9 encontramos três ciclos que não possuem excentricidade rítmica. Os ataques que formam os eixos 0/6 e 3/9, no exemplo da esquerda, e

4/10 no exemplo do centro, dividem o círculo ao meio. O exemplo da direita tem um número ímpar de pulsações, sendo então impossível haver dois ataques em posições diametralmente opostas—rigorosamente, não faz sequer sentido falar-se aqui da propriedade de que estamos a tratar.

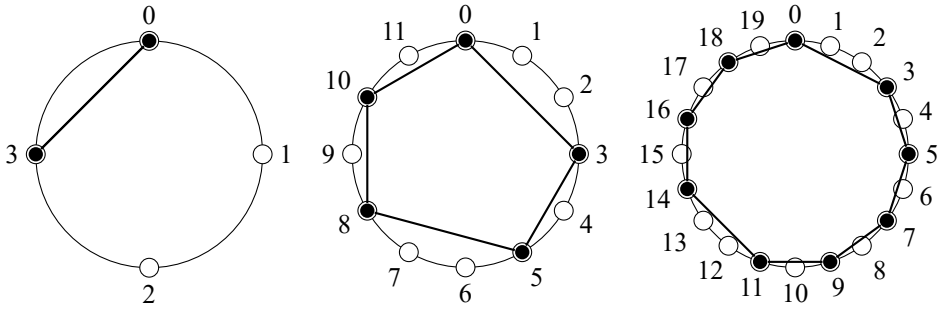


Figura 3.8: Três exemplos de ciclos com excentricidade rítmica.

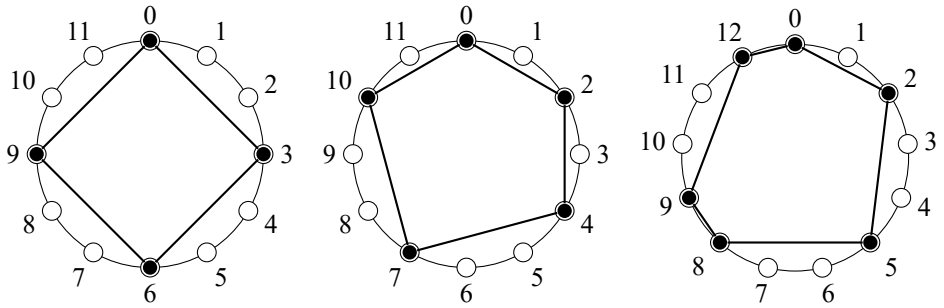


Figura 3.9: Três exemplos de ciclos sem excentricidade rítmica.

Arom (2004, p. 246) encontrou este tipo de assimetria na música tradicional da África Central, em particular entre os Pigmeus Aka, e baptizou-a como *rhythmic oddity*, o que na nossa língua podemos designar como *excentricidade rítmica*. Toussaint (2013, pp. 86-91) observou que, para além do caso dos Pigmeus Aka, a excentricidade rítmica não é muito frequente nas músicas do mundo, quando compreendida de forma binária—um ritmo *tem* ou *não tem* essa propriedade—mas está presente em algum grau na música tradicional de inúmeros povos. Reformulou-a assim como uma função que mede *quão* excêntrico é um ritmo, com base no número de partições do ciclo por ataques antipodais, verificando uma preferência, nos mais variados pontos do globo, por ritmos em que há poucas partições, portanto alta excentricidade. Por outro lado, esta propriedade não é, de forma alguma, exclusiva da música tradicional, já que a encontramos também no material rítmico utilizado por compositores como Ligeti (Taylor, 2012).

Relativamente à forma de gerar ritmos com excentricidade, Toussaint (2013, pp. 86 e 91-94) refere dois algoritmos. O primeiro, denominado *Walk*, corresponde apenas a colocar, para um ritmo de n pulsações, menos do que $n/2$ ataques adjacentes. Claramente, e como o autor assinala, ritmos assim construídos não são particularmente interessantes do ponto de vista

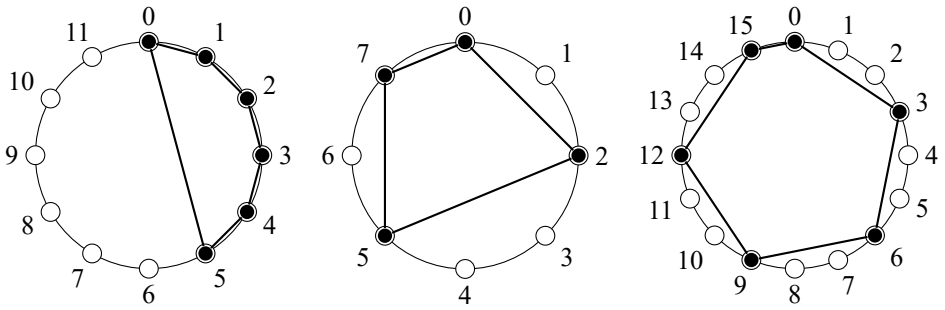


Figura 3.10: Um ciclo produzido pelo algoritmo *Walk* (esquerda) e dois produzidos pelo algoritmo *Hop-and-Jump* (centro e direita).

musical. O segundo algoritmo, *Hop-and-Jump*, coloca o primeiro ataque na pulsação zero, marcando a pulsação diametralmente oposta como indisponível. Depois avança a partir do zero um número determinado de pulsações (*hop*, pulo), colocando aí ataques, e marcando sempre a pulsação diametralmente oposta como indisponível, salvo quando isso já não for possível, caso em que se passa directamente para a próxima pulsação disponível (*jump*, salto), voltando-se depois aos "pulos" da dimensão inicial, até se percorrer todo o ciclo. Na Figura 3.10 encontramos um exemplo do algoritmo *Walk*, com doze pulsações e cinco ataques consecutivos (esquerda), e dois exemplos do algoritmo *Hop-and-Jump*, com oito pulsos e "pulos" de dimensão dois (centro), o que produz um ciclo de quatro ataques, e com 16 pulsos e "pulos" de dimensão três, o que produz um ciclo de seis ataques.

Este algoritmo tem porém a desvantagem de, para um certo número total de pulsações, gerar um número limitado de possibilidades, já que a única variável é o valor que atribuimos aos "pulos" e a cada valor corresponde apenas uma solução. Por esse motivo, pode ser interessante gerar *todos* os ciclos rítmicos de dada dimensão que possuam excentricidade. Uma primeira abordagem, que podemos classificar como *brute force*, consiste em enumerar, um a um, todos os ciclos rítmicos possíveis, verificando, para cada caso, se se verifica ou não a excentricidade. Um ciclo rítmico pode ser descrito numa sequência de uns e zeros—1 para os ataques e 0 para as pulsações sem ataque. Por exemplo, os ciclos da Figura 3.10 seriam representados assim (da esquerda para a direita): *111111000000*, *10100101* e *1001001001001001*. Ora, um qualquer número expresso no sistema binário (ou de *base-2*), é também representado por dois símbolos diferentes, tipicamente o 1 e o 0. Assim, uma forma simples de obter todos os ciclos com duração n é enumerar em base-2 todos os números inteiros entre 1 e $2^n - 1$.

É o que faz a função `all-necklaces`, sendo possível chamá-la com a *keyword* `filter` e indicando uma função booleana para filtrar os resultados. Para o objectivo agora em causa, essa função é `rhythmic-oddity-p`, que verifica se uma dada sequência possui a propriedade da excentricidade rítmica.

Obtêm-se assim facilmente todos os ciclos com excentricidade. Porém, entre eles vamos encontrar exemplares que constituem meras permutações circulares uns dos outros, portanto tidos como equivalentes quando não nos interessa a posição absoluta de cada um dos ataques. Salvo

```
(3 3 2) (3 2 3) (3 2 1 2) (2 3 3) (2 3 2 1) (2 1 2 3) (1 2 3 2) (6 4) (4 6) (4 4 2) (4 3 3) (4
3 1 2) (4 2 4) (4 2 2 2) (4 2 1 3) (3 4 3) (3 4 2 1) (3 3 4) (3 3 3 1) (3 3 1 3) (3 3 1 2 1)
(3 1 3 3) (3 1 2 4) (3 1 2 1 3) (2 4 4) (2 4 3 1) (2 4 2 2) (2 2 4 2) (2 2 2 4) (2 2 2 2 2) (2
1 3 4) (2 1 3 3 1) (1 3 4 2) (1 3 3 3) (1 3 3 1 2) (1 2 4 3) (1 2 1 3 3) (7 5) (5 7) (5 5 2)
(5 4 3) (5 3 4) (5 2 5) (4 5 3) (4 4 4) (4 4 3 1) (4 4 1 3) (4 4 1 2 1) (4 3 5) (4 3 4 1) (4 3
2 3) (4 3 2 2 1) (4 3 1 4) (4 3 1 3 1) (4 3 1 1 3) (4 1 4 3) (4 1 3 4) (4 1 3 1 3) (4 1 2 2 3)
(4 1 2 1 4) (3 5 4) (3 4 5) (3 4 4 1) (3 4 3 2) (3 4 3 1 1) (3 4 1 4) (3 4 1 3 1) (3 4 1 2 2)
(3 2 3 4) (3 2 3 2 2) (3 2 2 3 2) (3 2 2 1 4) (3 2 2 1 2 2) (3 1 4 4) (3 1 4 3 1) (3 1 3 4 1)
(3 1 3 1 4) (3 1 1 3 4) (2 5 5) (2 3 4 3) (2 3 4 1 2) (2 3 2 3 2) (2 3 2 2 3) (2 3 2 2 1 2) (2
2 3 4 1) (2 2 3 2 3) (2 2 3 2 2 1) (2 2 1 4 3) (2 2 1 2 2 3) (2 1 4 4 1) (2 1 4 3 2) (2 1 2 2
3 2) (1 4 4 3) (1 4 4 1 2) (1 4 3 4) (1 4 3 2 2) (1 4 3 1 3) (1 3 4 4) (1 3 4 3 1) (1 3 4 1 3)
(1 3 1 4 3) (1 3 1 3 4) (1 2 2 3 4) (1 2 2 3 2 2) (1 2 1 4 4) (1 1 3 4 3) (8 6) (6 8) (6 6 2)
(6 5 3) (6 4 4) (6 3 5) (6 2 6) (5 6 3) (5 5 4) (5 5 3 1) (5 5 1 3) (5 4 5) (5 4 4 1) (5 4 2
3) (5 4 1 4) (5 3 6) (5 3 5 1) (5 3 3 3) (5 3 2 4) (5 3 1 5) (5 1 5 3) (5 1 4 4) (5 1 3 5) (4
6 4) (4 5 5) (4 5 4 1) (4 5 3 2) (4 5 1 4) (4 4 6) (4 4 5 1) (4 4 4 2) (4 4 4 1 1) (4 4 2 4)
(4 4 2 3 1) (4 4 2 2 2) (4 4 2 2 1 1) (4 4 1 5) (4 4 1 4 1) (4 4 1 3 2) (4 4 1 1 4) (4 4 1 1 2
2) (4 2 4 4) (4 2 4 2 2) (4 2 3 5) (4 2 3 3 2) (4 2 3 1 4) (4 2 3 1 2 2) (4 2 2 4 2) (4 2 2 2
4) (4 2 2 2 2 2) (4 2 2 1 3 2) (4 2 2 1 1 4) (4 1 5 4) (4 1 4 5) (4 1 4 4 1) (4 1 4 1 4) (4 1
3 2 4) (4 1 3 2 3 1) (4 1 1 4 4) (4 1 1 2 2 4) (3 6 5) (3 5 6) (3 5 5 1) (3 5 4 2) (3 5 3 3)
(3 5 1 5) (3 3 5 3) (3 3 3 5) (3 3 3 3 2) (3 3 3 2 3) (3 3 3 2 1 2) (3 3 2 4 2) (3 3 2 3 3) (3
3 2 3 1 2) (3 3 2 1 3 2) (3 3 2 1 2 3) (3 3 2 1 2 1 2) (3 2 4 5) (3 2 4 4 1) (3 2 4 2 3) (3 2
4 2 2 1) (3 2 3 3 3) (3 2 3 3 2 1) (3 2 3 1 4 1) (3 2 3 1 2 3) (3 2 3 1 2 2 1) (3 2 1 3 2 3)
(3 2 1 2 3 3) (3 2 1 2 1 2 3) (3 1 5 5) (3 1 4 4 2) (3 1 4 1 3 2) (3 1 2 3 3 2) (3 1 2 2 4 2)
(3 1 2 2 1 3 2) (2 6 6) (2 4 5 3) (2 4 4 4) (2 4 4 2 2) (2 4 4 1 3) (2 4 4 1 1 2) (2 4 2 4 2)
(2 4 2 3 3) (2 4 2 3 1 2) (2 4 2 2 4) (2 4 2 2 2 2) (2 4 2 2 1 3) (2 3 5 4) (2 3 3 3 3) (2 3 3
3 2 1) (2 3 3 2 4) (2 3 3 2 3 1) (2 3 3 2 1 3) (2 3 3 2 1 2 1) (2 3 1 4 4) (2 3 1 4 1 3) (2 3
1 2 3 3) (2 3 1 2 2 4) (2 3 1 2 2 1 3) (2 2 4 4 2) (2 2 4 4 1 1) (2 2 4 2 4) (2 2 4 2 3 1) (2
2 4 2 2 2) (2 2 2 4 4) (2 2 2 4 2 2) (2 2 2 2 4 2) (2 2 2 2 2 4) (2 2 2 2 2 2 2) (2 2 1 3 2 4)
(2 2 1 3 2 3 1) (2 2 1 1 4 4) (2 1 3 2 4 2) (2 1 3 2 3 3) (2 1 3 2 3 1 2) (2 1 2 3 3 3) (2 1 2
3 3 2 1) (2 1 2 1 2 3 3) (2 1 1 4 4 2) (1 5 5 3) (1 5 4 4) (1 5 3 5) (1 4 5 4) (1 4 4 5) (1 4
4 4 1) (1 4 4 2 3) (1 4 4 2 2 1) (1 4 4 1 4) (1 4 1 4 4) (1 4 1 3 2 3) (1 3 5 5) (1 3 2 4 4)
(1 3 2 4 2 2) (1 3 2 3 3 2) (1 3 2 3 1 4) (1 3 2 3 1 2 2) (1 2 3 3 3 2) (1 2 3 3 2 3) (1 2 3 3
2 1 2) (1 2 2 4 4 1) (1 2 2 4 2 3) (1 2 2 1 3 2 3) (1 2 1 2 3 3 2) (1 1 4 4 4) (1 1 4 4 2 2)
(1 1 2 2 4 4) (9 7) (7 9) (7 7 2) (7 6 3) (7 5 4) (7 4 5) (7 3 6) (7 2 7) (6 7 3) (6 6 4) (6 5
5) (6 4 6) (6 3 7) (5 7 4) (5 6 5) (5 5 6) (5 5 5 1) (5 5 4 2) (5 5 4 1 1) (5 5 2 4) (5 5 2 3
1) (5 5 2 2 2) (5 5 1 5) (5 5 1 4 1) (5 5 1 3 2) (5 5 1 1 4) (5 4 7) (5 4 5 2) (5 4 5 1 1) (5
4 3 4) (5 4 3 3 1) (5 4 3 2 2) (5 4 2 5) (5 4 2 4 1) (5 4 2 3 2) (5 4 2 1 4) (5 4 1 5 1) (5 4
1 4 2) (5 4 1 2 4) (5 4 1 1 5) (5 2 5 4) (5 2 5 2 2) (5 2 4 5) (5 2 4 3 2) (5 2 4 1 4) (5 2 3
4 2) (5 2 3 2 4) (5 2 3 1 5) (5 2 2 5 2) (5 2 2 3 4) (5 2 2 2 5) (5 1 5 5) (5 1 5 4 1) ...
```

Figura 3.11: Ciclos com excentricidade rítmica representados como listas de intervalos entre ataques.

quanto, *v.g.*, há uma sobreposição de dois ou mais ciclos rítmicos, caso em que as combinações resultantes dependem da concreta rotação em que cada um se encontra, numa boa parte das aplicações práticas o que é importante é a posição de cada um dos ataques relativamente aos demais. Por exemplo, para uma duração limite de quatro pulsações, as combinações que possuem excentricidade rítmica são: *10*, *1000*, *1001* e *1100*. Como bem se vê, a não ser se atribuirmos especial significado à ocorrência da primeira pulsação, os dois últimos ciclos mencionados são rotações um do outro e por isso equivalentes. Por outro lado, verificamos ainda que todos os ciclos começam com um ataque, o que é consequência da desconsideração dos zeros à esquerda na notação binária que utilizámos para gerar as combinações—*1*, *01*, *001* e *0001* são a mesma coisa.

Uma forma de obviar a estas circunstâncias é gerar os ciclos rítmicos através de um algoritmo diferente. Partimos então da consideração segundo a qual, para além da notação binária

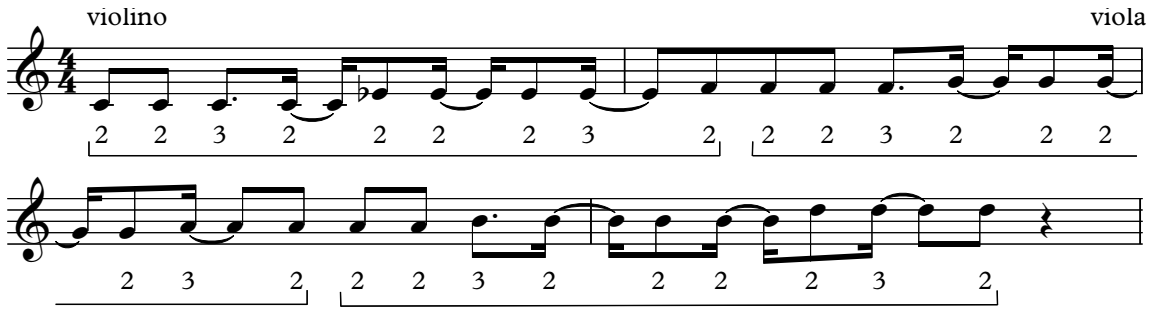


Figura 3.12: Compassos 43-46: ciclo rítmico [2-2-3-2-2-2-2-3-2].

que atribui *1* a um ataque e *0* a uma pulsação sem ataque, também é possível representar os ciclos pela sucessão de intervalos entre ataques (*inter-onset intervals*)—*vd.* Figura 3.11. Isto é, o ciclo rítmico que atrás representámos por *10100101*, passa a ser agora [2-3-2-1], porque entre o primeiro e o segundo ataques decorrem duas pulsações, três entre o segundo e o terceiro ataques, etc. Expressos nesta notação, os ciclos prestam-se ao tratamento através de um conceito da teoria combinatória denominado *palavras de Lyndon* (Chemillier & Truchet, 2003; Chemillier, 2004) ou através de um conceito próximo, as *palavras "rop"* (Jedrzejewski, 2016). Não cabendo aqui distendermo-nos sobre os referidos conceitos, sempre se dirá que através deles se logra seleccionar um dos representantes de cada classe de ciclos cujos membros são invariantes na rotação, uma ideia em tudo semelhante (mas não totalmente coincidente) à *forma normal*, de que se fala correntemente na teoria dos conjuntos das classes de alturas. Implementando uma adaptação do algoritmo delineado por Duval (1988), as funções `lyndon-words` e `lyndon-words-with-duration` devolvem as palavras de Lyndon até dada dimensão, com base num *alfabeto* que, para os nossos efeitos, é constituído pelos números correspondentes aos intervalos entre ataques de que falámos já. Isto é, se o ciclo rítmico é representado por [2-3-2-1], o alfabeto tem como "letras" 2, 3 e 1. Nos estudos acima referidos, o alfabeto é constituído apenas pelos intervalos de duas e três pulsações, por uma questão de simplicidade analítica na explicitação das respectivas propriedades combinatórias e porque os ritmos originalmente estudados por Arom só continham esses períodos. Na falta de vantagem em manter essa limitação, o alfabeto dado à função pode ser estabelecido como um intervalo entre quaisquer números. Assim, regressando à questão anterior, de saber quais as estruturas cíclicas que possuem excentricidade rítmica para uma duração de até quatro pulsações, a resposta é [1-3], [2] e [4], que em notação binária correspondem a *1100*, *10*, e *1000*—conseguimos assim "limpar" a solução de permutações equivalentes.

De uma vasta lista de que a Figura 3.11 é apenas um fragmento recolhi vários ritmos com excentricidade, que utilizei na peça *Efémero*. Muito claramente é o caso dos compassos 33-36, com o ciclo [3-2-3-2-2]; bem como dos compassos 39-42, 53-67, 89-93, com o ciclo [3-2-2-2-2-2-3-2-2-2-2]; compassos 43-46, com o ciclo [2-2-3-2-2-2-2-3-2] (*vd.* Figura 3.12); compassos 109-120, melodia do violino [3-1-3-4-1]; compassos 149-152, [4-2-3-2-5]; compassos 156-157, [3-2-2-3-2].

3.3.2 Isomorfismo tonal–temporal.

Temos estado debruçados sobre questões relacionadas com o ritmo, em especial com ritmos periódicos. Mas naturalmente que encontramos as mesmas representações circulares no domínio das alturas. Diz-se que há um *isomorfismo* entre os dois domínios quando as estruturas presentes num podem ser mapeadas em estruturas presentes no outro, isto é, preservam-se as relações entre os elementos (as alturas das notas preservam-se nas durações rítmicas, e vice-versa). Esta correlação pode ser observada em variadas estruturas musicais (Rahn, 1983, pp. 56-76). Pressing (1983) observou que as similitudes são suficientemente convincentes para justificar a hipótese de que decorrem de processos cognitivos generalizados, afirmando assim que os domínios das alturas e do ritmo se encontram ligados por um *isomorfismo cognitivo*. Porém, isto não é consensual. London (2002), verificando que não existem análogos temporais para os fenómenos da equivalência das oitavas e da equivalência inarmónica, e que, por outro lado, não existem análogos tonais para vários limites à nossa percepção e acuidade temporal, mobiliza representações espaciais e um ramo da matemática denominado teoria dos grafos, com o desiderato de demonstrar que o espaço tonal e o espaço métrico são fundamentalmente não-isomórficos. Do ponto de vista dos estudos sobre percepção, permanece em aberto o debate sobre se a tonalidade e a métrica interagem ou, pelo contrário, funcionam de forma independente na apreensão musical dos ouvintes, existindo evidência experimental que suporta tanto uma como a outra posição (Monahan & Carterette, 1985; Peretz & Kolinsky, 1993; Prince & Schmuckler, 2014; Wen & Krumhansl, 2016).

Certo é, ao menos, que se trata de uma relação que desde há muito vem interessando os compositores. Messiaen, para dar um exemplo, afirma de forma muito clara que os seus modos de transposição limitada realizam, na "direcção vertical", o que os ritmos não-retrogradáveis realizam na "direcção horizontal" (Messiaen, 1944, p. 21).

Exista ou não um paralelo entre idênticas representações hierárquicas nos dois domínios, é em todo caso útil poder lançar mão do mesmo arsenal teórico e analítico tanto para os ritmos como para as alturas, maximizando a relevância potencial dos conceitos desenvolvidos.

Assim, e examinando de novo o ciclo que representámos como *2321*, vemos que a mesma sequência de valores pode significar uma série de intervalos (ou classes de intervalos), pelo que partindo arbitrariamente da nota dó, ascendemos um intervalo de valor *2* para alcançar a nota ré, e depois um intervalo *3* para a nota fá, sucedendo-se *2* para sol e *1* para lá bemol. Se quisermos prosseguir a partir daqui, o mais natural será recomeçar a partir do lá bemol em que havíamos ficado.

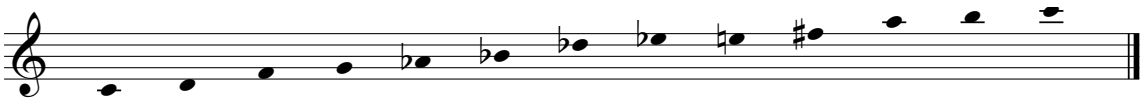


Figura 3.13: Escala [2-3-2-1].

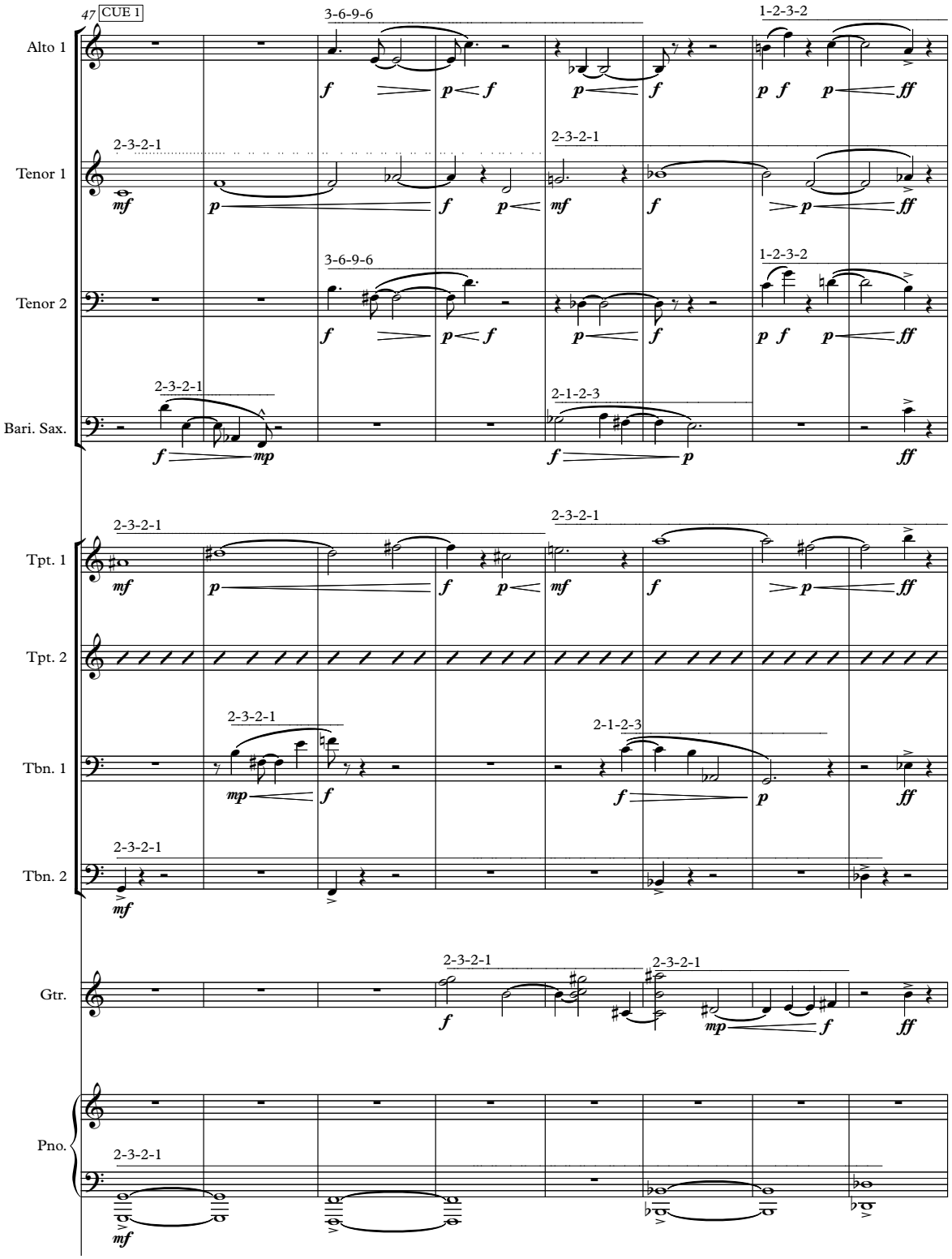


Figura 3.14: *Um*, compassos 47-54. Escala e ritmo [2-3-2-1].

Obtemos então a escala da Figura 3.13, que ao fim de três ciclos regressa ao dó inicial, mas duas oitavas acima, tendo entretanto percorrido as doze diferentes classes de alturas, sem repetir nenhuma. Para todos os ciclos de dimensão (*i.e.*, somatório dos valores dos intervalos) igual a oito, aliás, regressa-se sempre à nota inicial ao fim de três ciclos, sem que se repitam classes de alturas.

Na peça *Um* surge esta escala, associada a sequências de proporções rítmicas [2-3-2-1]—*vd.* fragmento reproduzido na Figura 3.14.

3.3.3 Uniformidade.

Como vimos já *supra*, ritmos como o da Figura 3.10, à esquerda, são em princípio mais pobres, já que a assimetria que os caracteriza se exprime numa mera distribuição desiquilibrada dos ataques entre as pulsações que compõem o ciclo. Assim uma outra propriedade que interessa analisar e quantificar diz respeito à uniformidade (*evenness*) do ciclo, *i.e.*, quão bem distribuídos temporalmente estão os ataques, ou quão bem distribuídas estão as notas, representadas num campo harmónico circular.

Uma primeira aproximação consiste em estabelecer a proporção entre o número de ataques n e o total de pulsações k, face à duração máxima manifestada por um ataque m. A uniformidade é assim dada pela fórmula:

$$U = \frac{1}{m - h + 1},$$

cujos valores variam entre 0 e 1. A função `evenness` calcula esse índice de uniformidade, para um dado ciclo rítmico ou tonal expresso em intervalos entre eventos.

Outra aproximação consiste no seguinte. Block e Douthett (1994), tratando da medida da uniformidade em conjuntos de classes de alturas, propõem a construção de um vector de ponderação (*weighting vector*), através do qual se podem ordenar hierarquicamente os conjuntos conforme a respectiva uniformidade. O método, naturalmente, também serve para ciclos rítmicos. Sendo c o número total de pulsos ou alturas, as coordenadas do vector sucedem-se desde o comprimento da corda da circunferência que conecta cada intervalo de valor 1 (meio-tom, uma pulsação) até ao comprimento da corda da circunferência que conecta cada intervalo de valor $c/2$. Assim, e com recurso à trigonometria, calculam-se as coordenadas do vector $W_c = [w_1, w_2, \ldots, w_{c/2}]$, em que:

$$W_k = 2\sin(\frac{k\pi}{c}).$$

Finalmente, computa-se o produto escalar entre o vector de ponderação e o vector intervalar. A função `evenness-weight` devolve este resultado.

O valor obtido, porém, só permite comparar conjuntos que apresentem igual cardinalidade. Por exemplo, para [3-2-1-4] o valor é 8,93, enquanto que para [3-2-1-4-8] o valor é 12,96, muito embora se trate este de um ciclo menos uniforme, mas com mais ataques. Para ultrapassar esta desvantagem criei ainda a função `evenness-weight-index`, que nos dá a proporção entre a uniformidade, calculada pela função `evenness-weight`, de um dado ciclo, e a uniformidade

máxima possível para um ciclo com a mesma cardinalidade. O valor obtido é agora sempre entre 0 e 1, permitindo a comparação entre conjuntos de cardinalidade diferente. Os ciclos atrás mencionados possuem assim os valores índice de 0,92 e 0,84, respectivamente, sendo patente que o primeiro é, afinal, mais uniforme.

3.3.4 Pesquisa específica.

As propriedades a que vimos aludindo são susceptíveis de serem combinadas, priveligiando soluções de sentidos opostos. A excentricidade vem temperar a uniformidade, excluindo ciclos excessivamente regulares. De facto, não queremos demasiada uniformidade, o que se torna monótono e aborrecido, mas também não queremos uniformidade a menos, que equivale à desordem. À premência de encontrar o ponto certo entre estes dois pólos chama Tymoczko (2011, p. 123), o *princípio Goldilocks*.

Por outro lado, podemos pretender descobrir soluções que se submetam a constrangimentos suplementares. Por exemplo, que não tenham mais do que x ataques consecutivos, ou meios-tons consecutivos.

Com o objectivo de definir uma escala ou campo harmónico, desenhei a função auxiliar `necklace-specific-search` para pesquisar entre todos os ciclos aqueles que possuíssem: (a) âmbito de até 20 meios-tons; (b) excentricidade; (c) pelo menos nove notas; (d) não mais do que dois meios-tons consecutivos; (e) pelo menos uma uniformidade de 0,99, calculada segundo a função `evenness-weight-index`; e (f) todas as classes de alturas diferentes.

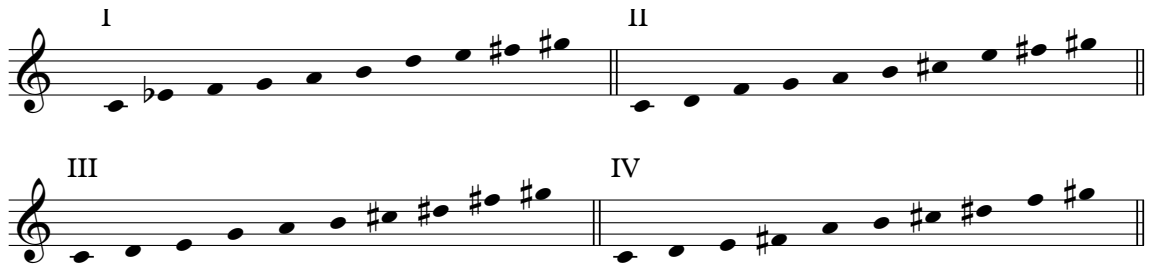


Figura 3.15: Quatro modos de uma escala de duas oitavas, obtida a partir dos conceitos de excentricidade e uniformidade, com constrangimentos suplementares.

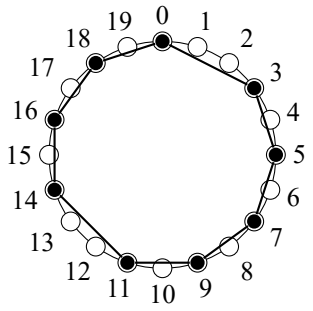


Figura 3.16: Representação circular na escala da Figura 3.15.

As únicas colecções que cumprem integralmente os indicados constrangimentos são as da Figura 3.15. Como se vê, trata-se na verdade de uma solução e respectivas rotações (*vd.* Figura 3.16), portanto podemos falar de quatro modos, numerados de I a IV. Possuem algumas características interessantes, como a combinação entre sequências pentatónicas e outras por tons inteiros, e uma grande diversidade intervalar, não entre notas consecutivas, mas se considerados os intervalos formados entre cada uma das notas e as demais.

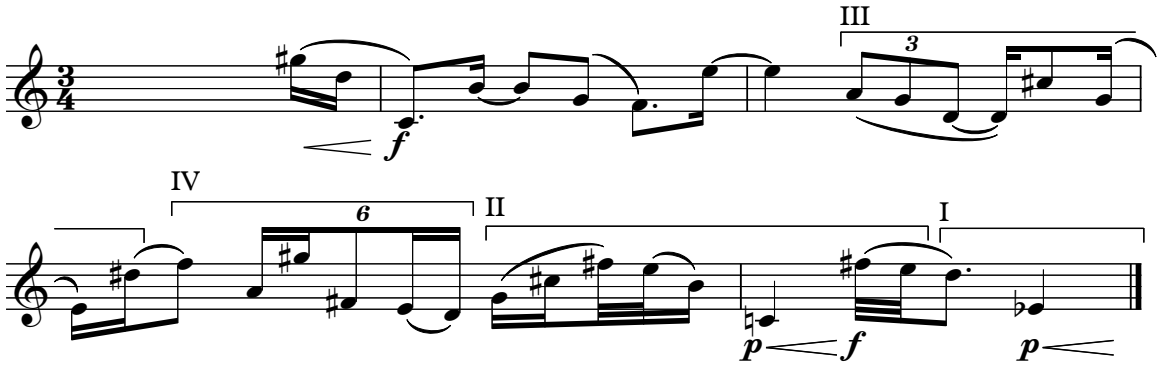


Figura 3.17: Compassos 152-155: sucessão de modos.

Efémero contém extensivamente esta sonoridade, tanto na vertente rítmica como na tonal. Já nos tínhamos cruzado com o modo III no ritmo da Figura 3.12, sendo que, simultaneamente, o fragmento melódico aí representado está no modo I. A melodia inicial, que de alguma forma se mantém presente até ao fim, também está no modo I. Outros exemplos: nos compassos 65, 67 e 67 aparecem, respectivamente, os modos I, IV e II; e os quatro modos sucedem-se no fragmento da Figura 3.17.

3.4 Cadeias de Markov na Amostragem Digital

As cadeias de Markov são um processo estocástico (*i.e.*,dependente de probabilidades), em que um evento se segue ao outro de acordo com uma tabela de transições, previamente definida, que para cada evento estabelece as probabilidades de lhe suceder cada um dos outros. Para uma definição formal e propriedades ver, *v.g.*, Asmussen (2003, pp. 3-7). As cadeias de Markov denominadas de *primeira ordem* só se "lembram" das relações probabilísticas entre um evento anterior e um posterior. As cadeias de ordem n têm em consideração as relações probabilísticas entre n eventos anteriores, com a contrapartida de, a cada incremento, aumentar exponencialmente a dimensão da tabela de transições. A utilização de cadeias de Markov para fins musicais é uma ideia que vem desde os primórdios da composição assistida por computadores. Já em 1956, Pinkerton, mobilizando conceitos da nascente teoria da informação, descreveu um sistema em que a análise de um *corpus* formado por melodias de canções de embalar, contando-se o número de vezes em que cada par de notas consecutivas aparece, conduz à construção de uma tabela representando a probabilidade de uma nota transitar para outra. Depois, é possível cons-

truir novas melodias a partir dessa tabela, escolhendo sucessivamente cada nota através de uma ponderação das probabilidades anteriormente calculadas. Observa então:

> A clara implicação é que podemos construir máquinas para criar música. Um conjunto de tabelas poderia ser construído que compusesse melodias ao estilo de Mozart, ou temas mais típicos de Shostakovich do que o próprio Shostakovich conseguiria escrever. Poderíamos aproximarmo-nos tanto quanto quiséssemos do estilo de qualquer tipo de música sem copiar verdadeiramente as melodias, e alterando as probabilidades, poderíamos desenvolver estilos completamente novos.[8] (p. 86)

E, na verdade, os anos e décadas seguintes são ricos em implementações desta ideia. No quarto andamento do quarteto de cordas *Illiac Suite*, escrito em 1957, e uma das primeiras experiências no domínio da composição assistida por computador, L. A. Hiller e L. M. Isaacson utilizaram tabelas de probabilidades para controlar a distribuição de intervalos melódicos nas quatro vozes, considerando relações de consonância e dissonância (Hiller & Isaacson, 1958; para uma visão mais recente sobre a obra, Sandred, Laurson & Kuuskankare, 2009).

Ao longo dos anos manteve-se popular a utilização deste tipo de modelos na composição algorítmica, por vezes em combinação com outras técnicas. Tanto Nierhaus (2009, pp. 67-81) como Fernández e Vico (2013, pp. 536-541) fazem um levantamento dos inúmeros sistemas assim concebidos (e descritos em publicações académicas). Entre as aplicações musicais, encontramos: formalização da melodia, harmonia ou ritmo, composição de sarabandas ou de corais ao estilo de Bach, "improvisações" jazz, ritmos de *tabla*, contraponto, improvisação interactiva, etc. Como é necessária uma prévia análise de um *corpus*, com o qual o sistema vai ser *treinado*, as cadeias de Markov e outros modelos semelhantes prestam-se especialmente bem a tarefas que têm a ver com a classificação e imitação de estilos pré-existentes.

Passando revista a estas implementações musicais, constata-se que elas operam no nível simbólico. Mas porque não utilizar as cadeias de Markov directamente ao nível do áudio digital? Miranda, Manzolli e Maia Jr. (2005) propõem um modelo em que as cadeias de Markov são combinadas com informação espectral organizada nos termos da síntese granular. Aqui. porém, refiro-me a uma técnica mais simples, que é a de partir da análise do áudio no domínio do tempo, e construir a tabela de transições contando o número de vezes em que um *sample* de valor x se sucede a cada *sample* de valor y.

Tal como sucede no domínio simbólico, também aqui as cadeias de Markov de primeira ou segunda ordem produzem resultados que se aproximam do aleatório e, nesse sentido, do ruído. Elevando-se a ordem do modelo, passam a considerar-se as probabilidades de um certo valor suceder a grupos mais alargados de valores antecedentes. Isto tem sentido até ao ponto em que o modelo mais não faz do que regurgitar blocos inteiros recortados do sinal original, ou

[8] The clear implication is that we can build machines which will create music. A set of tables could be constructed which would compose Mozartian melodies or themes which would out-Shostakovich Shostakovich. We could get as close as desired to the style of any type of music without actually copying the melodies, and by altering the probabilities, we might evolve whole new styles.

até o próprio sinal original inteiro, no caso extremo em que a tabela de transições acaba por só possuir um caminho possível determinístico—a cada grupo sucede sempre um único valor com probabilidade 1.

Com isto em conta, elaborei um programa que lê e analisa um ficheiro áudio[9] de entrada, constrói a tabela de transições para uma determinada ordem n, e cria um ficheiro novo gerado segundo o modelo que vimos tratando.

Uma das aplicações práticas consistiu na elaboração de uma parte *fixed media*, a combinar com duo de improvisadores, para a música do espectáculo de dança *Grau Zero* (2017) da coreógrafa e bailarina Elisabete Magalhães. Tema central era o corpo e a identidade. Gravei previamente a bailarina a ler um texto da sua autoria. Posteriormente, como era a materialidade do som que interessava, mais do que o conteúdo textual, escolhi uma secção com cerca de um minuto e meio e submeti-a ao processamento com cadeias de Markov. Gerei assim vários minutos de som utilizando valores de n entre 1 e 4. Os ficheiros correspondentes foram depois "lançados" em momentos específicos da *performance*.

Quando n é 1 o resultado apresenta-se como ruído, sem similitude com o sinal original. Já quando n é 4 identifica-se uma voz humana e a sua autora. De entre um murmúrio de fundo sobressaem, desconexos e entrecortados, sons sibilantes, ditongos, fonemas plosivos, partes de palavras. Como o que está a suceder é uma reorganização do sinal, estilhaçado em minúsculas células de fracção de segundo, o resultado acaba por fazer lembrar o que muitas vezes se obtém através da síntese granular.

As características e a própria duração do áudio original são absolutamente determinantes para o *output* produzido. Por outro lado, trata-se aqui de um processo cuja avaliação não se pretende objectiva, mas sim com base em critérios estéticos subjectivos do compositor. Em todo o caso será ainda necessária alguma experimentação suplementar, a fim de identificar possíveis caminhos de desenvolvimento ulterior deste sistema.

O CD que acompanha esta dissertação contém os referidos ficheiros *fixed media*, bem como uma gravação vídeo de uma primera versão do espectáculo, e ainda o código em Common Lisp. Este também está disponível *online*[10].

[9] O programa está pensado para áudio, mas em bom rigor funciona com qualquer tipo de ficheiro, dependendo o sucesso ou fracasso da operação do modo como a informação está nele codificada.

[10] https://github.com/ntrocado/markov-n

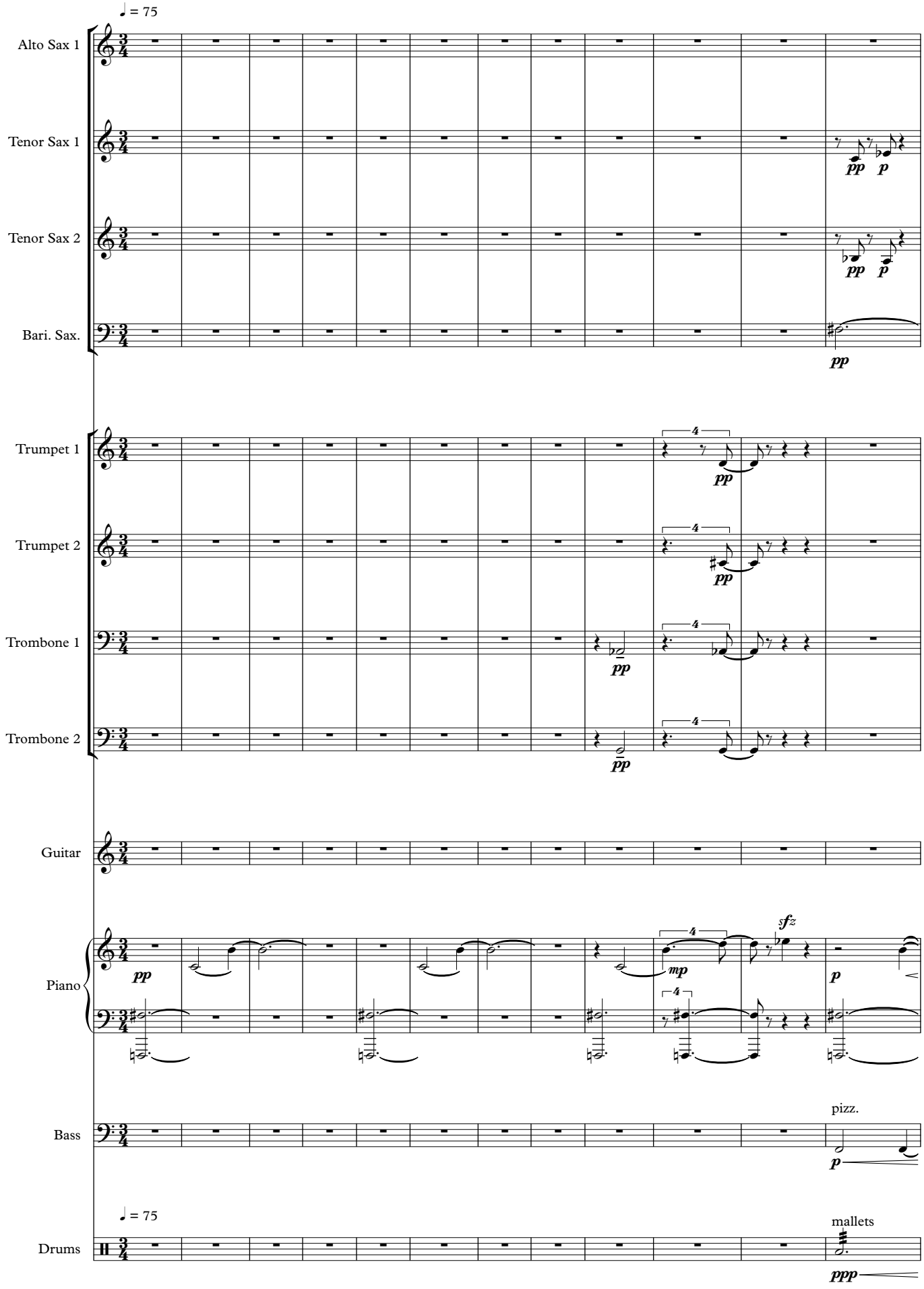
— 4 —

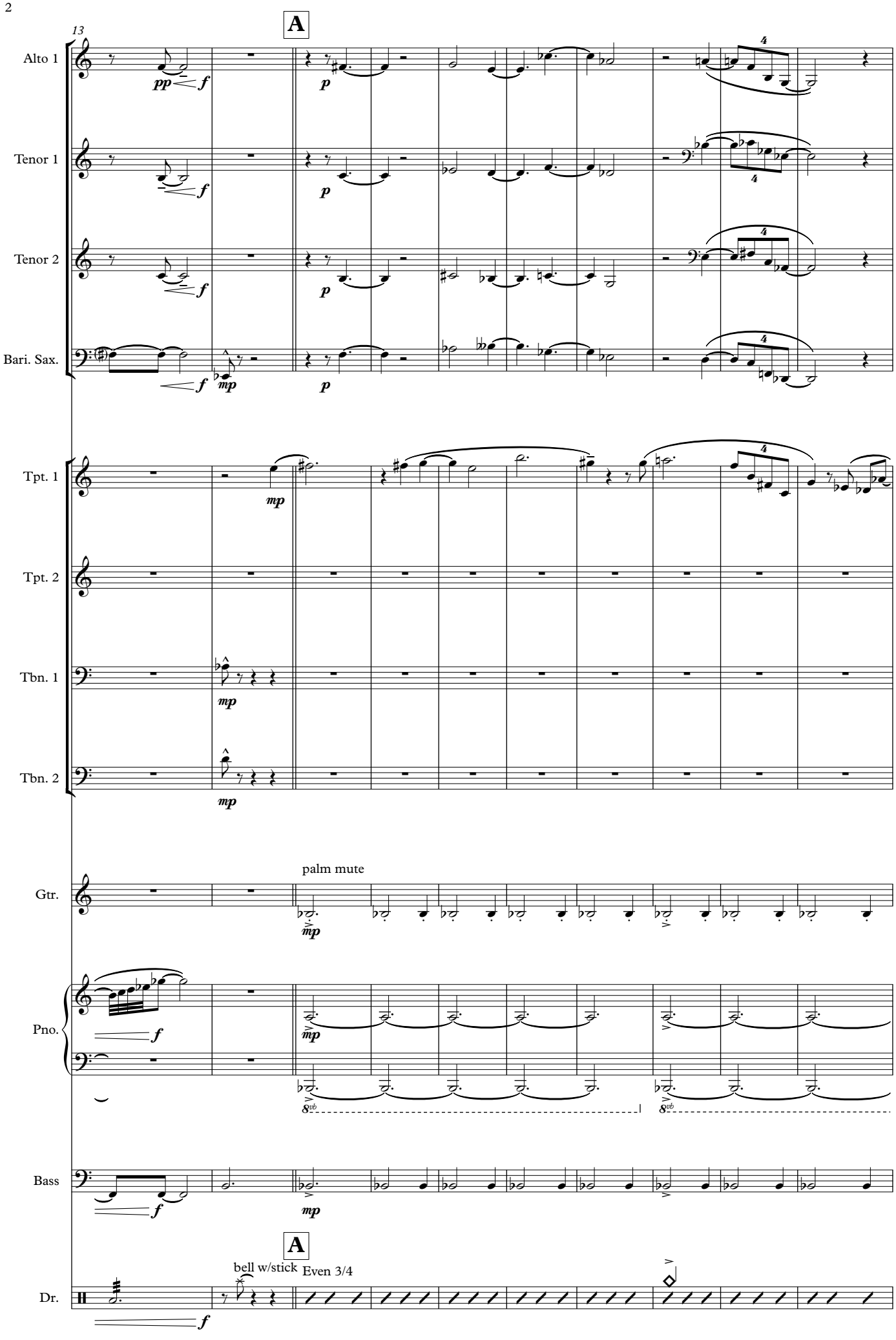
Partituras

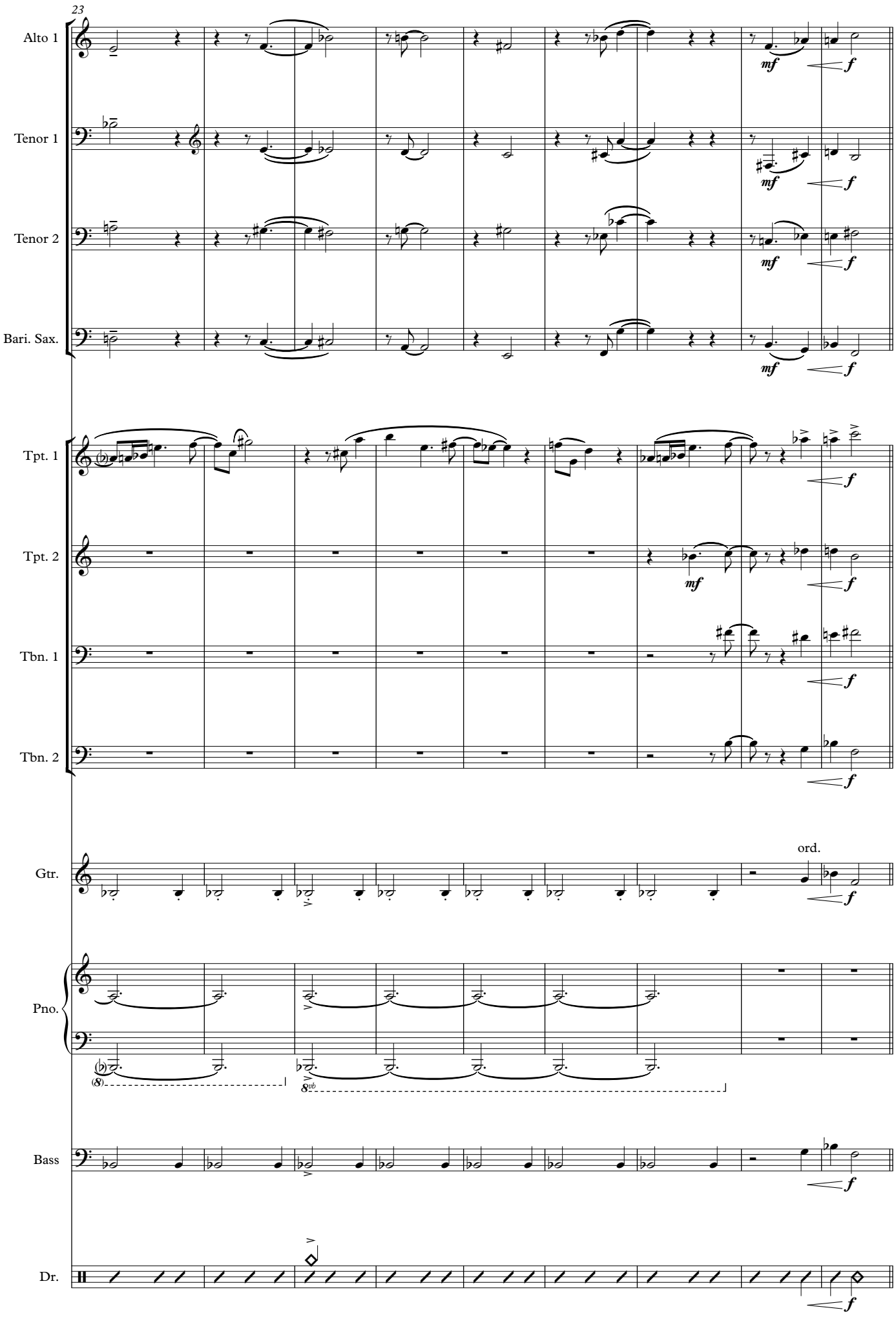
4.1 Um, para ensemble de Jazz

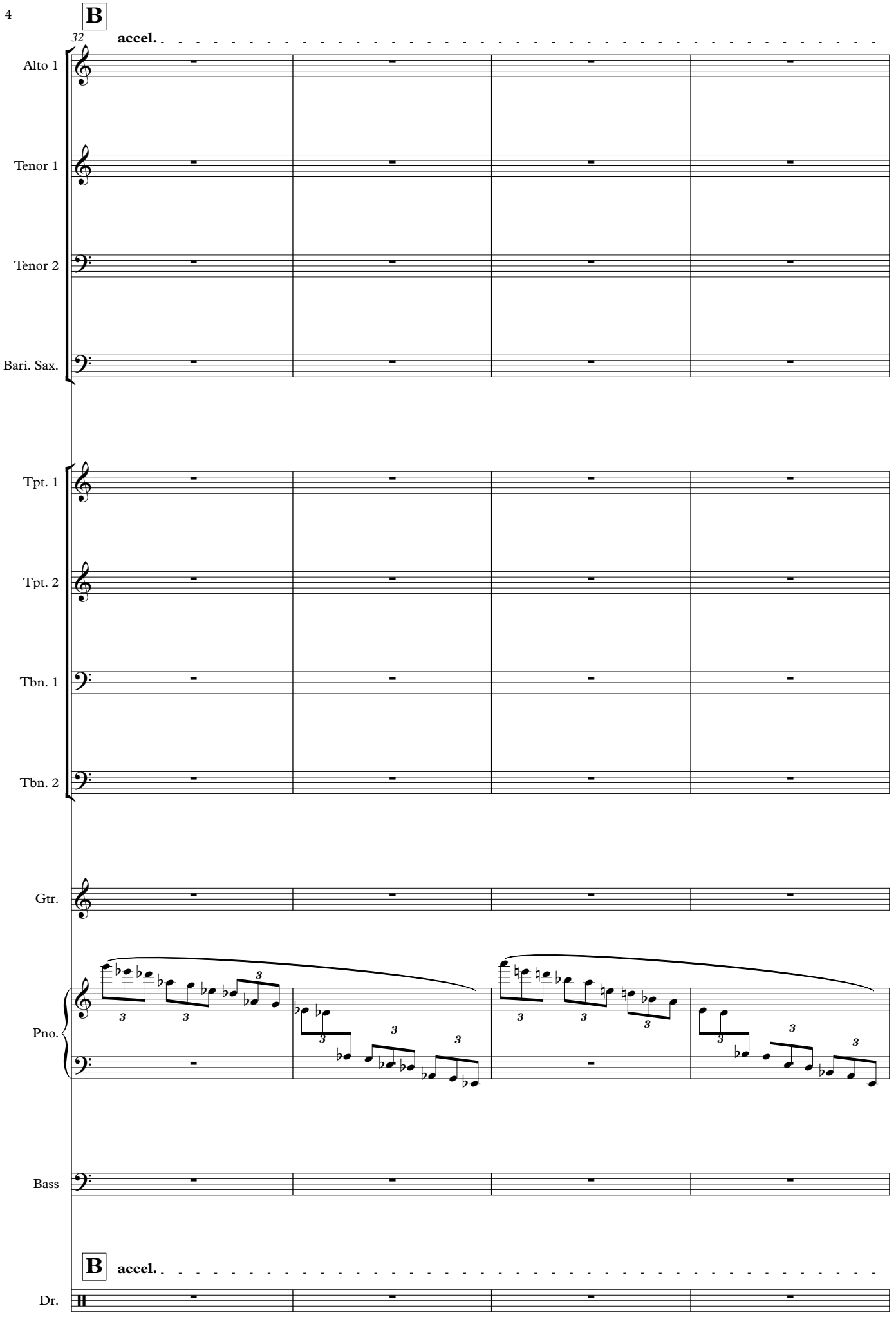
Um

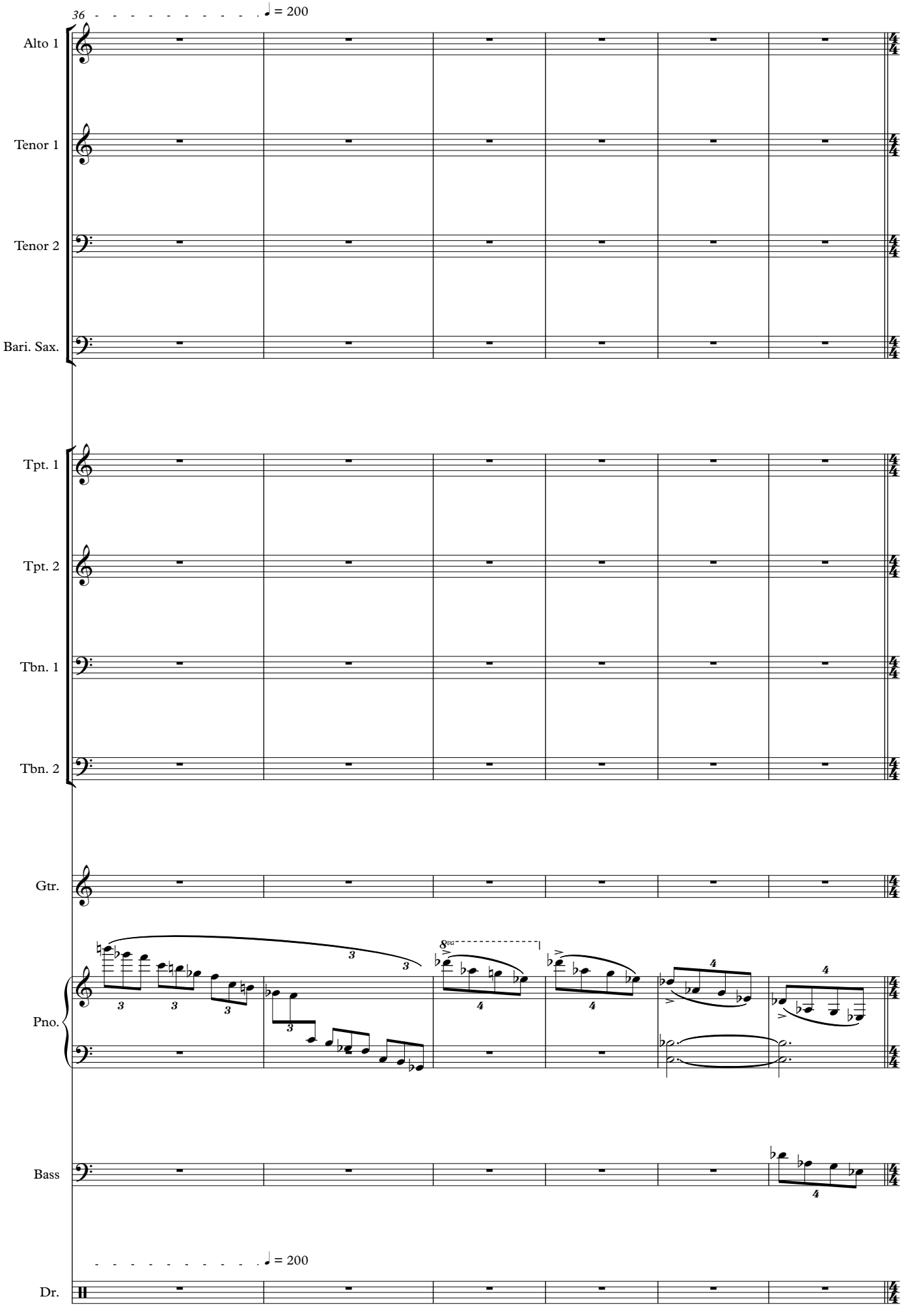
Nuno Trocado

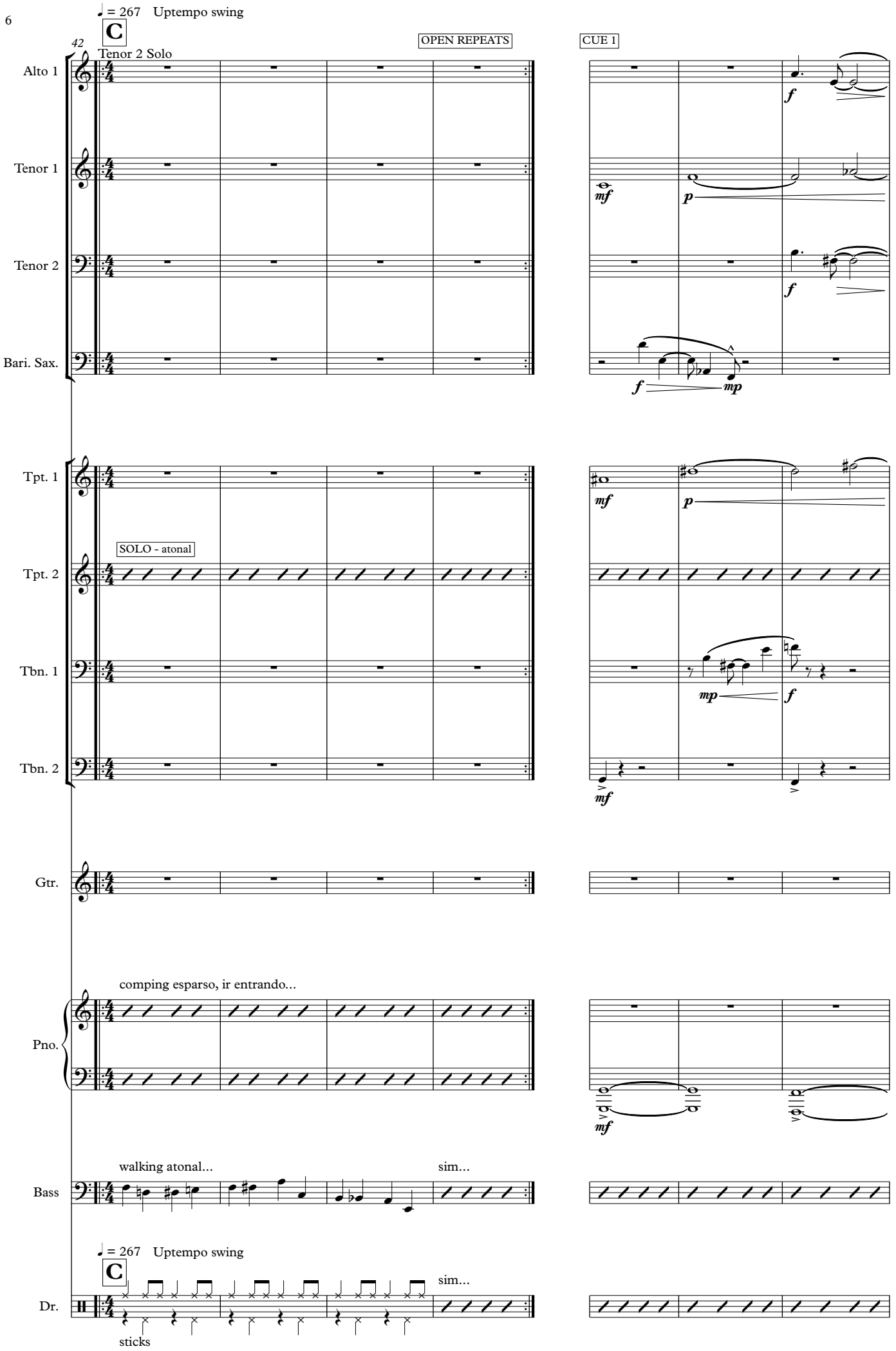


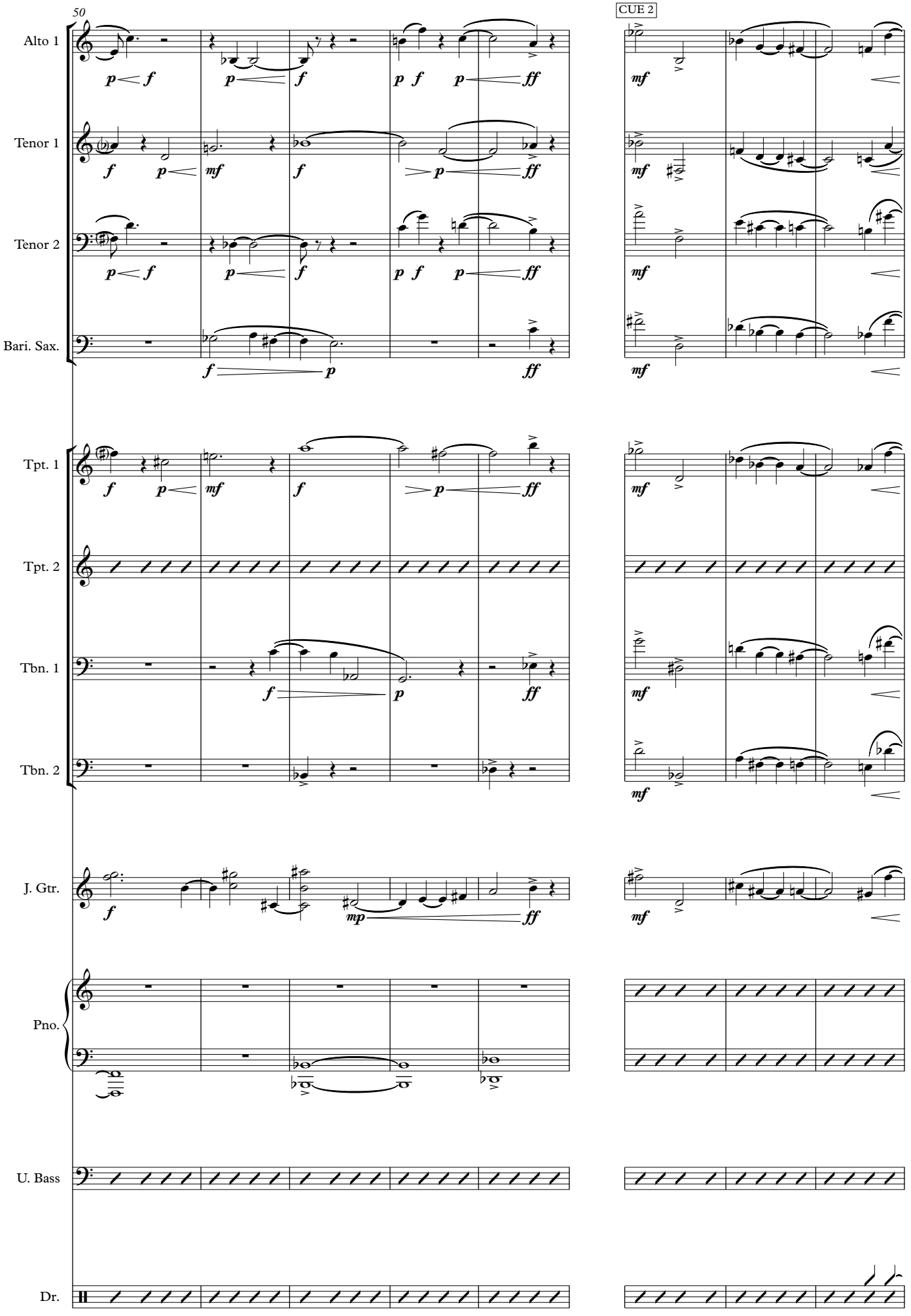

13
A
Alto 1
Tenor 1
Tenor 2
Bari. Sax.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
palm mute
Gtr.
Pno.
Bass
A
bell w/stick
Even 3/4
Dr.

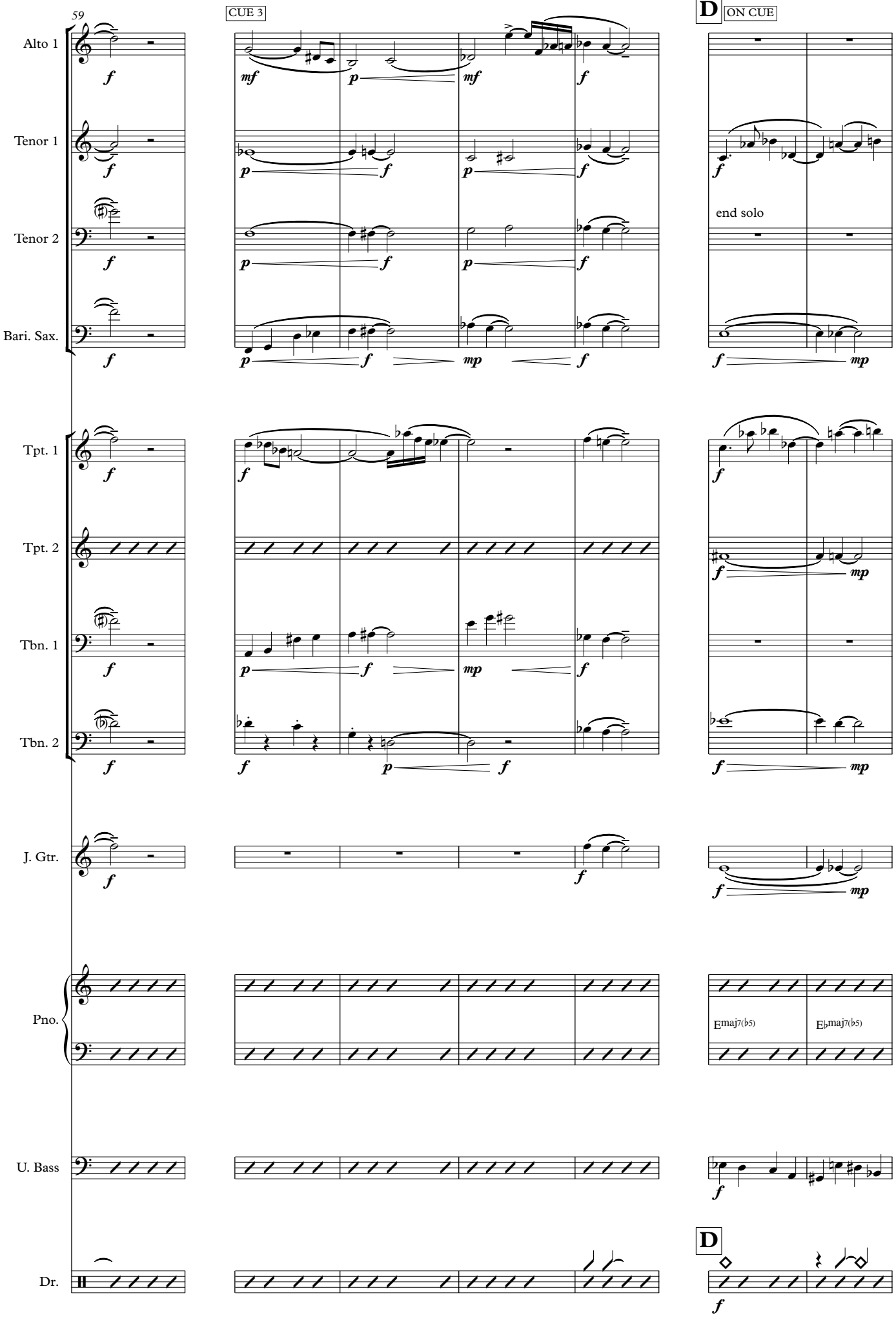

23
Alto 1
Tenor 1
Tenor 2
Bari. Sax.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
ord.
Pno.
Bass
Dr.
mf
f
8vb

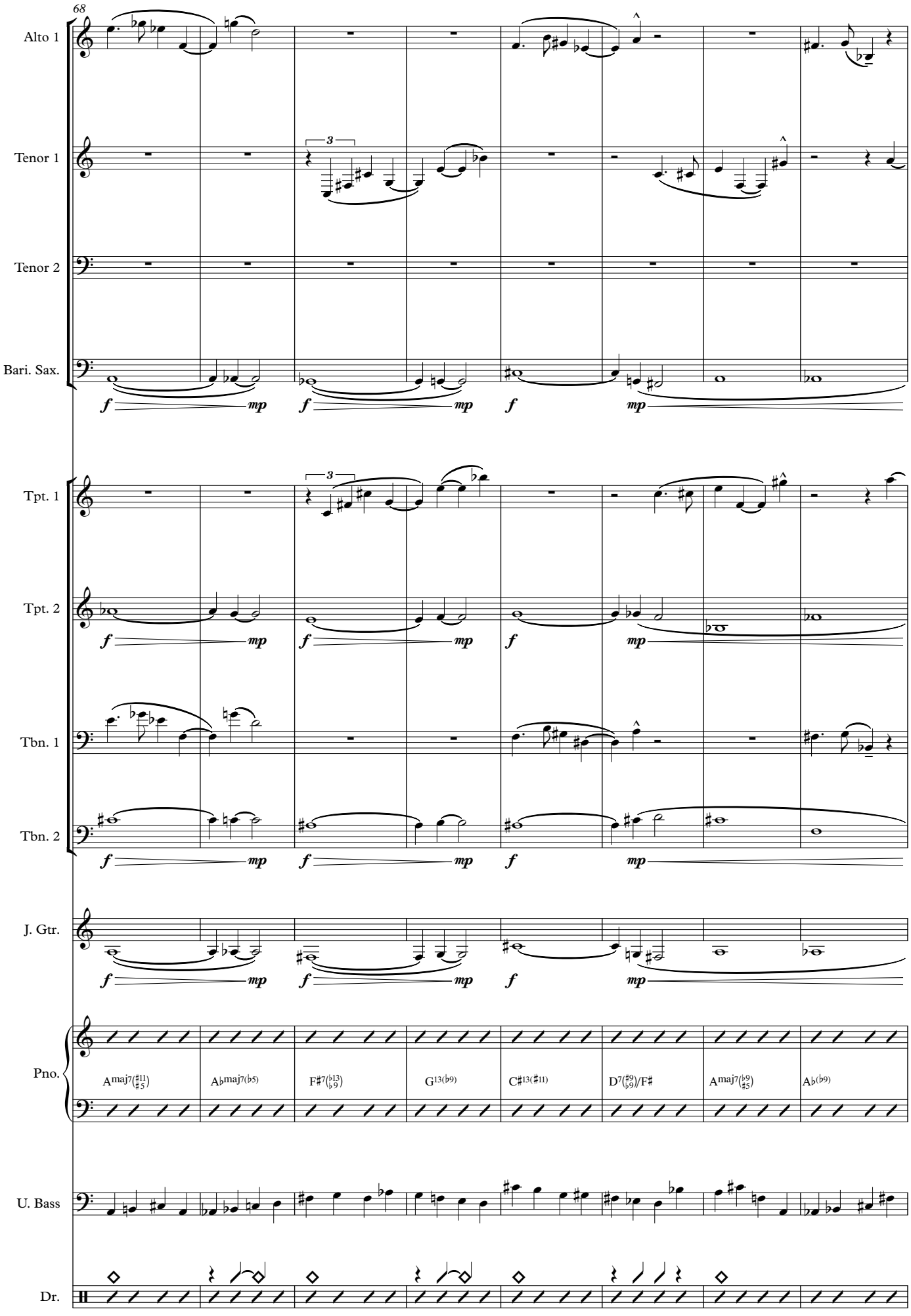

B
32
accel.
Alto 1
Tenor 1
Tenor 2
Bari. Sax.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bass
B
accel.
Dr.

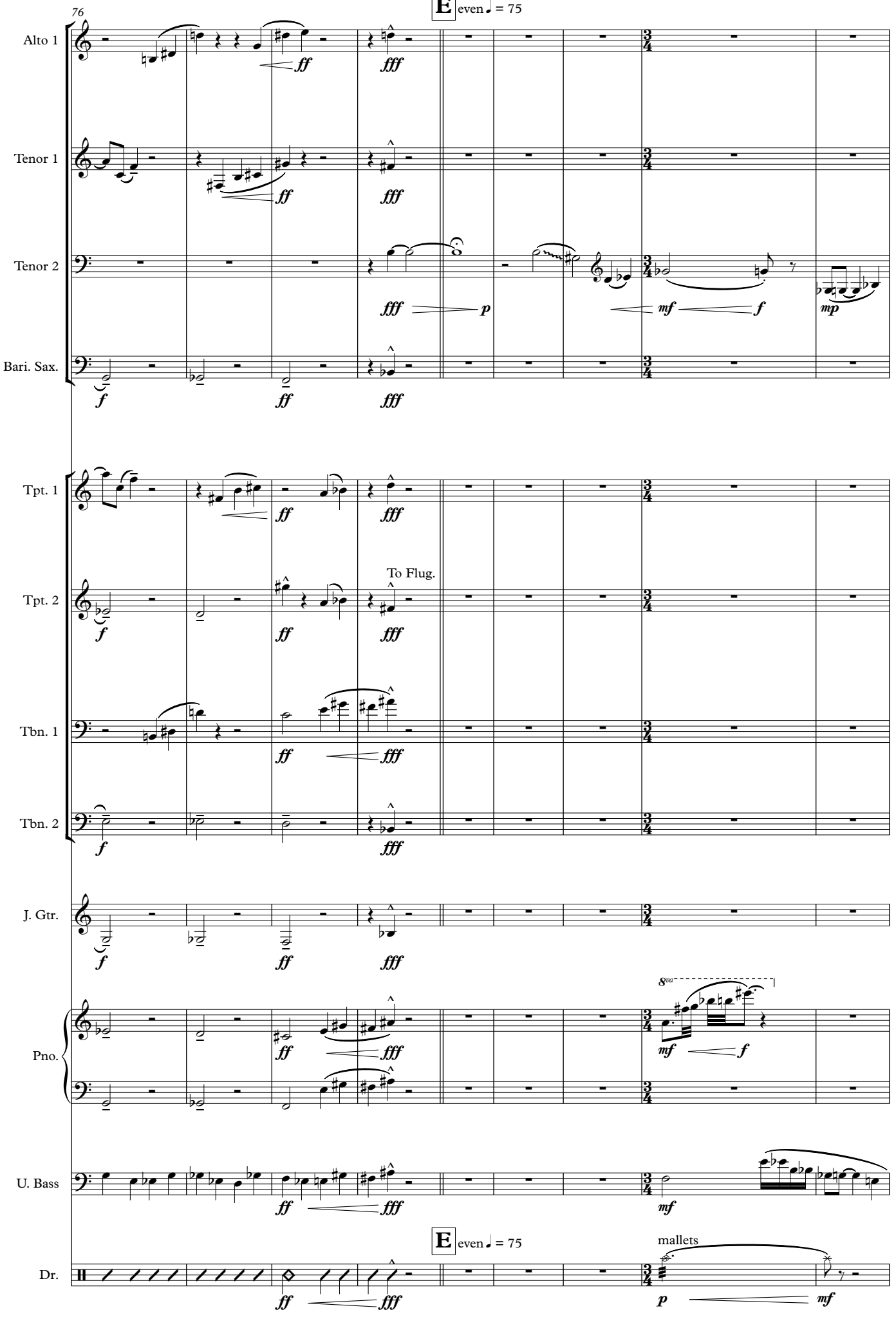

36
♩ = 200
Alto 1
Tenor 1
Tenor 2
Bari. Sax.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bass
♩ = 200
Dr.

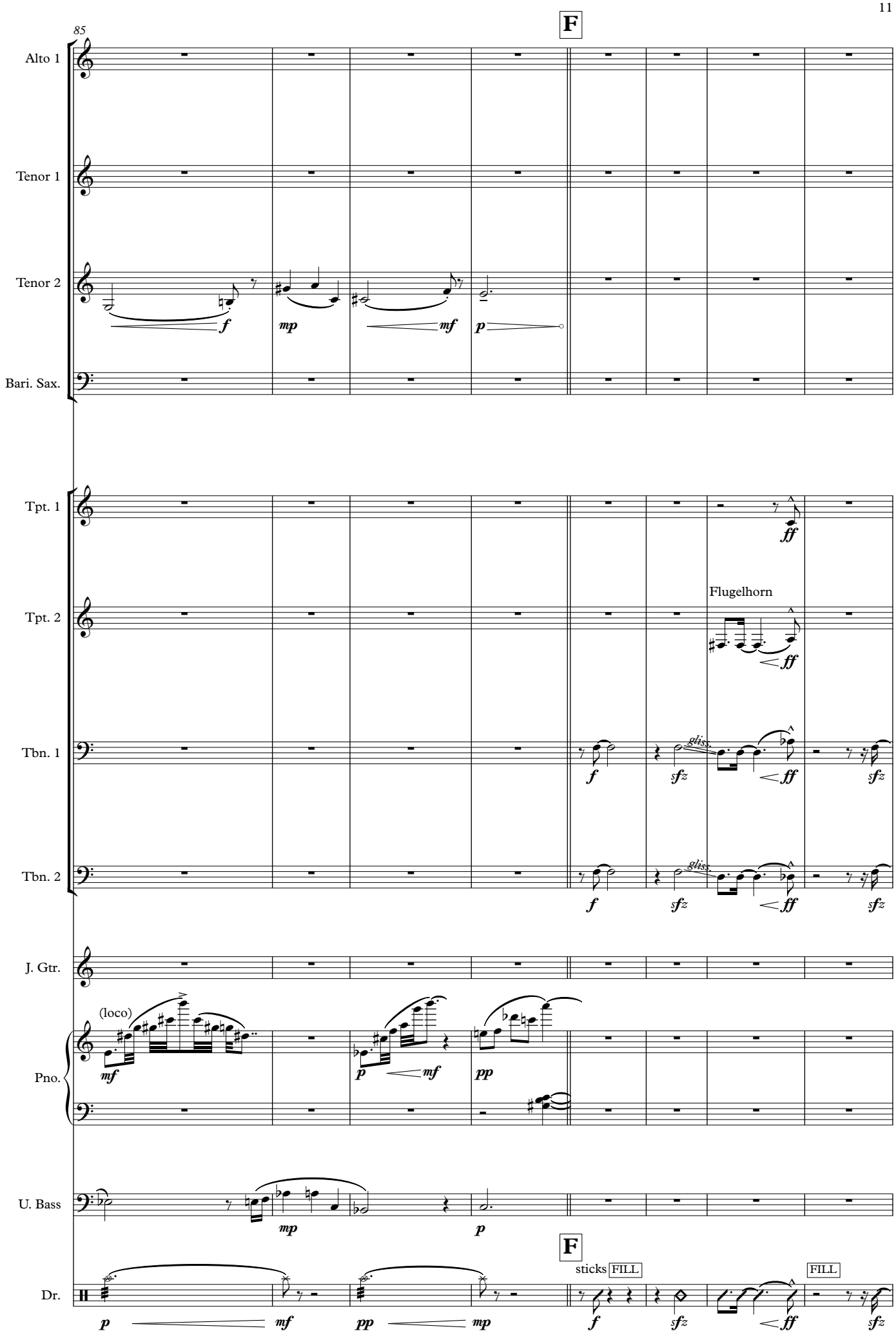

♩ = 267 Uptempo swing
C
42
Tenor 2 Solo
OPEN REPEATS
CUE 1
Alto 1
Tenor 1
Tenor 2
Bari. Sax.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
SOLO - atonal
Gtr.
Pno.
comping esparso, ir entrando...
Bass
walking atonal...
sim...
♩ = 267 Uptempo swing
C
Dr.
sim...
sticks

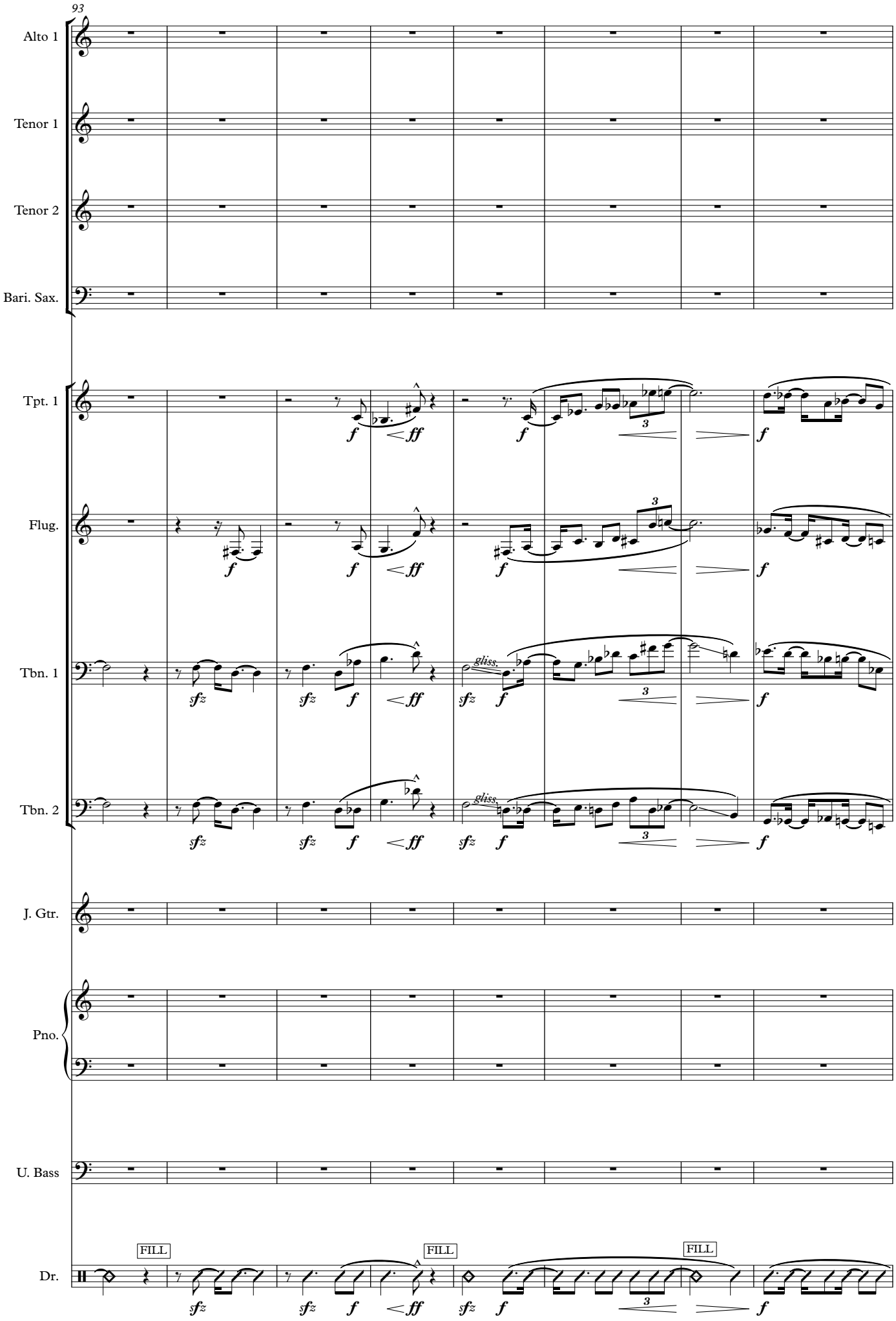

50
CUE 2
Alto 1
Tenor 1
Tenor 2
Bari. Sax.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
J. Gtr.
Pno.
U. Bass
Dr.

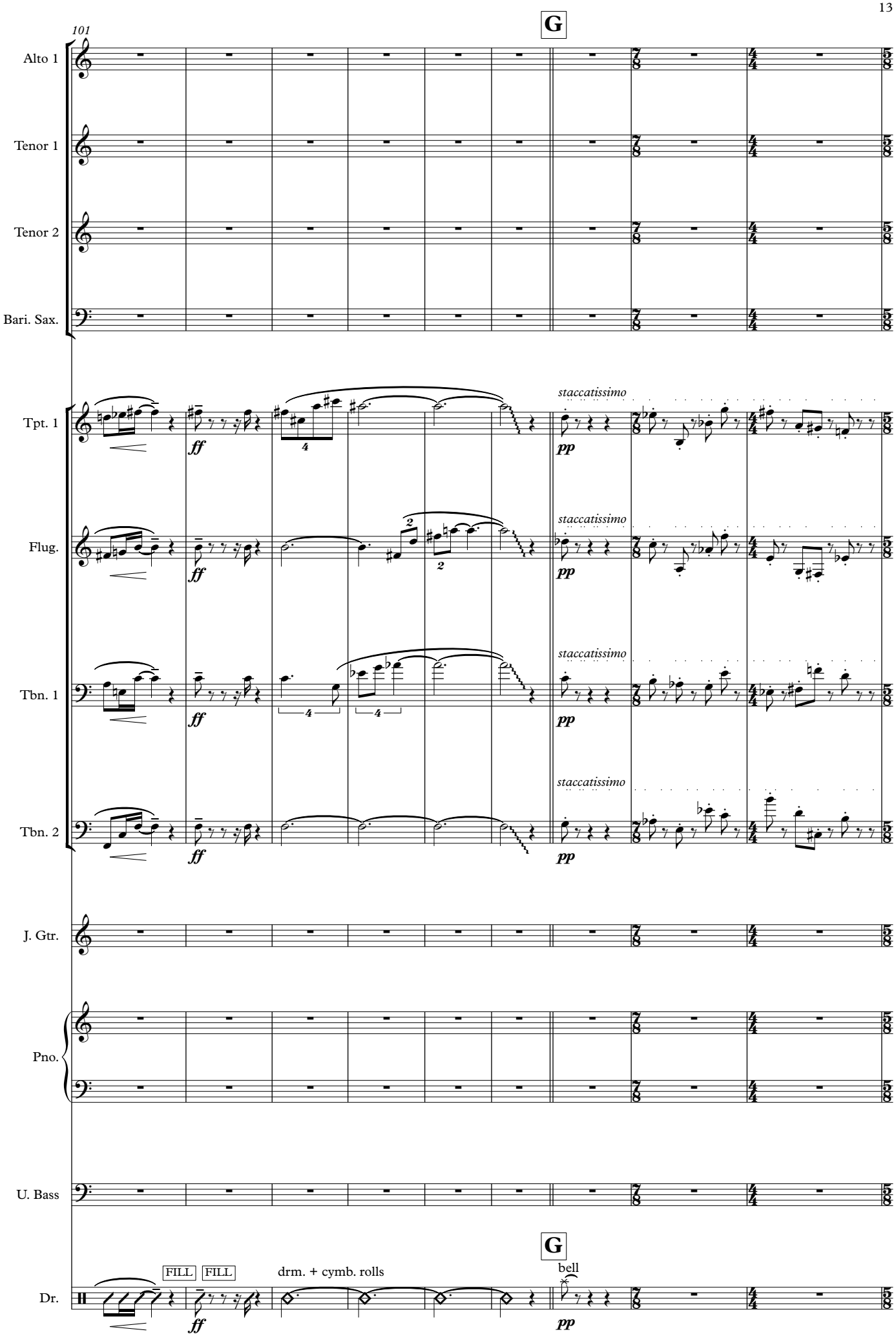

59
CUE 3
D
ON CUE
Alto 1
Tenor 1
Tenor 2
Bari. Sax.
end solo
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
J. Gtr.
Pno.
Emaj7(♭5)
E♭maj7(♭5)
U. Bass
Dr.
D

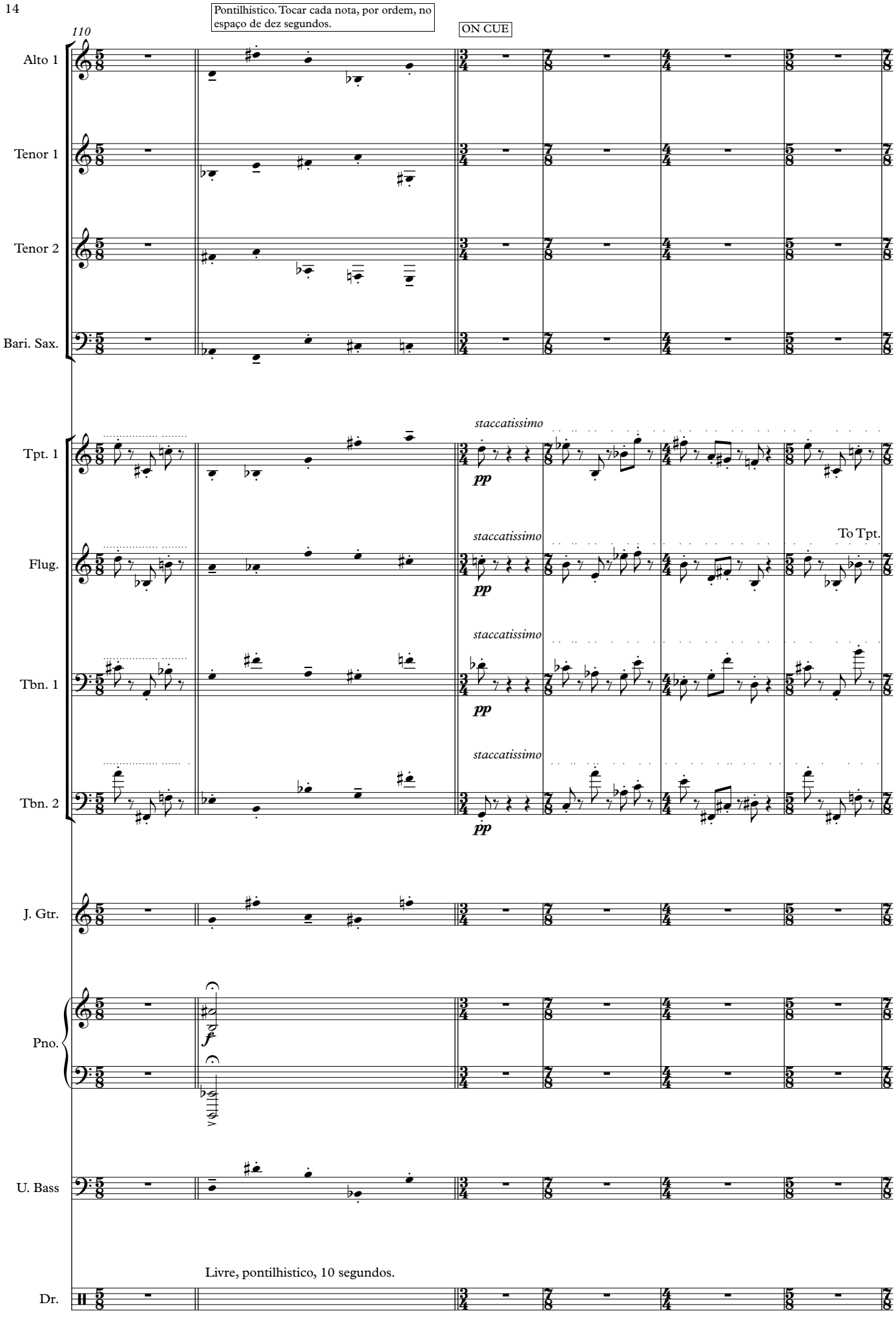

68
Alto 1
Tenor 1
Tenor 2
Bari. Sax.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
J. Gtr.
Pno.
U. Bass
Dr.
f
mp
3
A maj7(♯11 ♯5)
A♭ maj7(♭5)
F♯7(♭13 ♭9)
G13(♭9)
C♯13(♯11)
D7(♯9 ♭9)/F♯
A maj7(♭9 ♯5)
A♭(♭9)

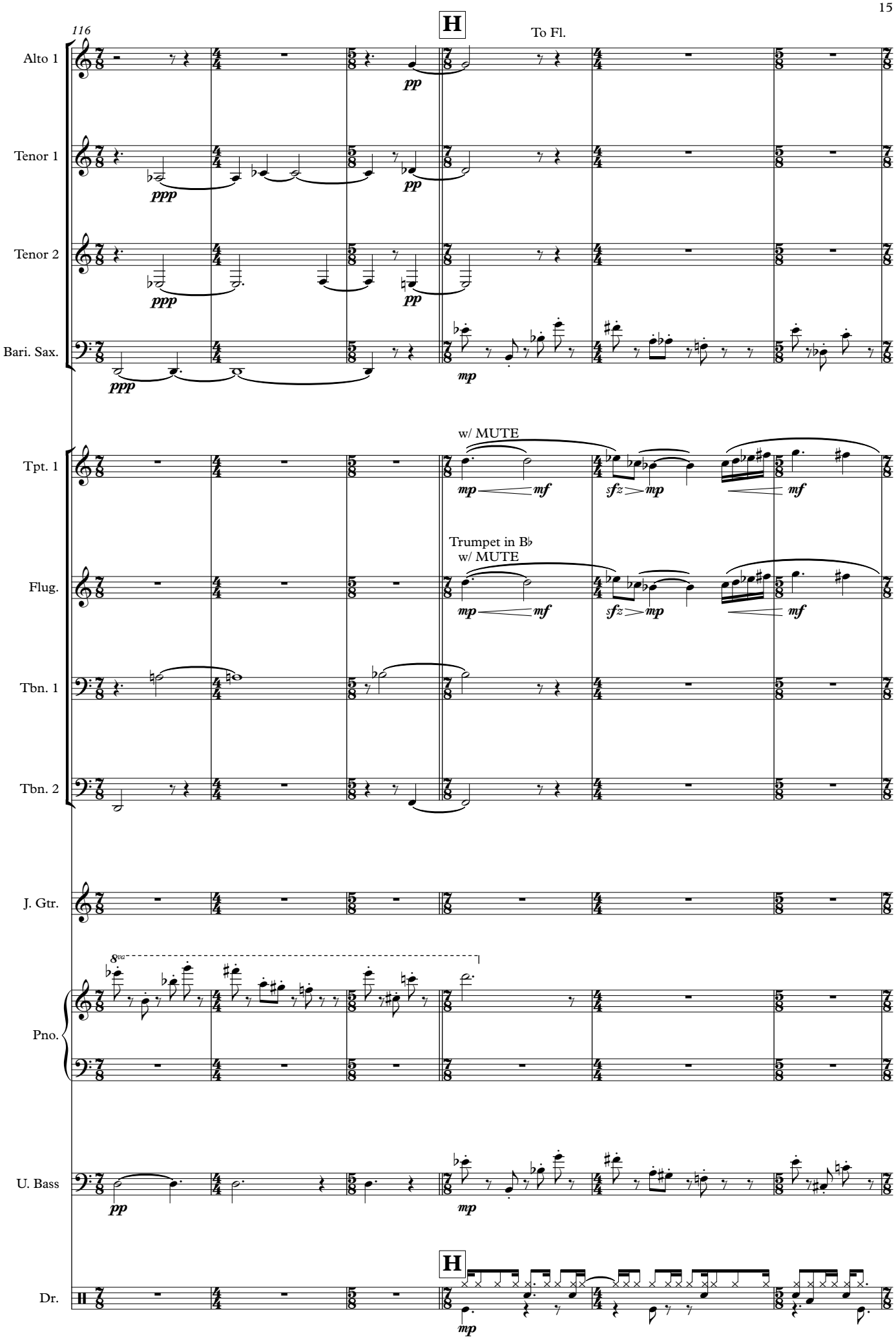

76
E even ♩ = 75
Alto 1
Tenor 1
Tenor 2
Bari. Sax.
Tpt. 1
Tpt. 2
To Flug.
Tbn. 1
Tbn. 2
J. Gtr.
Pno.
U. Bass
Dr.
mallets

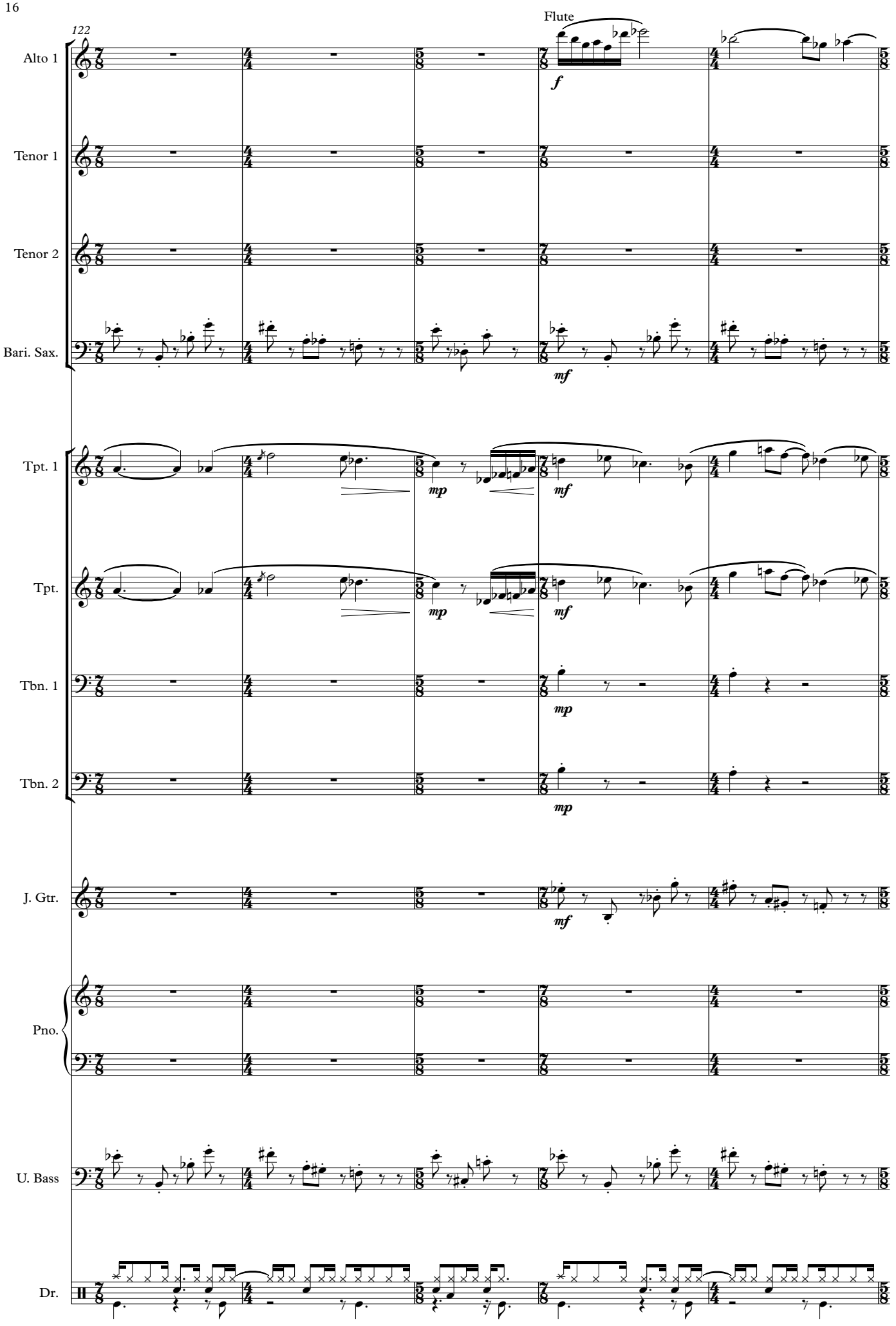

85
F
Alto 1
Tenor 1
Tenor 2
Bari. Sax.
Tpt. 1
Tpt. 2
Flugelhorn
Tbn. 1
Tbn. 2
J. Gtr.
(loco)
Pno.
U. Bass
F
sticks
FILL
FILL
Dr.

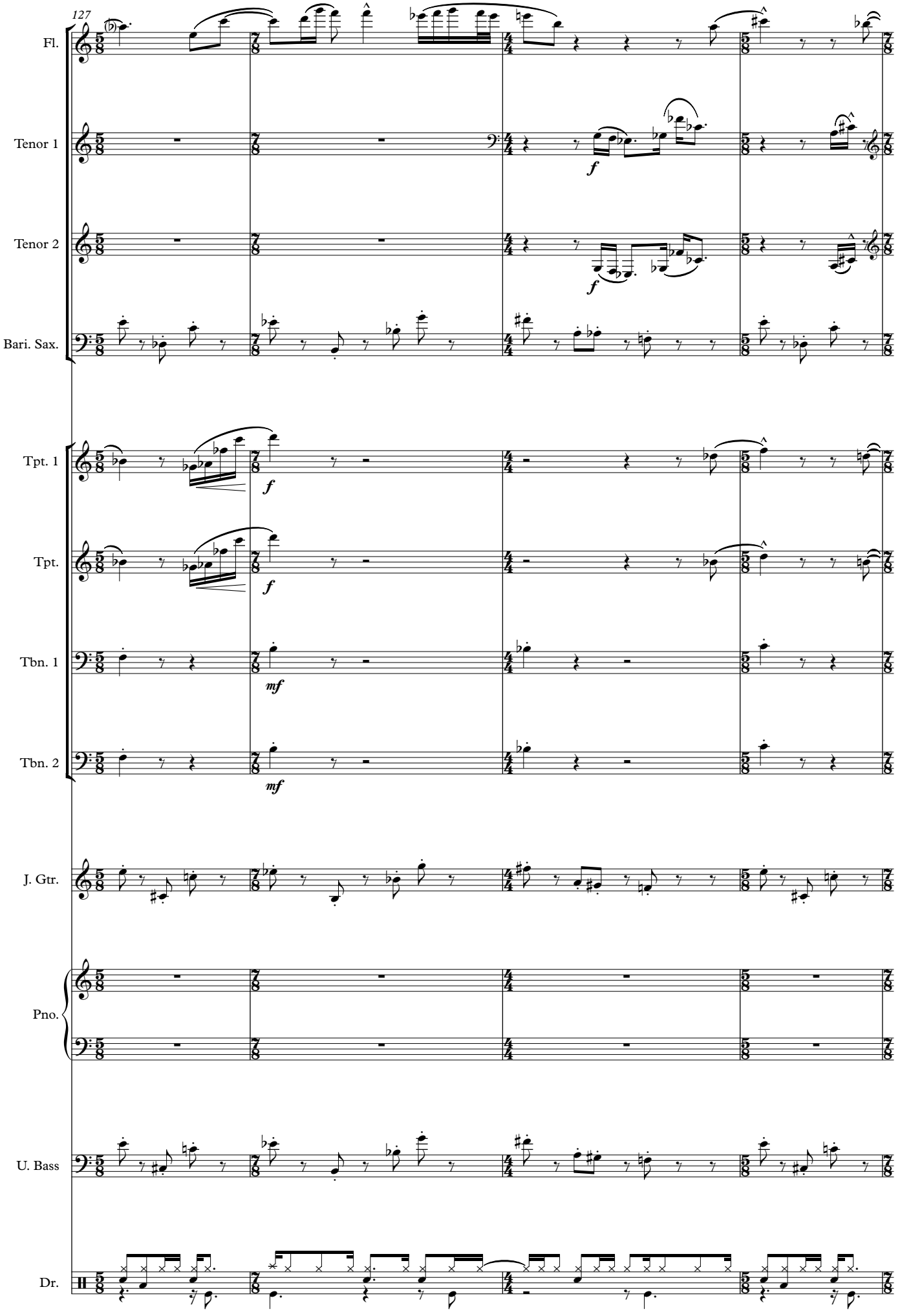

93
Alto 1
Tenor 1
Tenor 2
Bari. Sax.
Tpt. 1
Flug.
Tbn. 1
Tbn. 2
J. Gtr.
Pno.
U. Bass
Dr.
FILL
FILL
FILL
gliss.
gliss.

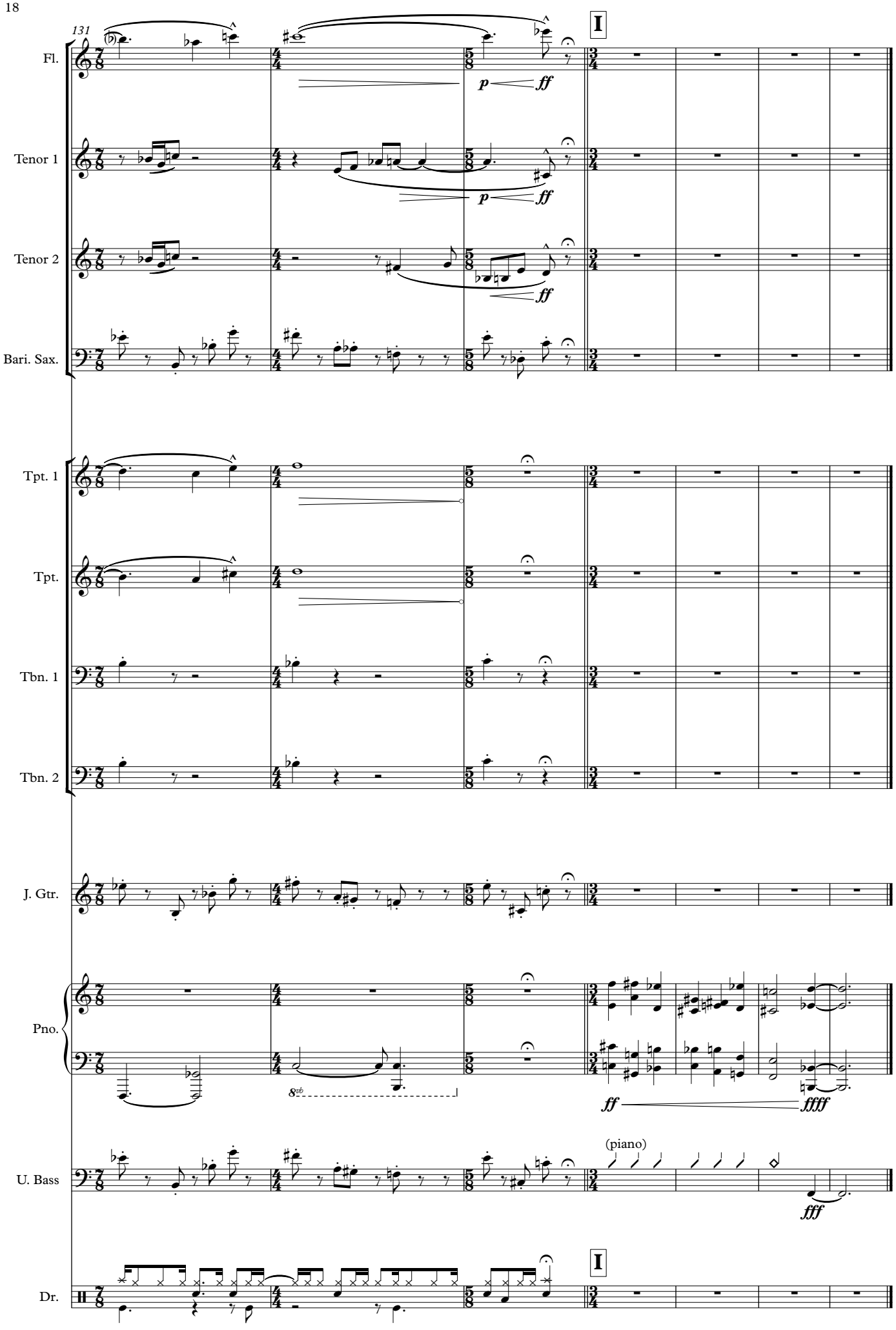

101
G
Alto 1
Tenor 1
Tenor 2
Bari. Sax.
Tpt. 1
Flug.
Tbn. 1
Tbn. 2
J. Gtr.
Pno.
U. Bass
Dr.
staccatissimo
ff
pp
FILL
FILL
drm. + cymb. rolls
bell


110
Pontilhístico. Tocar cada nota, por ordem, no espaço de dez segundos.
ON CUE
Alto 1
Tenor 1
Tenor 2
Bari. Sax.
staccatissimo
pp
Tpt. 1
Flug.
To Tpt.
Tbn. 1
Tbn. 2
J. Gtr.
Pno.
f
U. Bass
Livre, pontilhistico, 10 segundos.
Dr.


116
H
To Fl.
Alto 1
Tenor 1
Tenor 2
Bari. Sax.
w/ MUTE
Tpt. 1
Trumpet in B♭
w/ MUTE
Flug.
Tbn. 1
Tbn. 2
J. Gtr.
8va
Pno.
U. Bass
H
Dr.


122
Flute
Alto 1
Tenor 1
Tenor 2
Bari. Sax.
Tpt. 1
Tpt.
Tbn. 1
Tbn. 2
J. Gtr.
Pno.
U. Bass
Dr.
f
mf
mp


127
Fl.
Tenor 1
Tenor 2
Bari. Sax.
Tpt. 1
Tpt.
Tbn. 1
Tbn. 2
J. Gtr.
Pno.
U. Bass
Dr.
f
mf

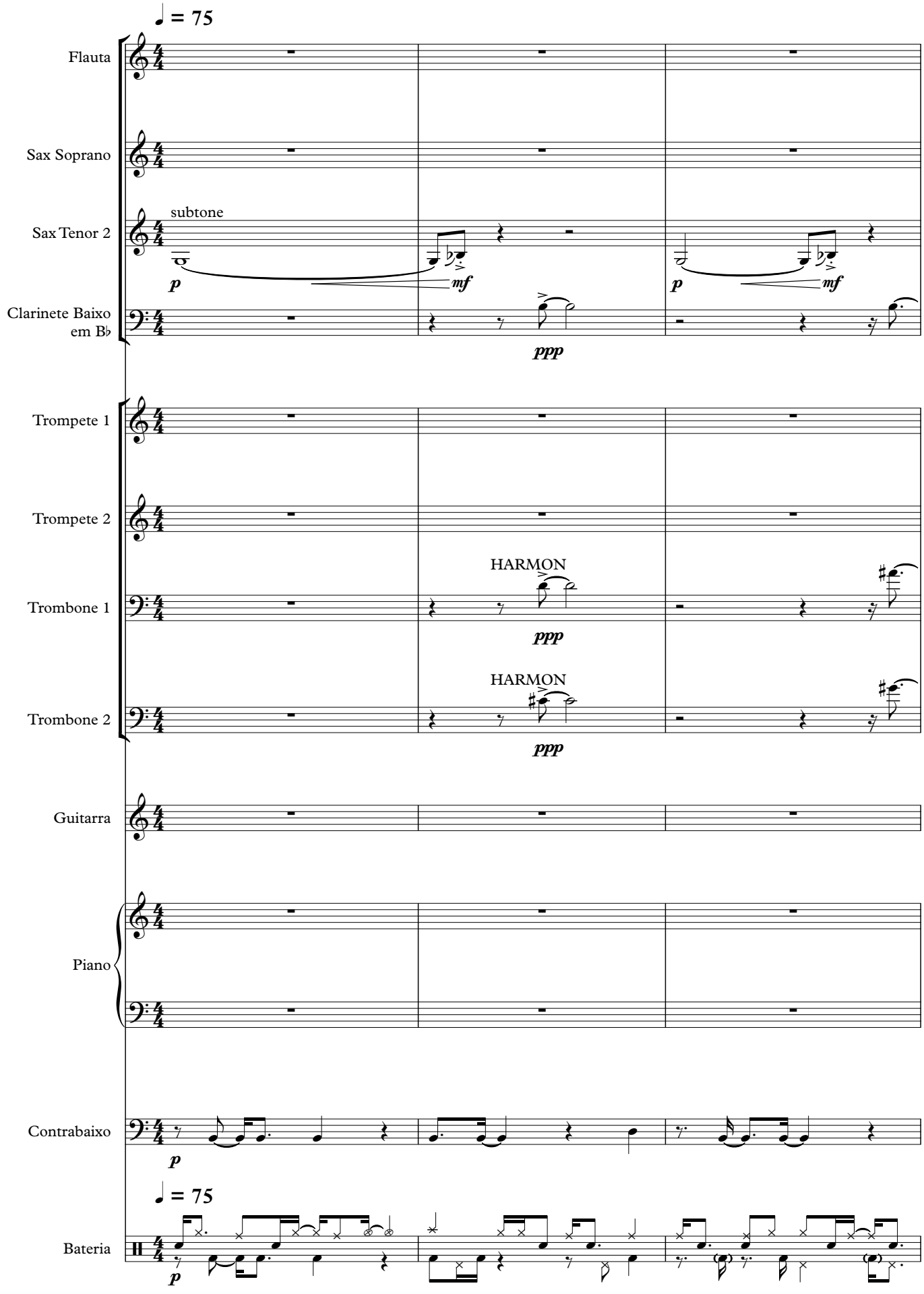

131
I
Fl.
Tenor 1
Tenor 2
Bari. Sax.
Tpt. 1
Tpt.
Tbn. 1
Tbn. 2
J. Gtr.
Pno.
U. Bass
Dr.
p
ff
8vb
(piano)
fff
ffff

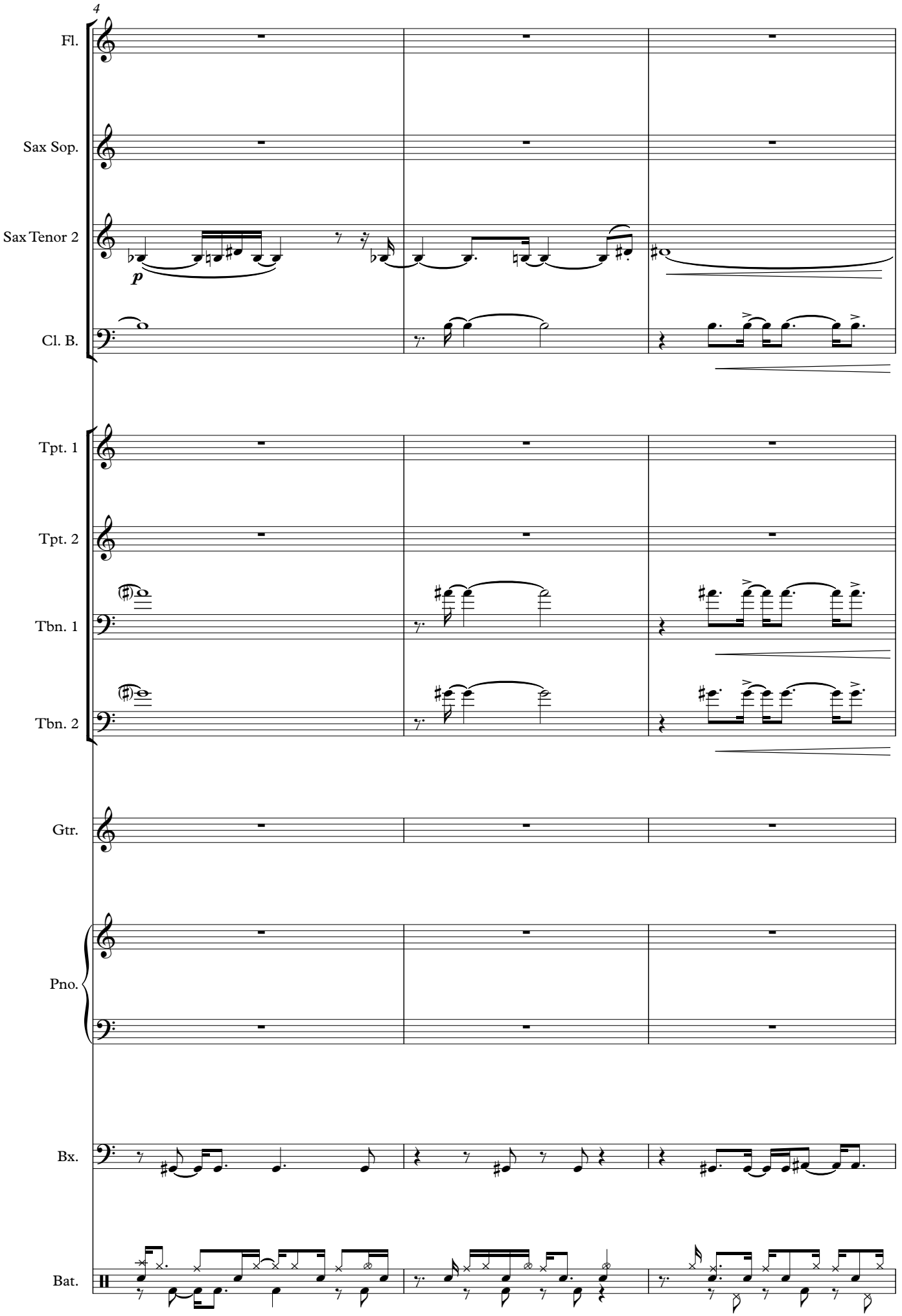
4.2 Dois, para ensemble de Jazz

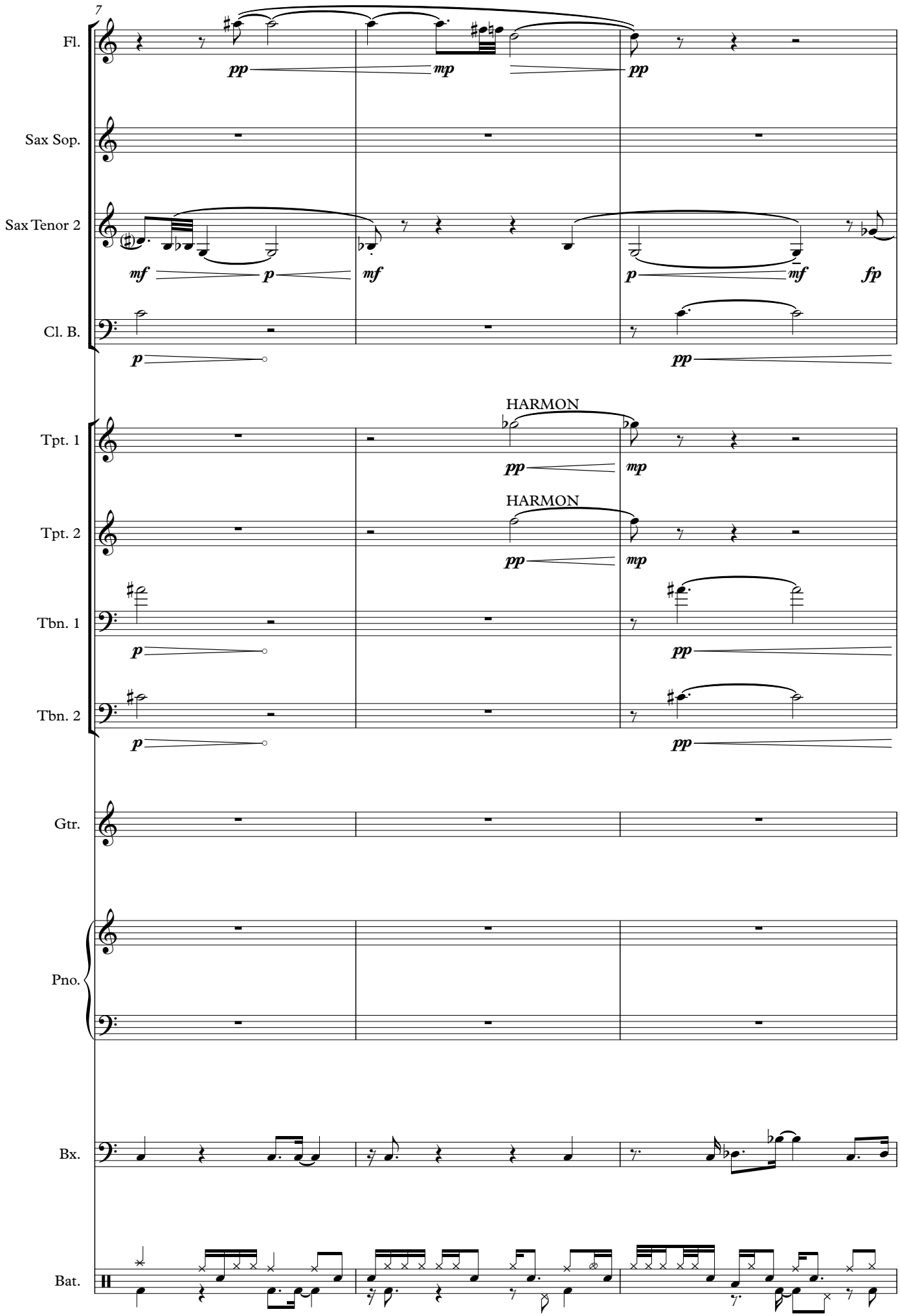
Partitura em C

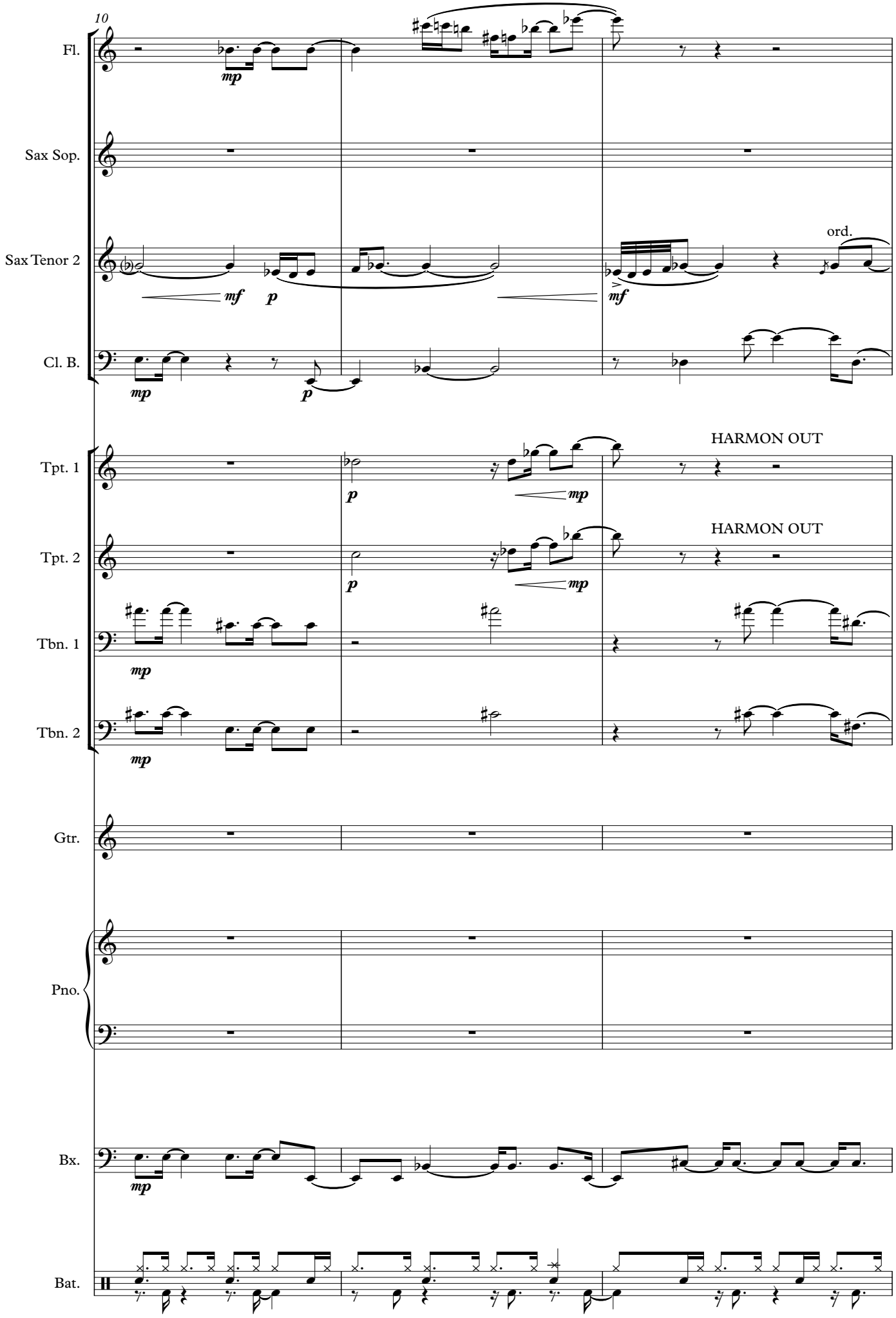
Dois

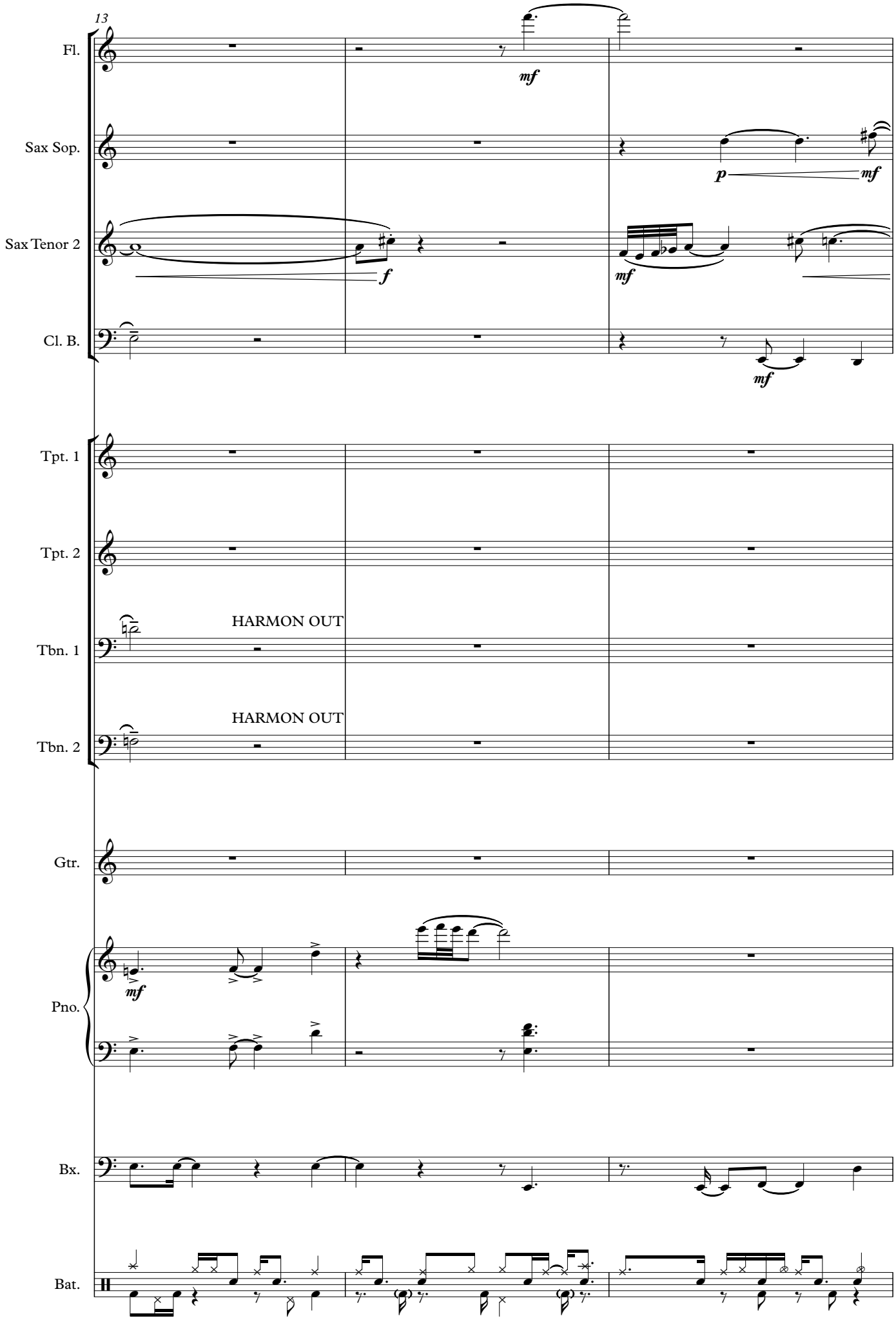
Nuno Trocado

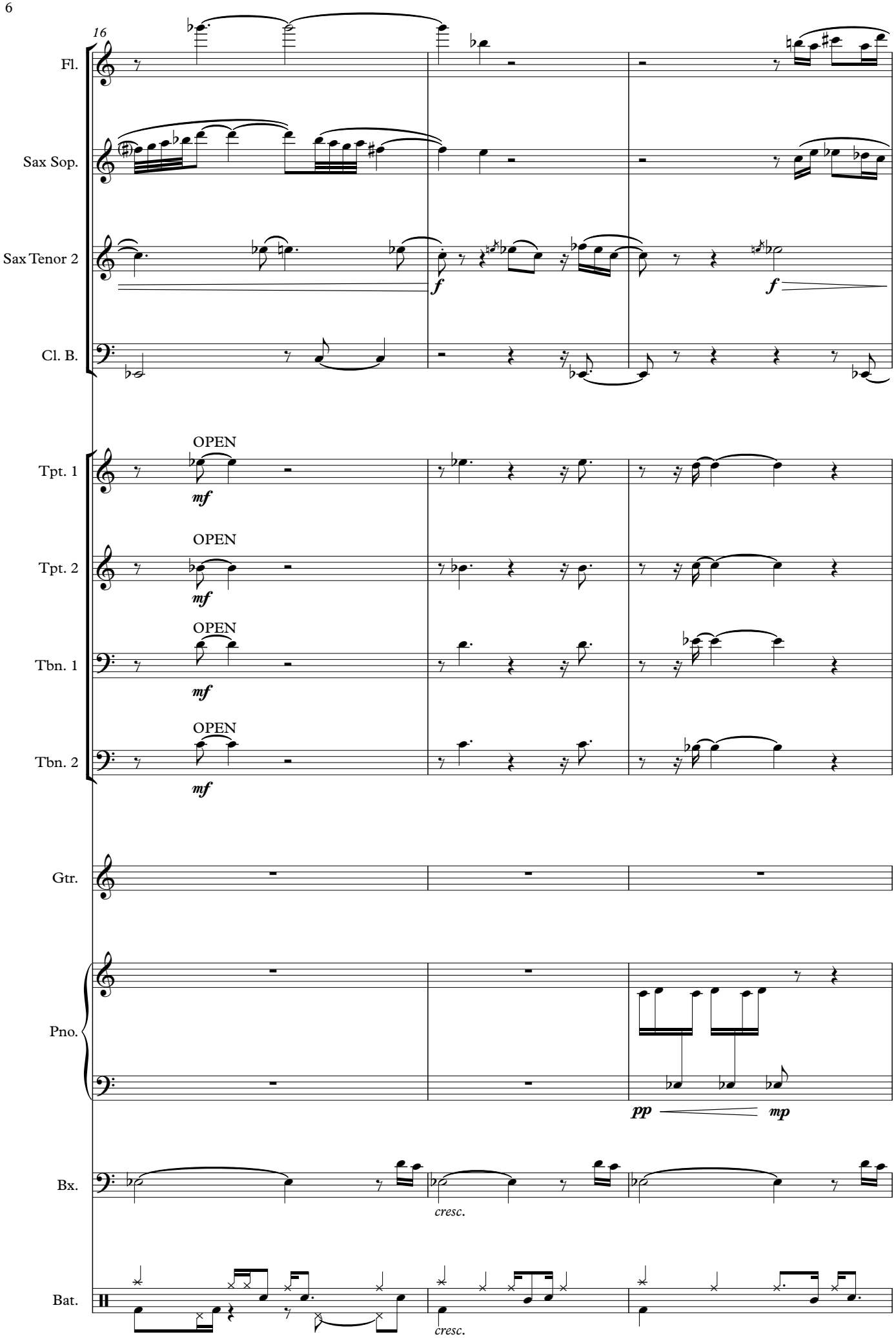


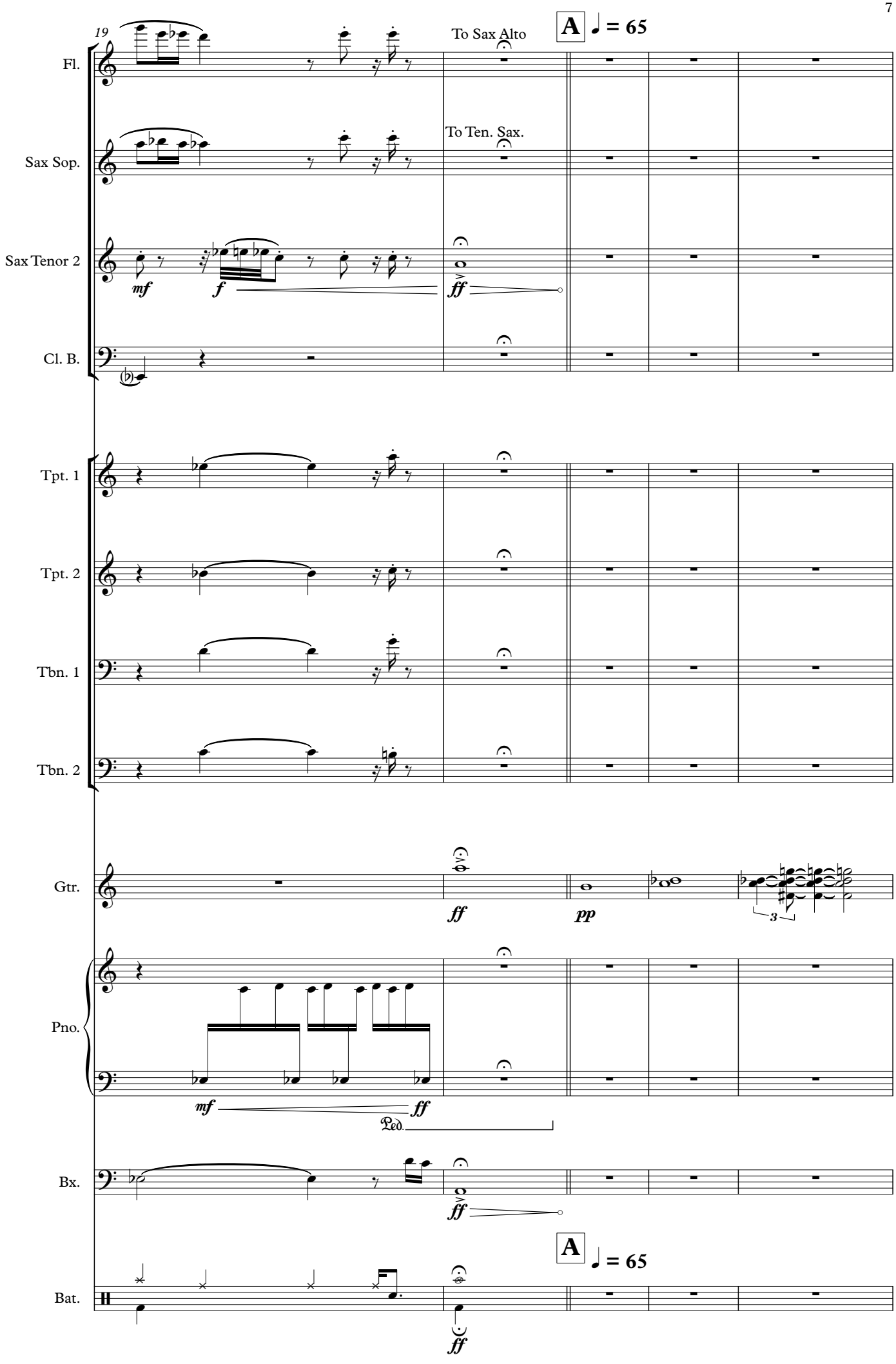

4
Fl.
Sax Sop.
Sax Tenor 2
p
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.

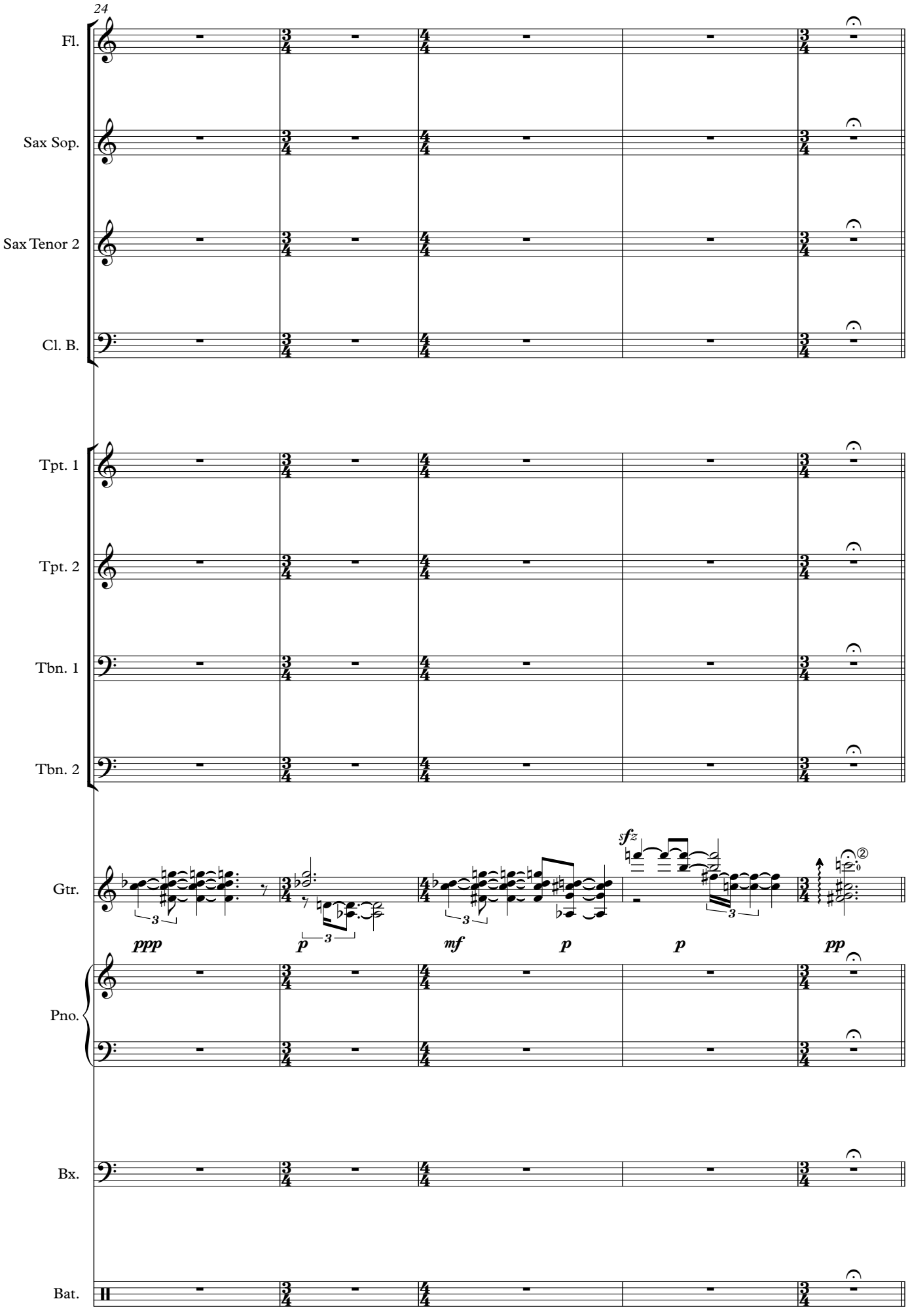

7
Fl.
Sax Sop.
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
HARMON
HARMON
pp
mp
mf
p
fp

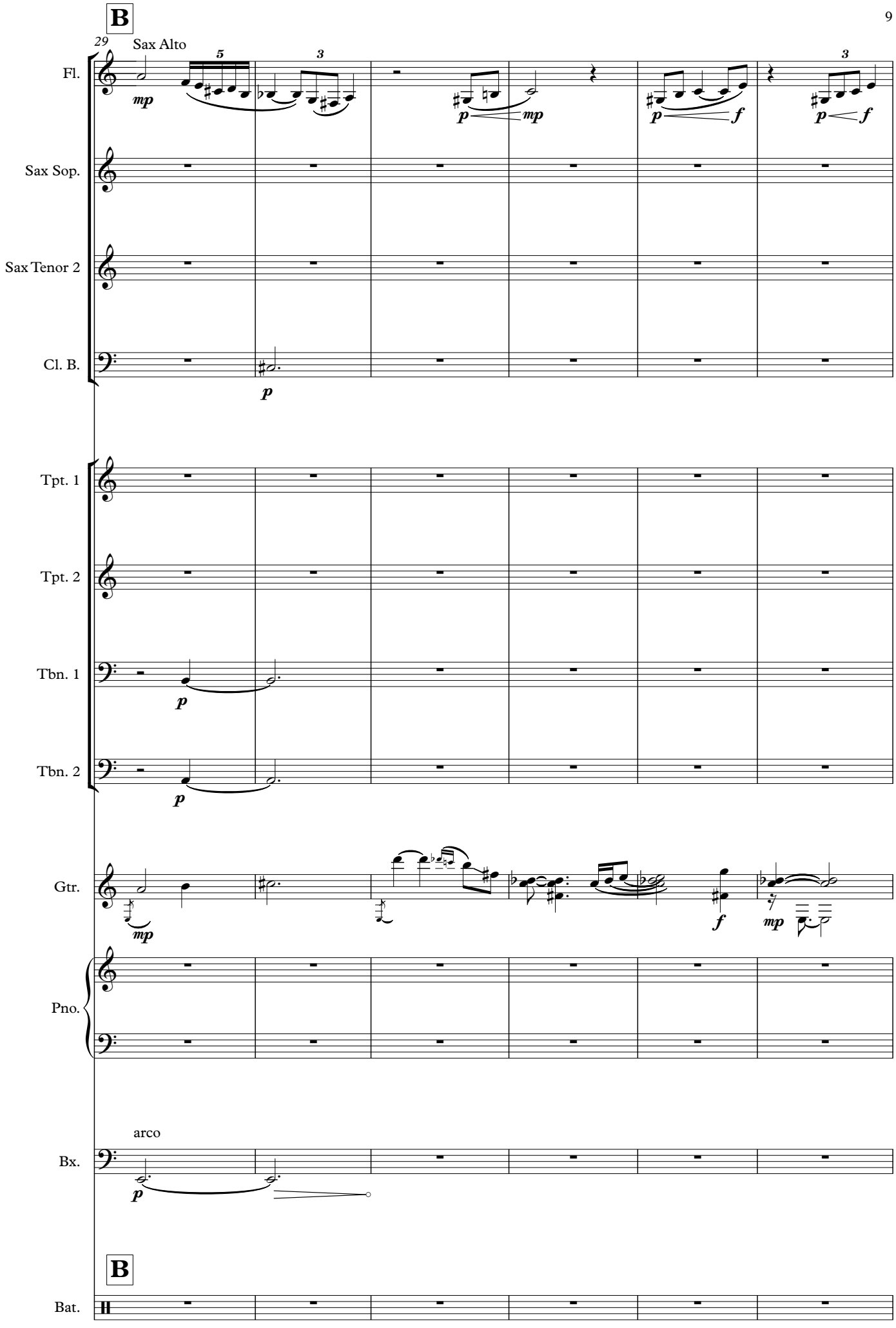

10
Fl.
Sax Sop.
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
mp
mf
p
ord.
HARMON OUT
HARMON OUT

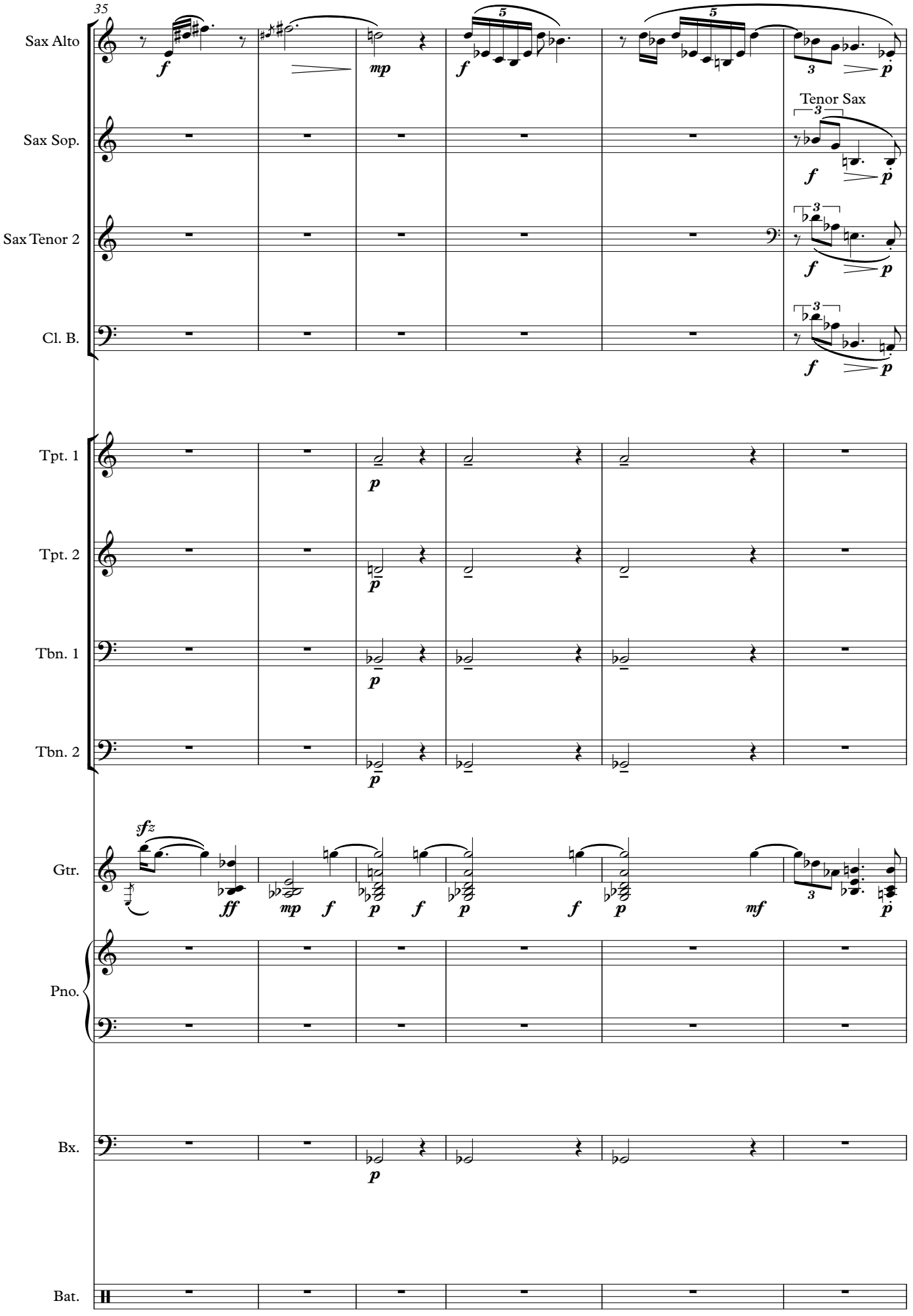

13
Fl.
mf
Sax Sop.
p
mf
Sax Tenor 2
f
mf
Cl. B.
mf
Tpt. 1
Tpt. 2
HARMON OUT
Tbn. 1
HARMON OUT
Tbn. 2
Gtr.
Pno.
mf
Bx.
Bat.

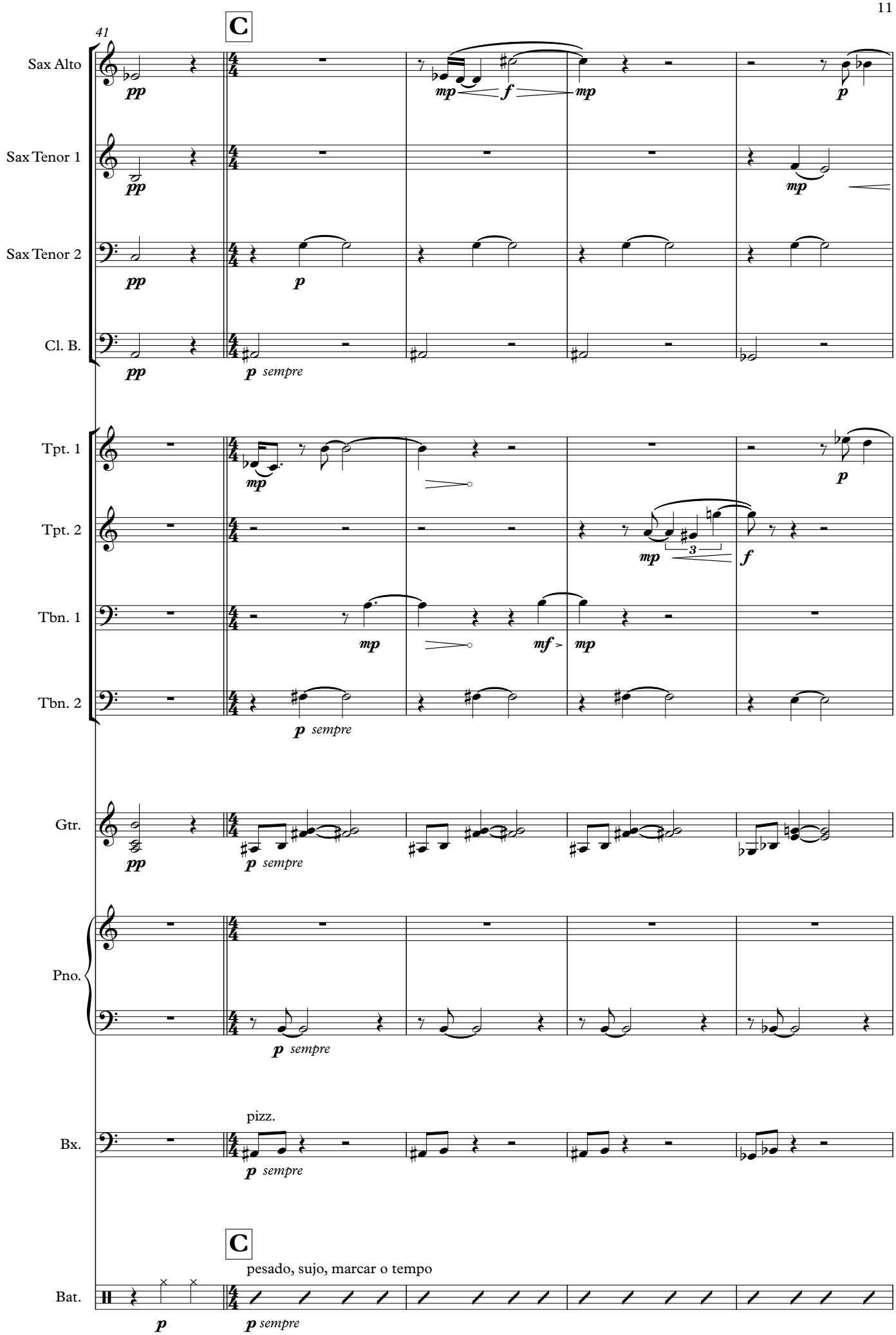

16
Fl.
Sax Sop.
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
OPEN
mf
f
pp
mp
cresc.

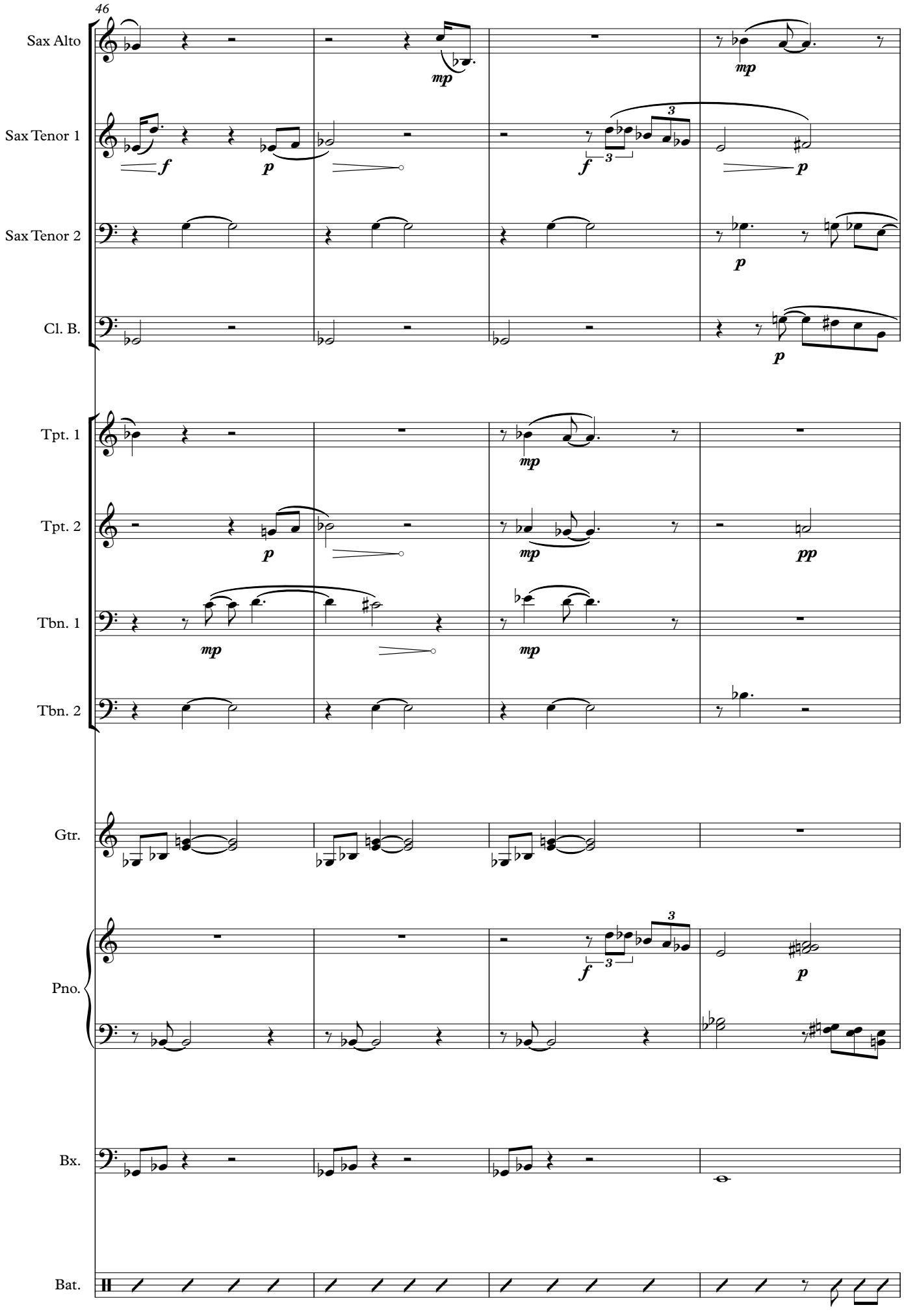

19
To Sax Alto
A
♩ = 65
Fl.
To Ten. Sax.
Sax Sop.
Sax Tenor 2
mf
f
ff
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
ff
pp
3
Pno.
mf
ff
Ped.
Bx.
ff
A
♩ = 65
Bat.
ff

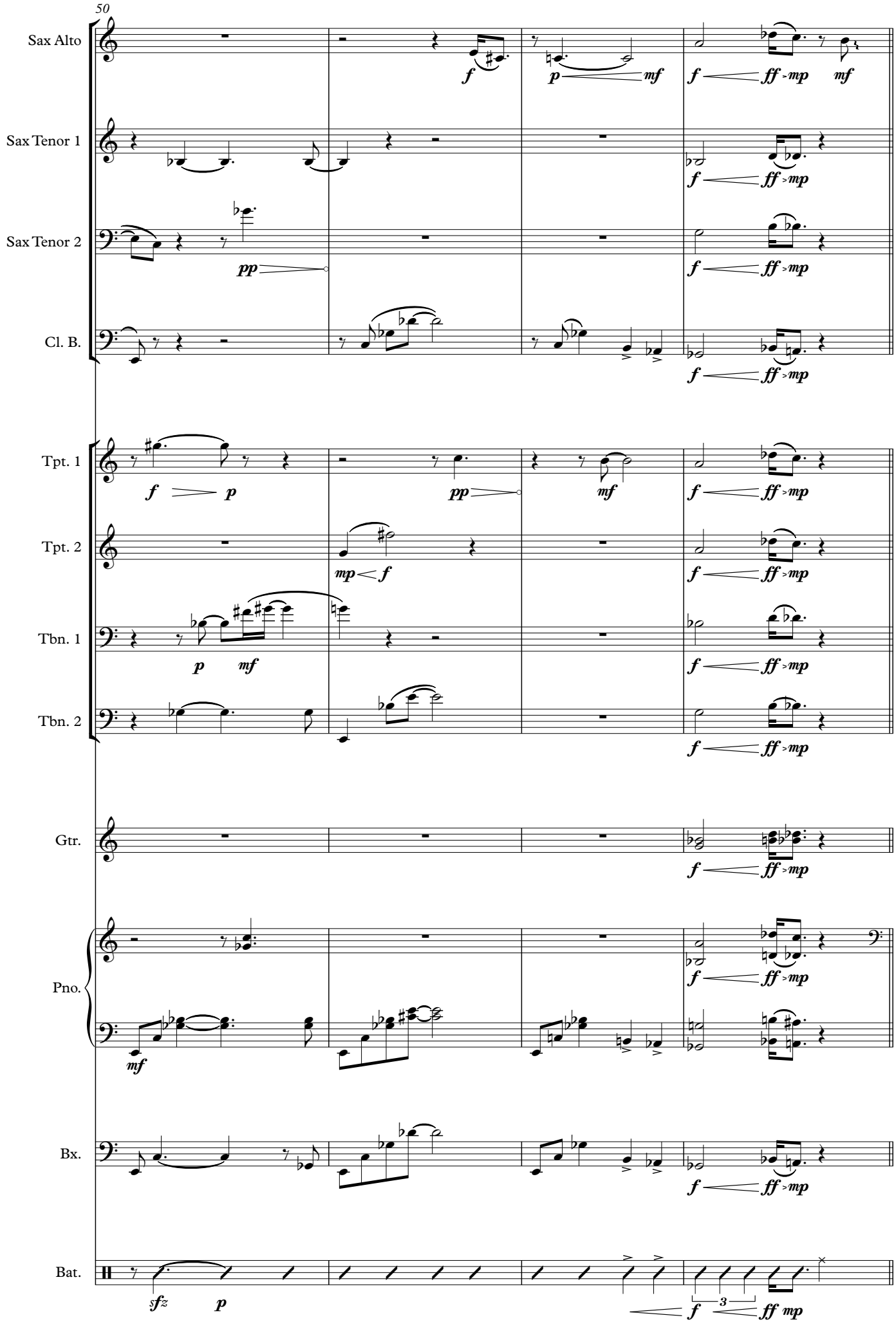

24
Fl.
Sax Sop.
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
ppp
p
mf
p
sfz
p
pp

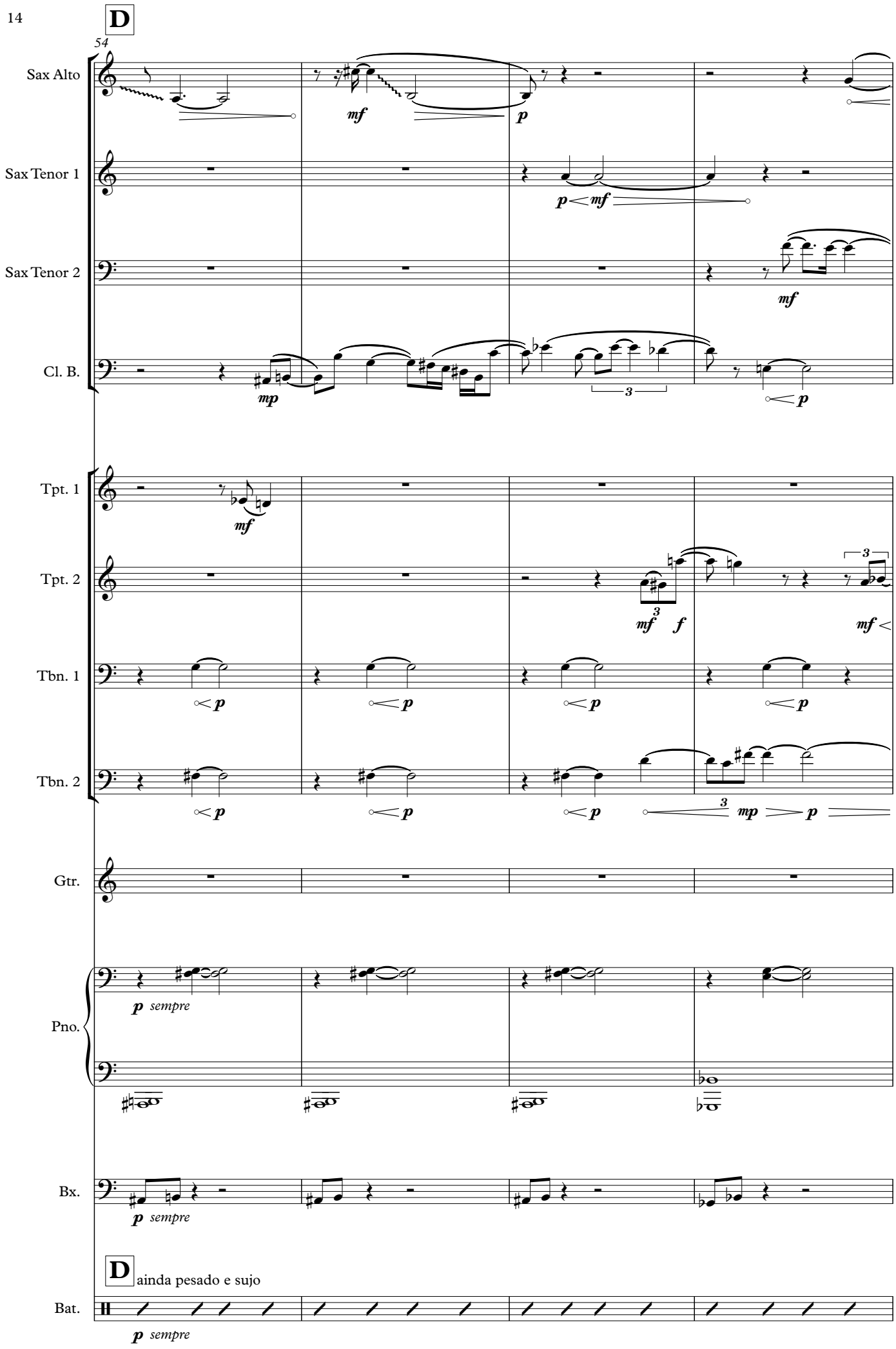

B
29
Sax Alto
Fl.
Sax Sop.
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
arco
B
Bat.

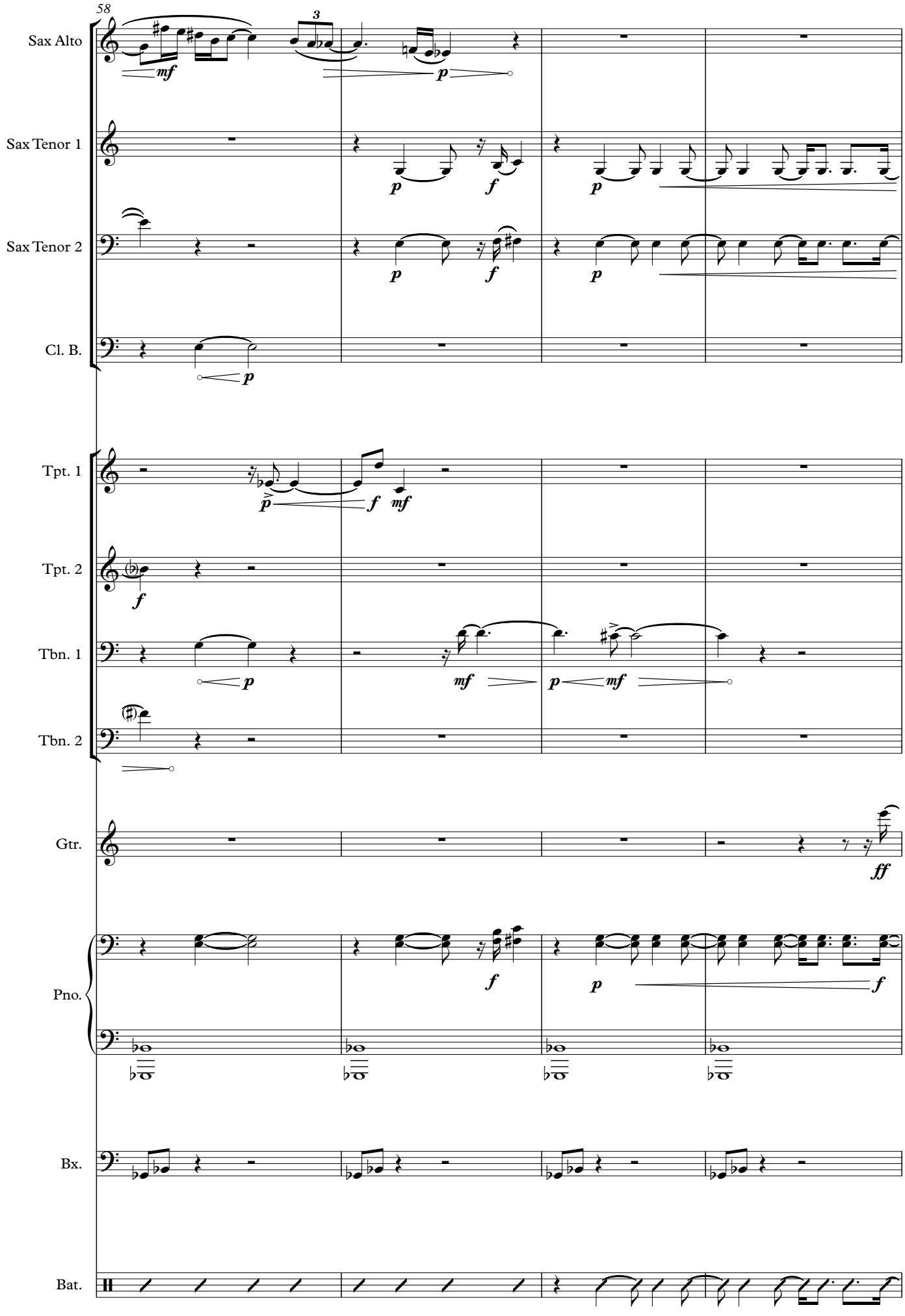

35
Sax Alto
Sax Sop.
Tenor Sax
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.

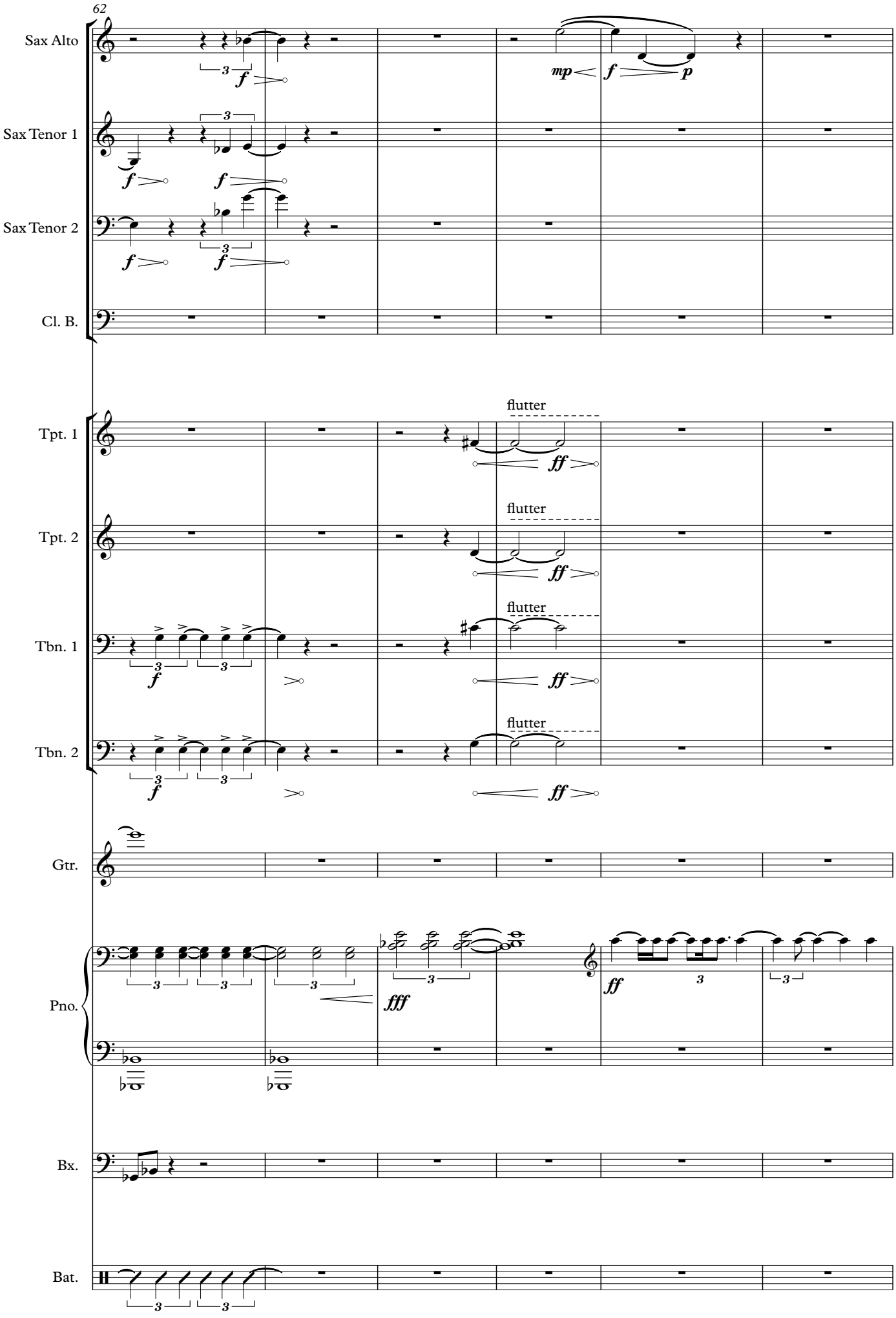

41
C
Sax Alto
Sax Tenor 1
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
pizz.
C
pesado, sujo, marcar o tempo
Bat.
pp
mp
f
p
sempre
mf
3


46
Sax Alto
Sax Tenor 1
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.


50
Sax Alto
Sax Tenor 1
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
f
p
mf
ff
mp
pp
mp < f
sfz
3


D
54
Sax Alto
Sax Tenor 1
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
D
ainda pesado e sujo
Bat.
sempre


58
Sax Alto
Sax Tenor 1
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
3
mf
p
f
ff


62
Sax Alto
Sax Tenor 1
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
flutter
flutter
flutter
flutter

E TENOR 1 SOLO

68

Sax Alto

Sax Tenor 1

Sax Tenor 2

Cl. B.

Tpt. 1

Tpt. 2

Tbn. 1

Tbn. 2

Gtr.

Pno.

Bx.

Bat.

pickup 2 cc.

flutter

flutter

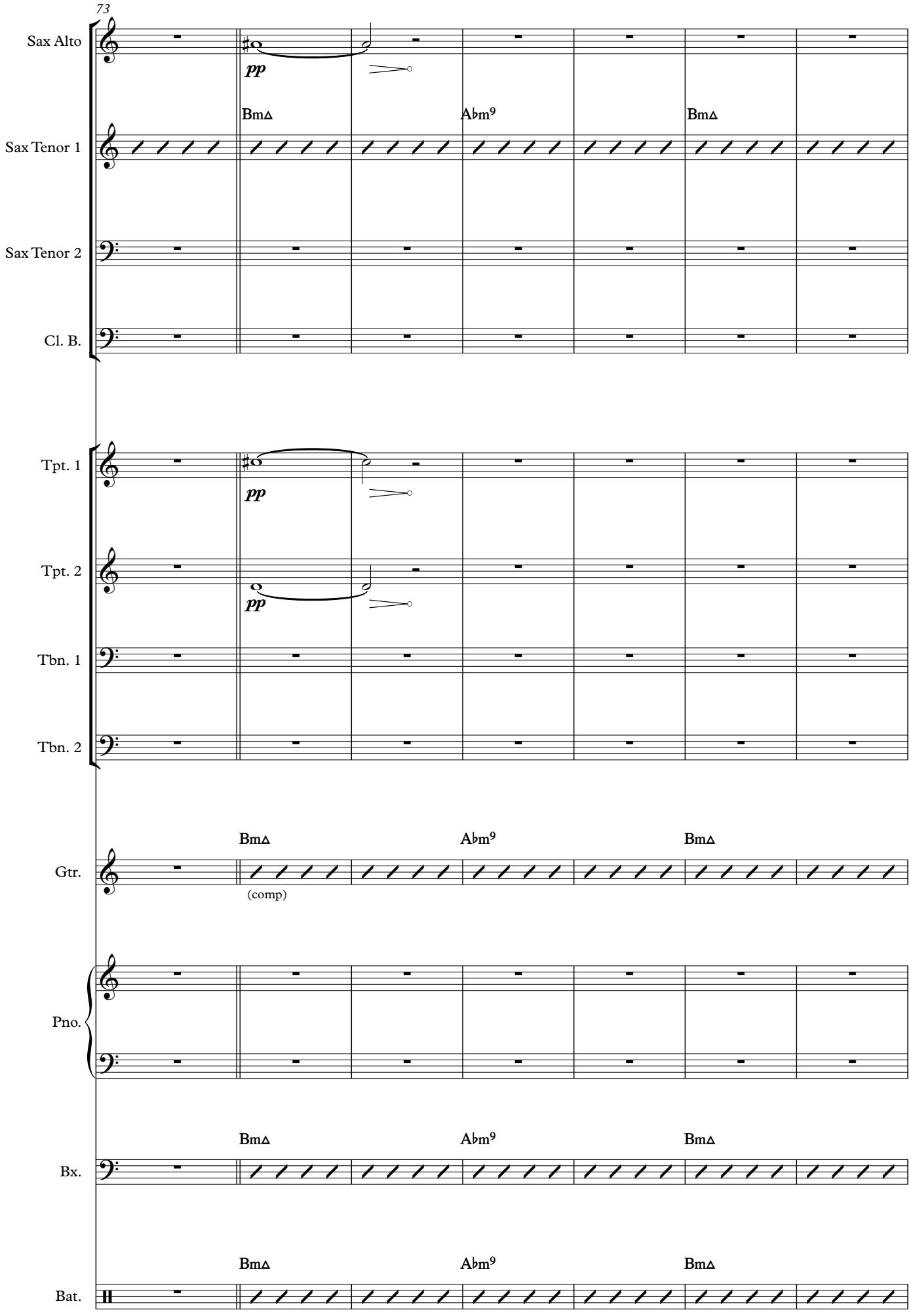
flutter

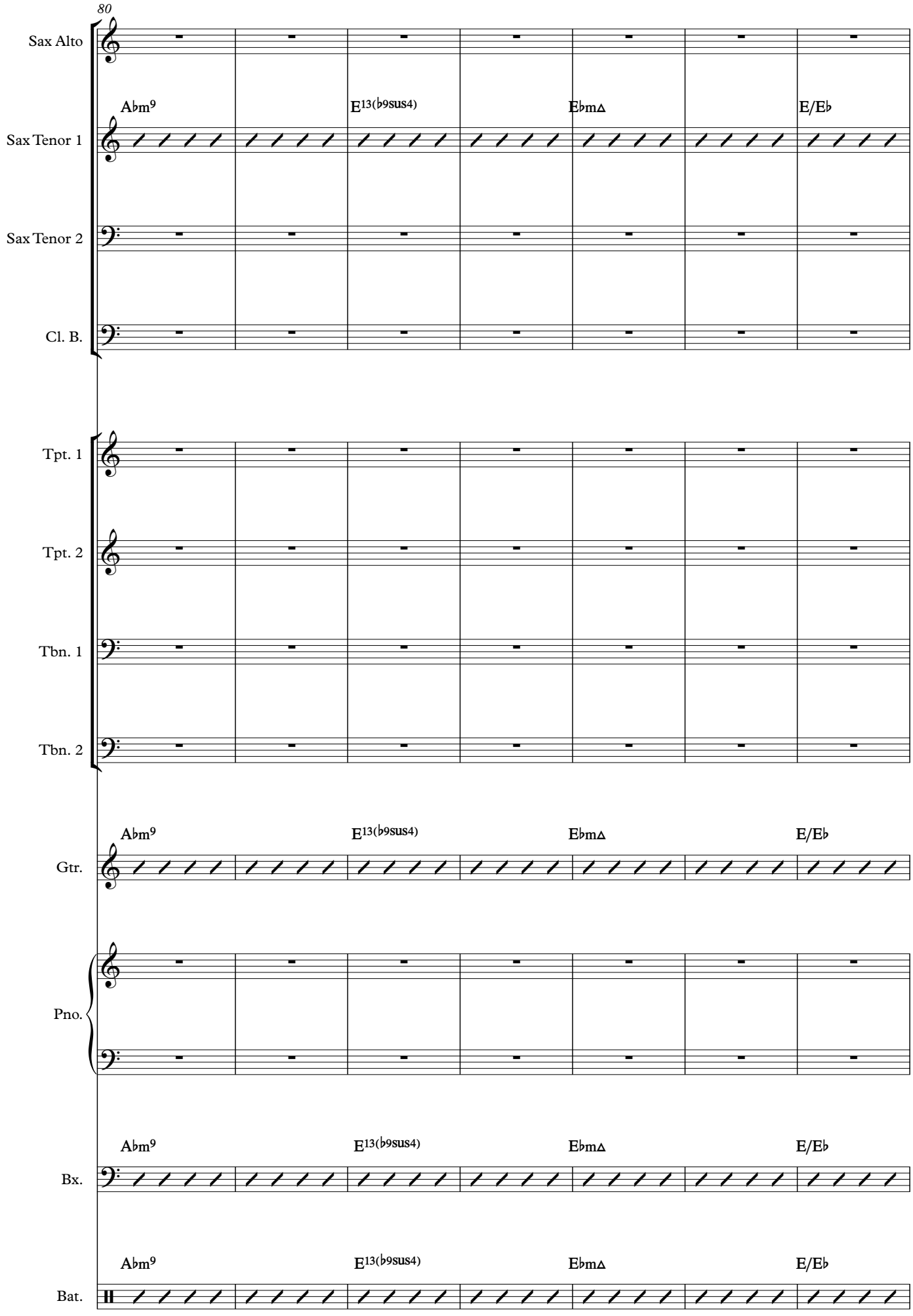
flutter

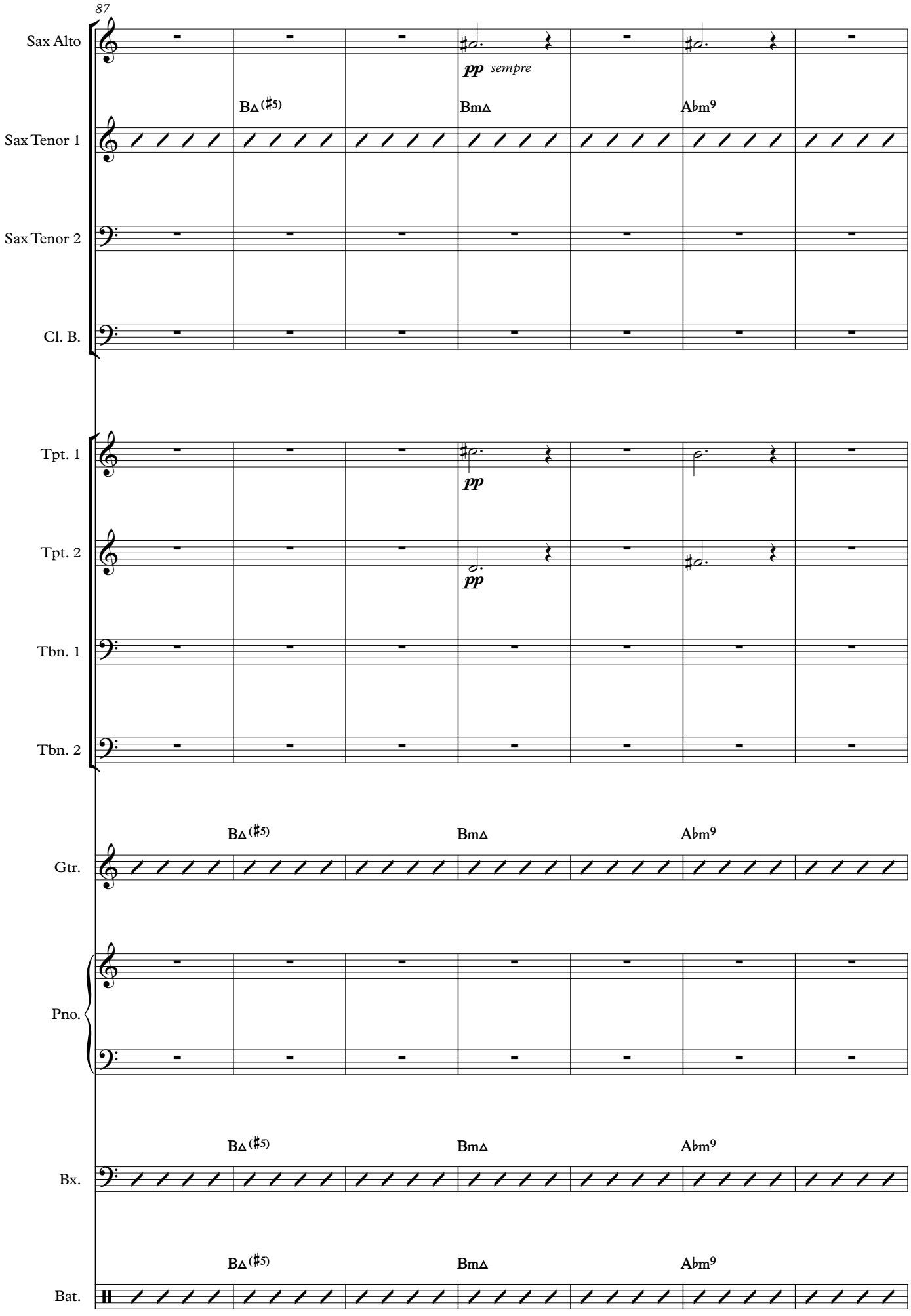
7

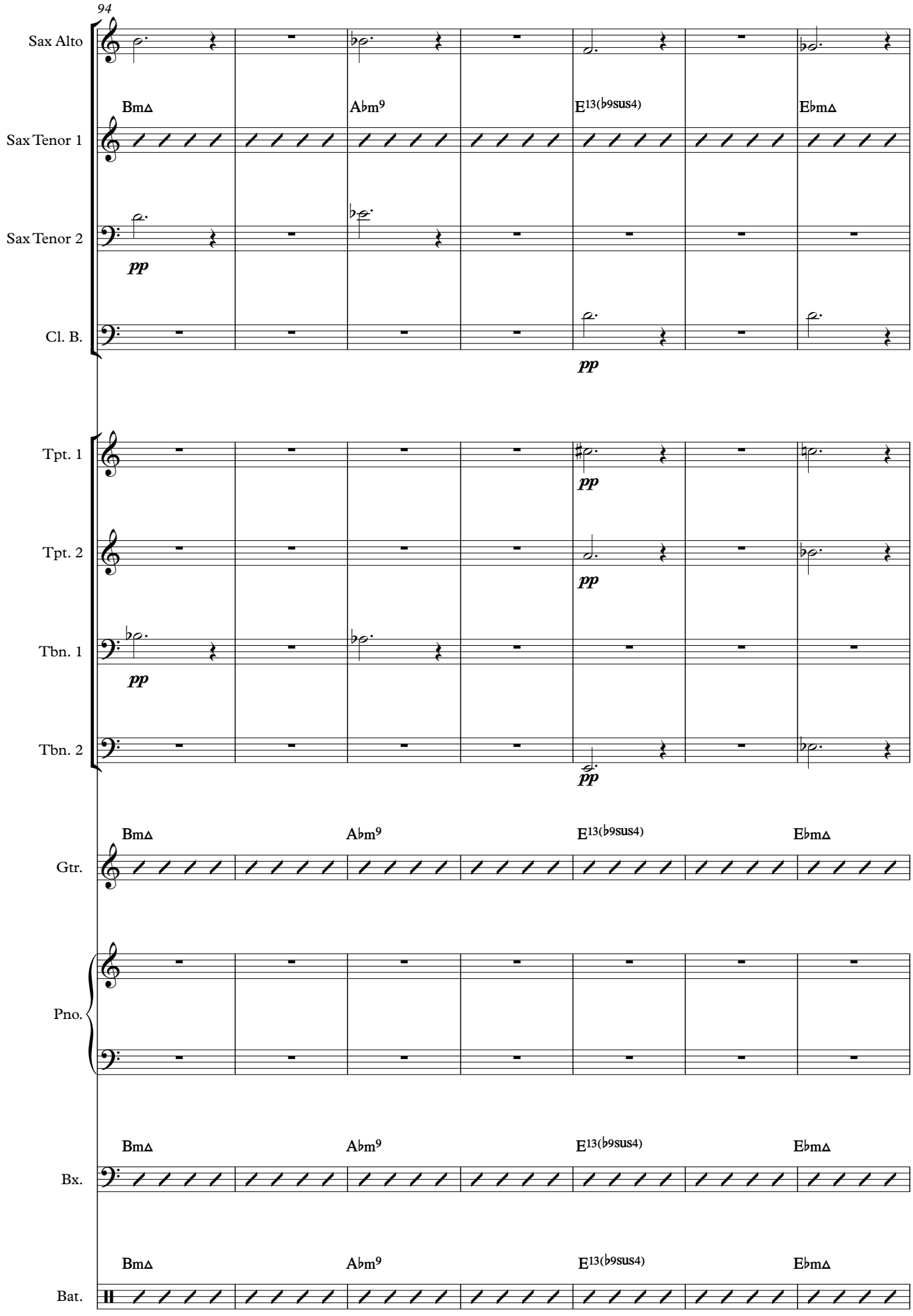
3

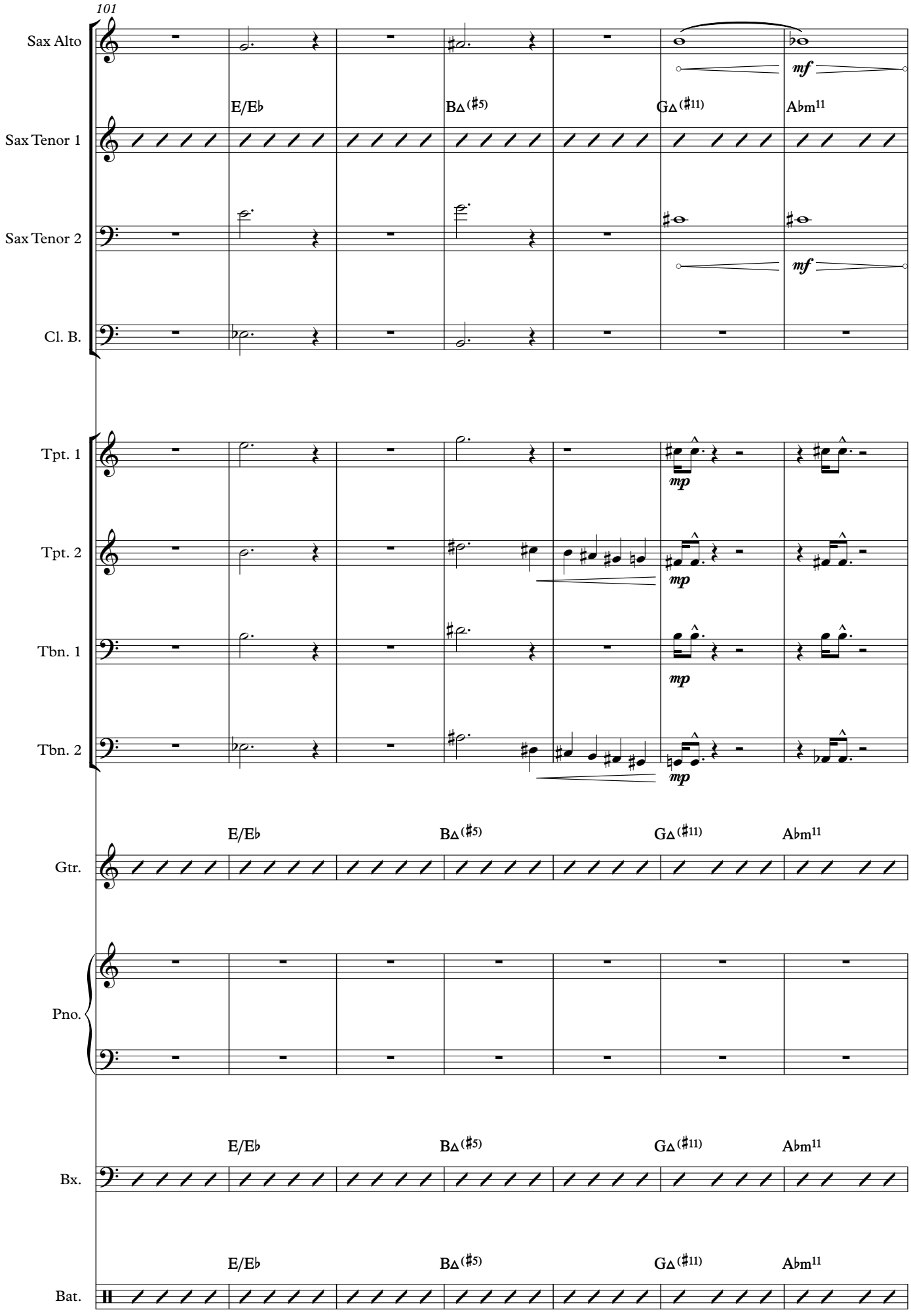
E

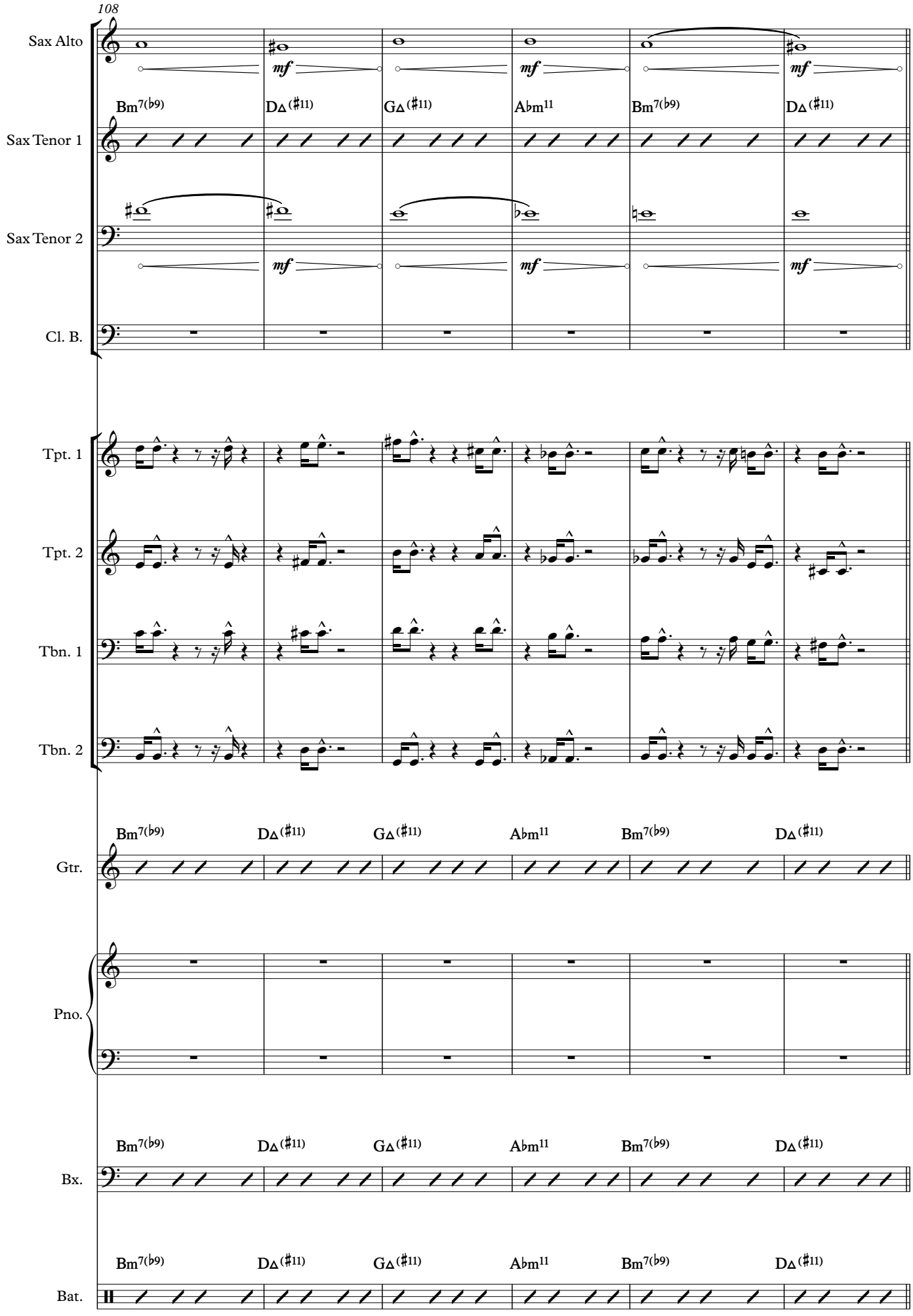

73
Sax Alto
Sax Tenor 1
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
pp
Bm△
A♭m9
Bm△
(comp)

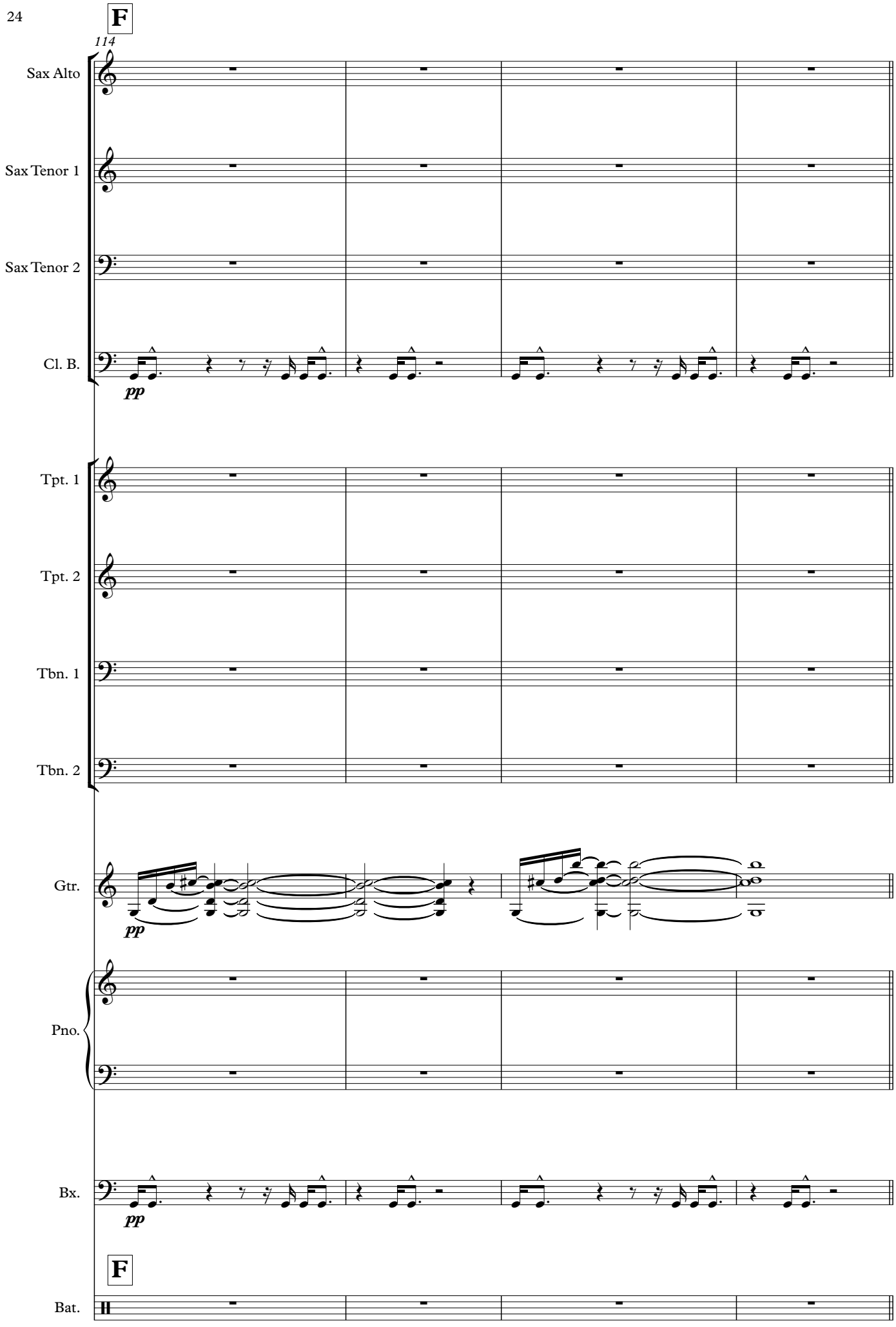

80
Sax Alto
Sax Tenor 1
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
A♭m9
E13(♭9sus4)
E♭m△
E/E♭

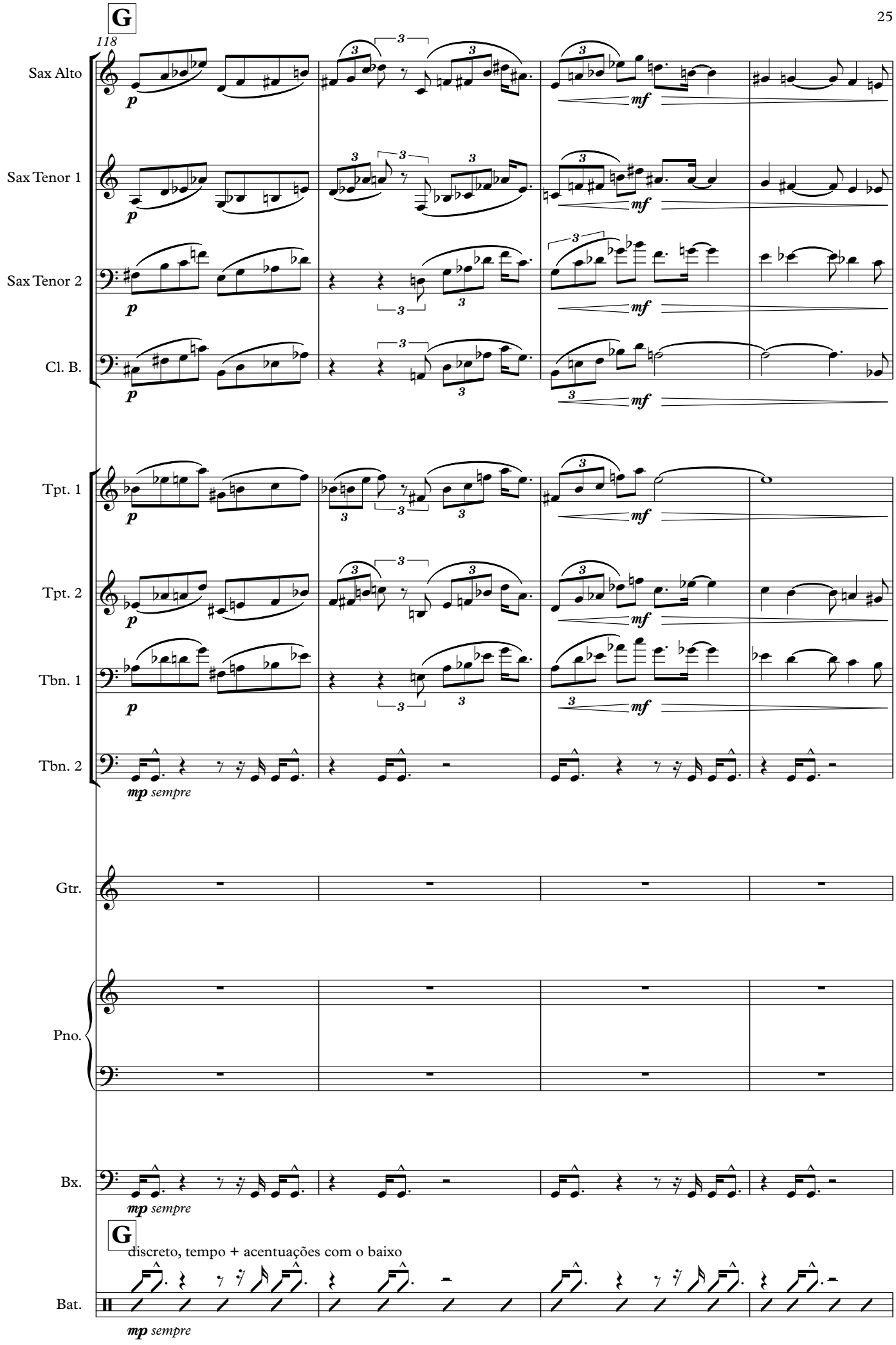

87
Sax Alto
Sax Tenor 1
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
pp sempre
pp
BΔ(♯5)
BmΔ
A♭m9

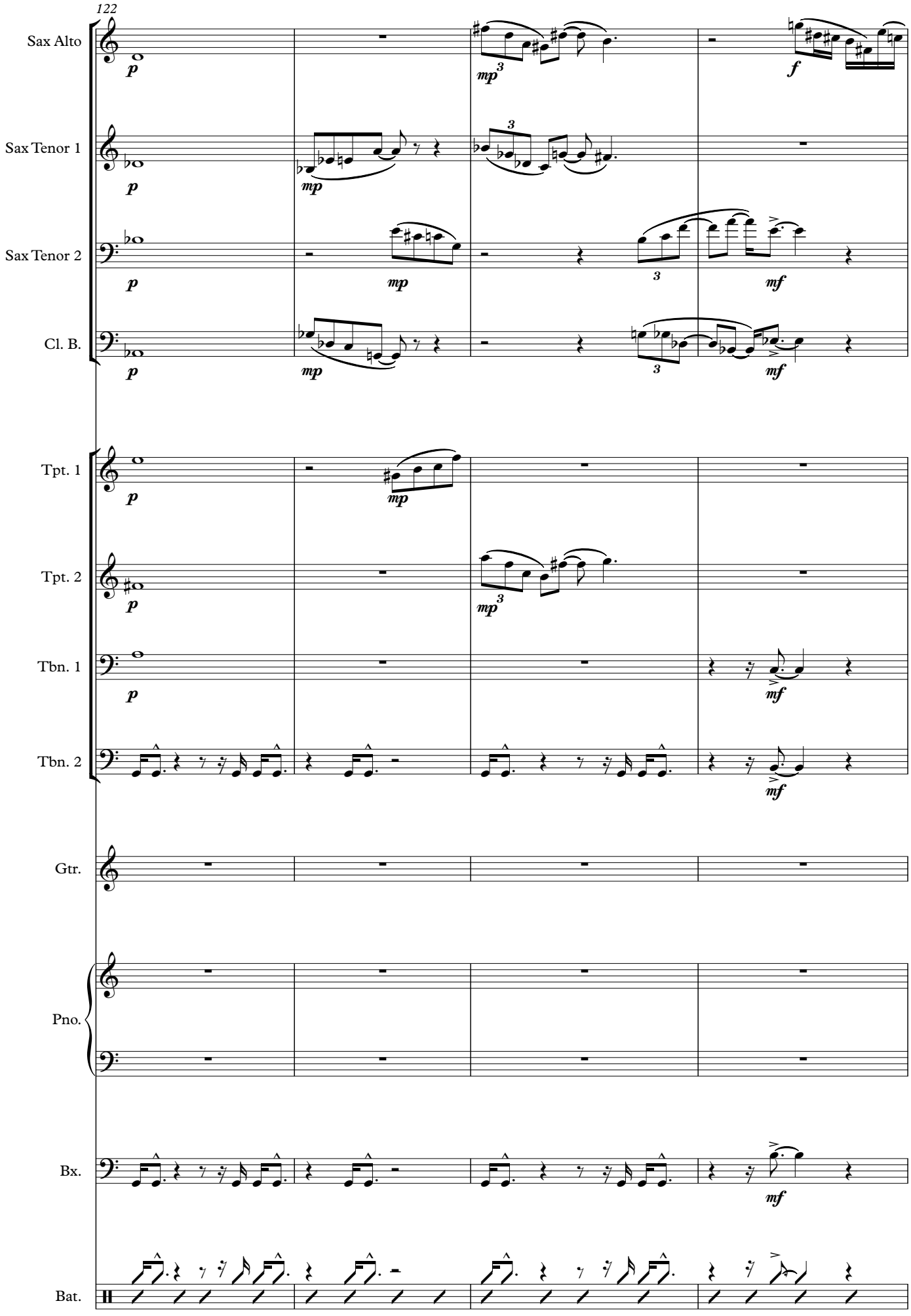

94
Sax Alto
Sax Tenor 1
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
BmΔ
A♭m9
E13(♭9sus4)
E♭mΔ
pp

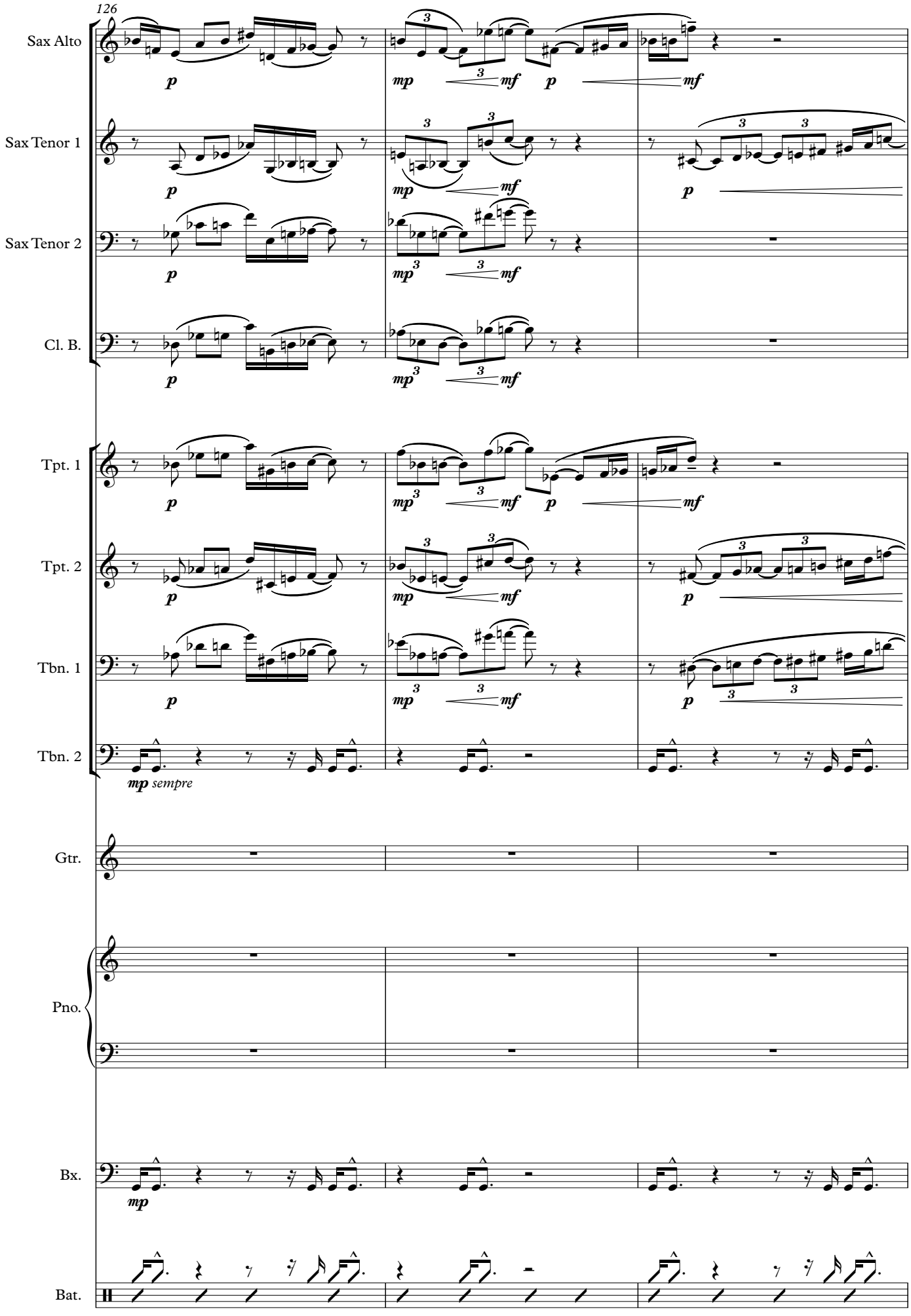

101
Sax Alto
Sax Tenor 1
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
mf
mp
E/E♭
B♭Δ(♯5)
G♭Δ(♯11)
A♭m11

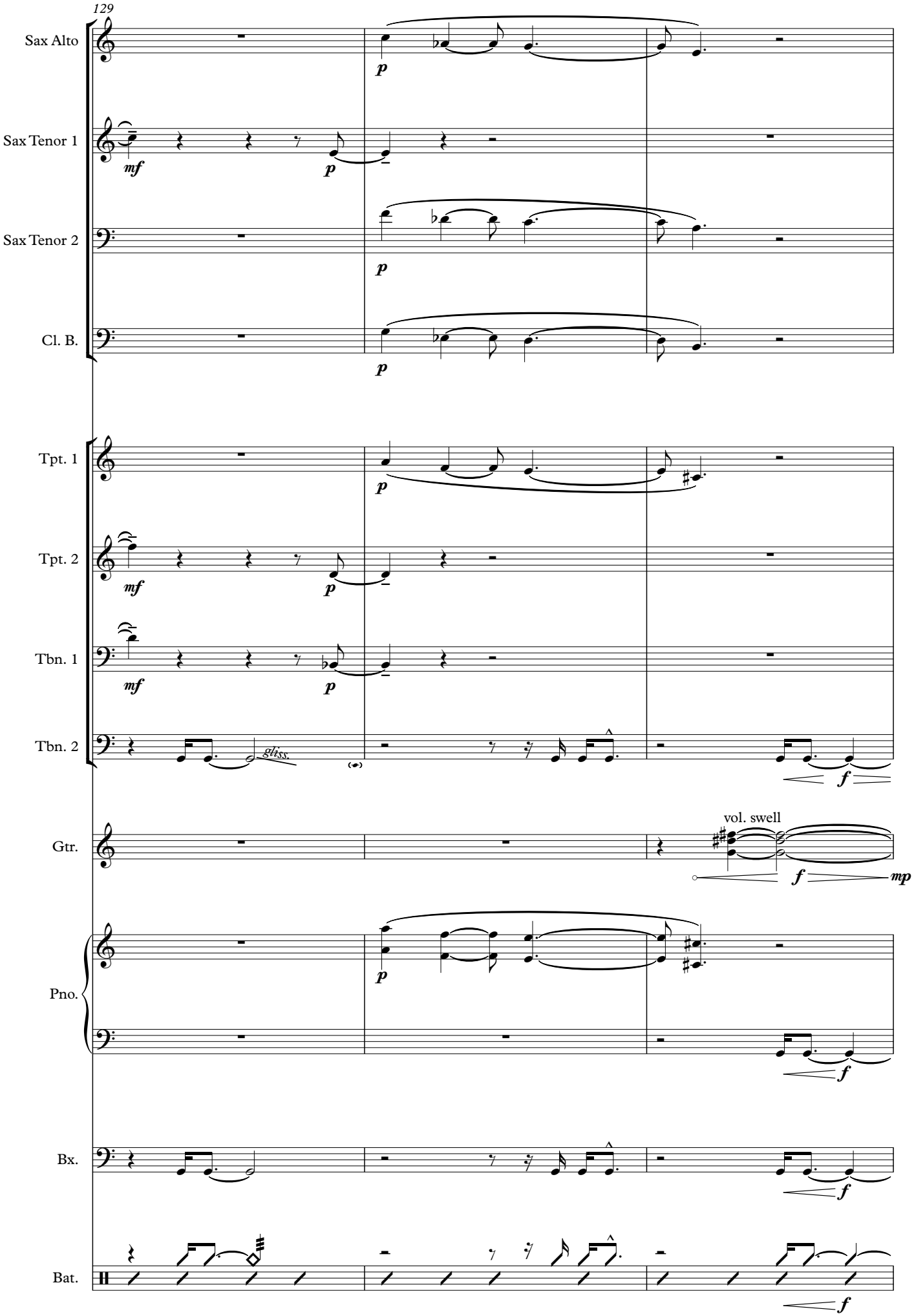

108
Sax Alto
Sax Tenor 1
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
mf
Bm7(♭9)
D△(♯11)
G△(♯11)
A♭m11
Bm7(♭9)
D△(♯11)

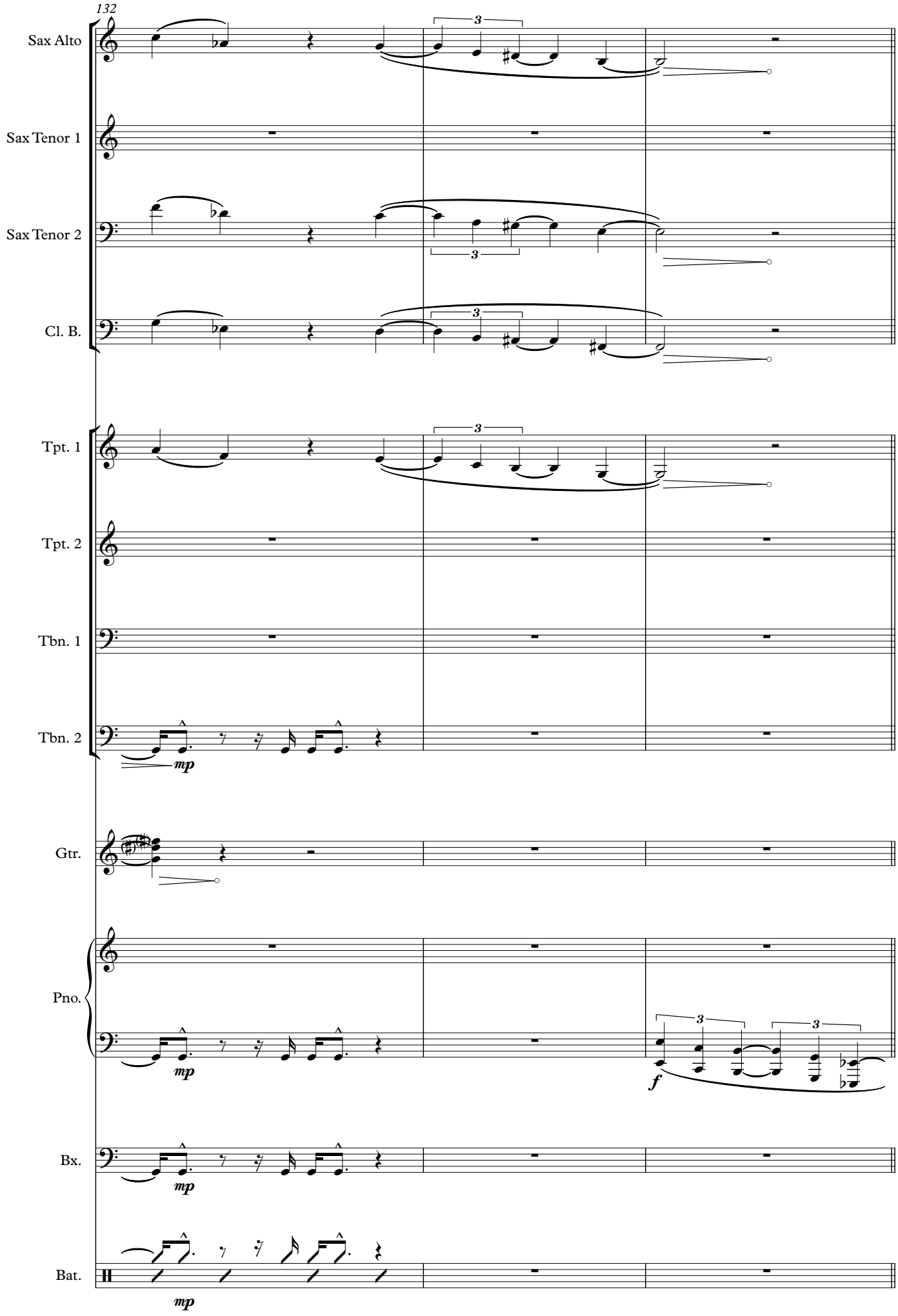

F
114
Sax Alto
Sax Tenor 1
Sax Tenor 2
Cl. B.
pp
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
pp
Pno.
Bx.
pp
F
Bat.

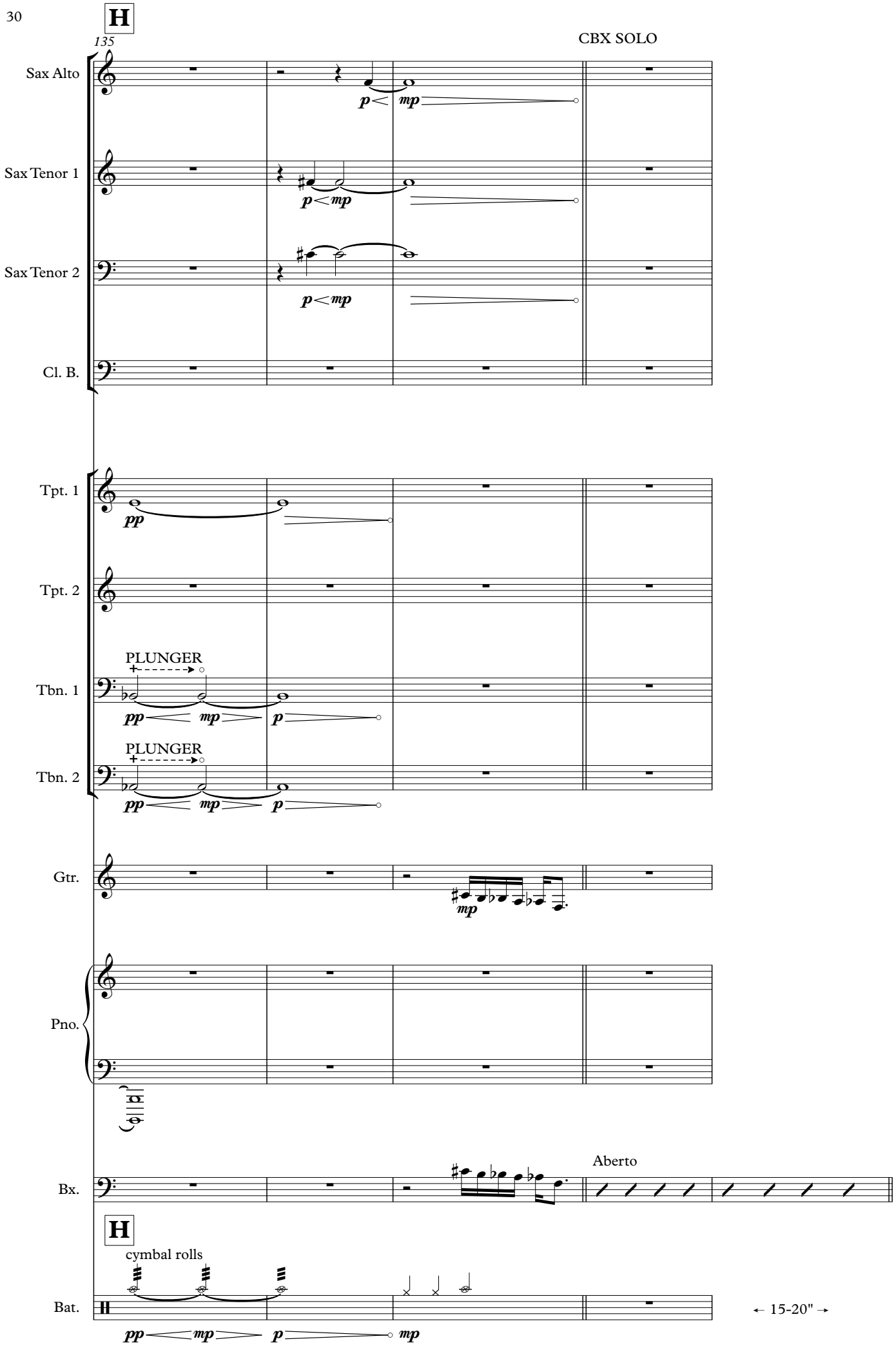

G
118
Sax Alto
Sax Tenor 1
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
p
mf
mp sempre
G
discreto, tempo + acentuações com o baixo

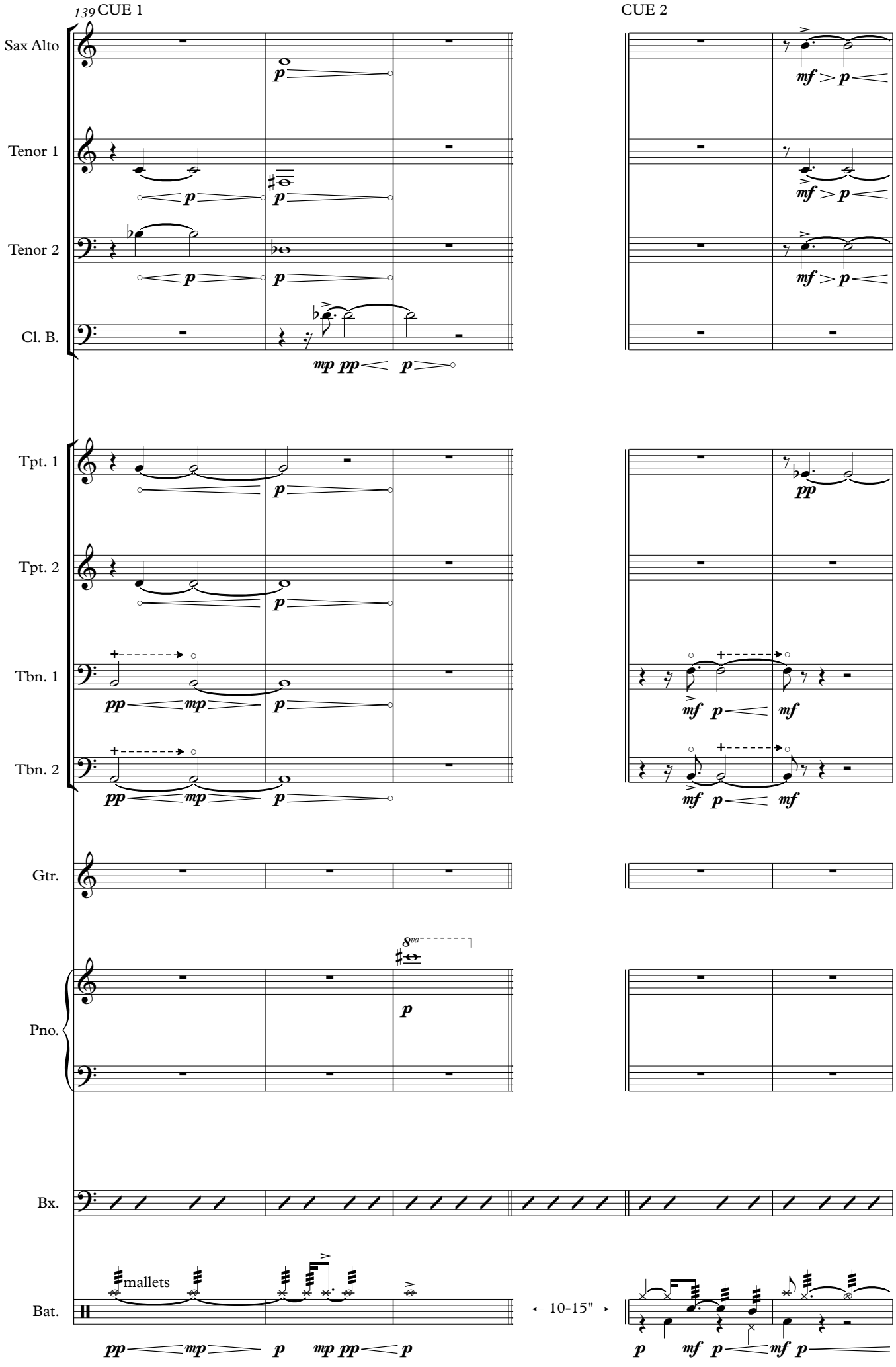

122
Sax Alto
Sax Tenor 1
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.

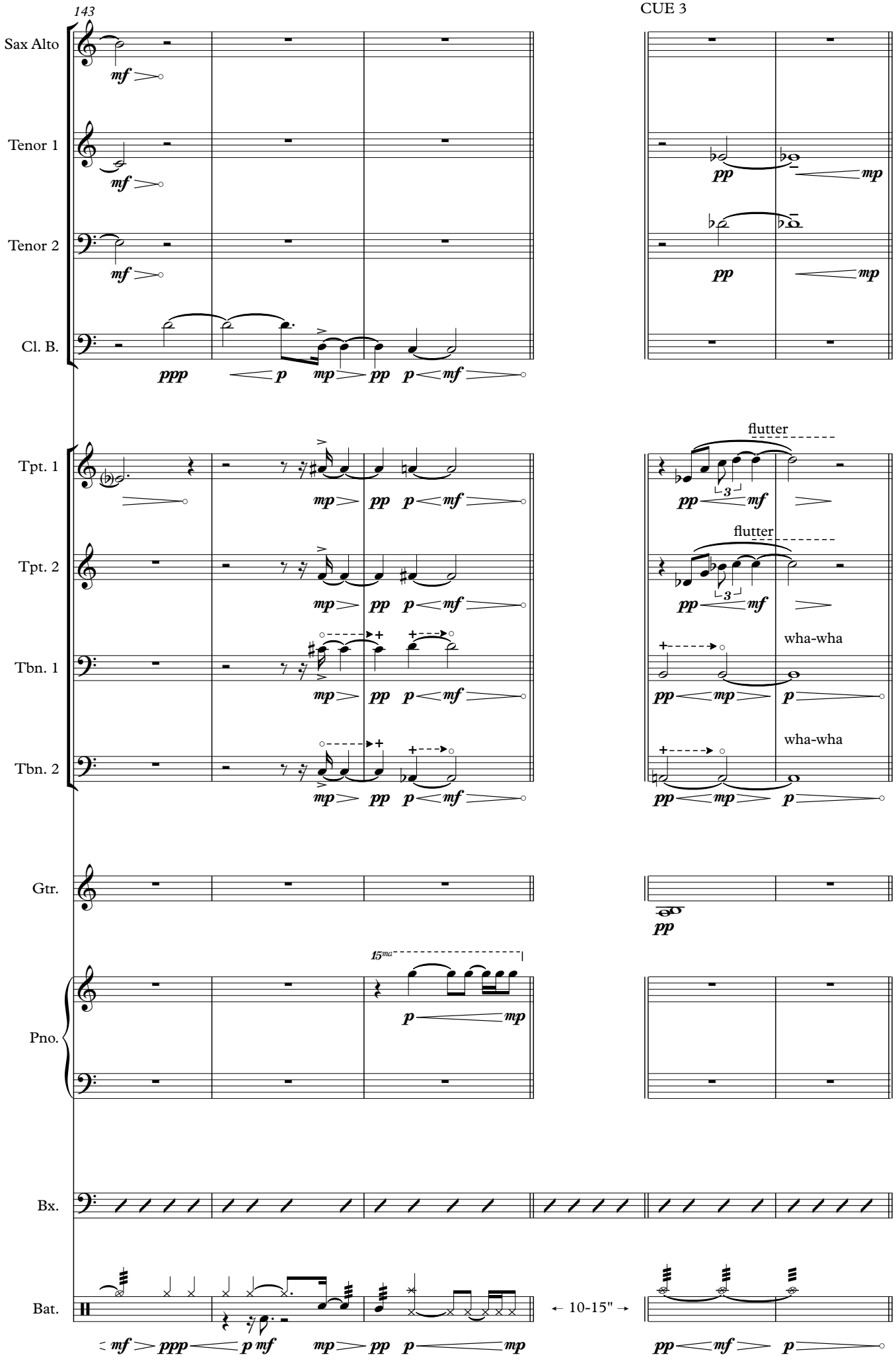

126
Sax Alto
Sax Tenor 1
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
mp sempre
Gtr.
Pno.
Bx.
Bat.


129
Sax Alto
Sax Tenor 1
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
gliss.
Gtr.
vol. swell
Pno.
Bx.
Bat.

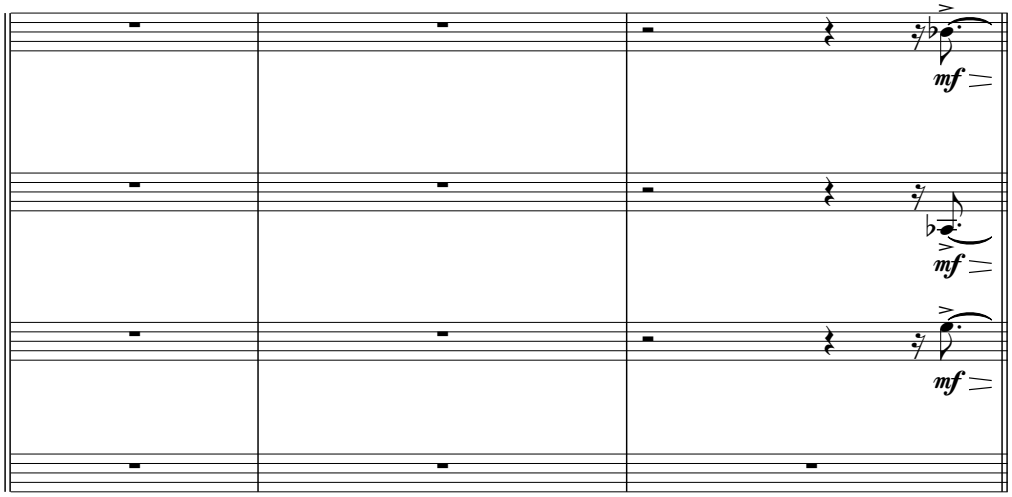

132
Sax Alto
Sax Tenor 1
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
3
mp
f

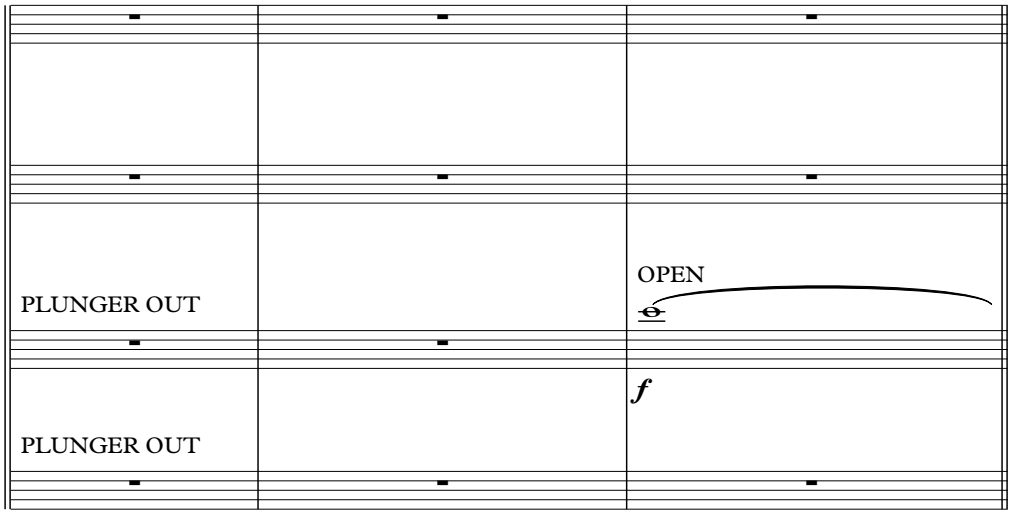

H
135
CBX SOLO
Sax Alto
Sax Tenor 1
Sax Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
PLUNGER
Tbn. 1
PLUNGER
Tbn. 2
Gtr.
Pno.
Bx.
Aberto
H
cymbal rolls
Bat.
← 15-20" →

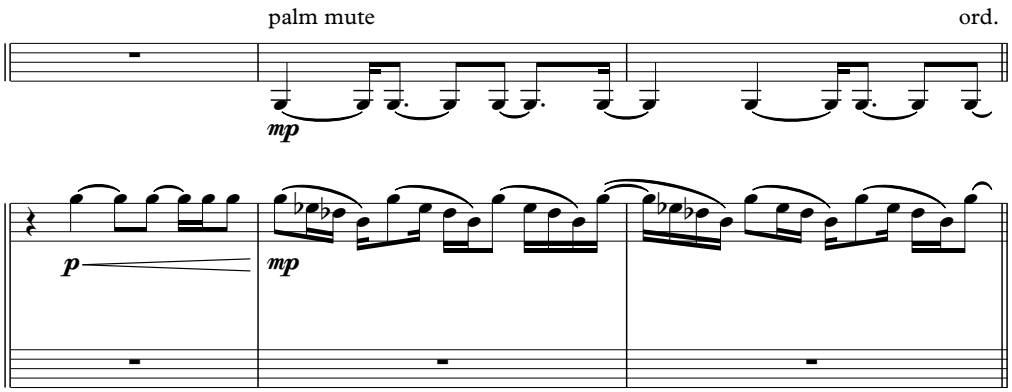

139 CUE 1
CUE 2
Sax Alto
Tenor 1
Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
mallets
8va
← 10-15" →

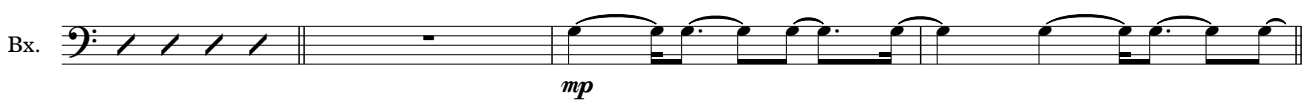

143
CUE 3
Sax Alto
Tenor 1
Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
flutter
flutter
wha-wha
wha-wha
15ma
← 10-15" →

ON CUE

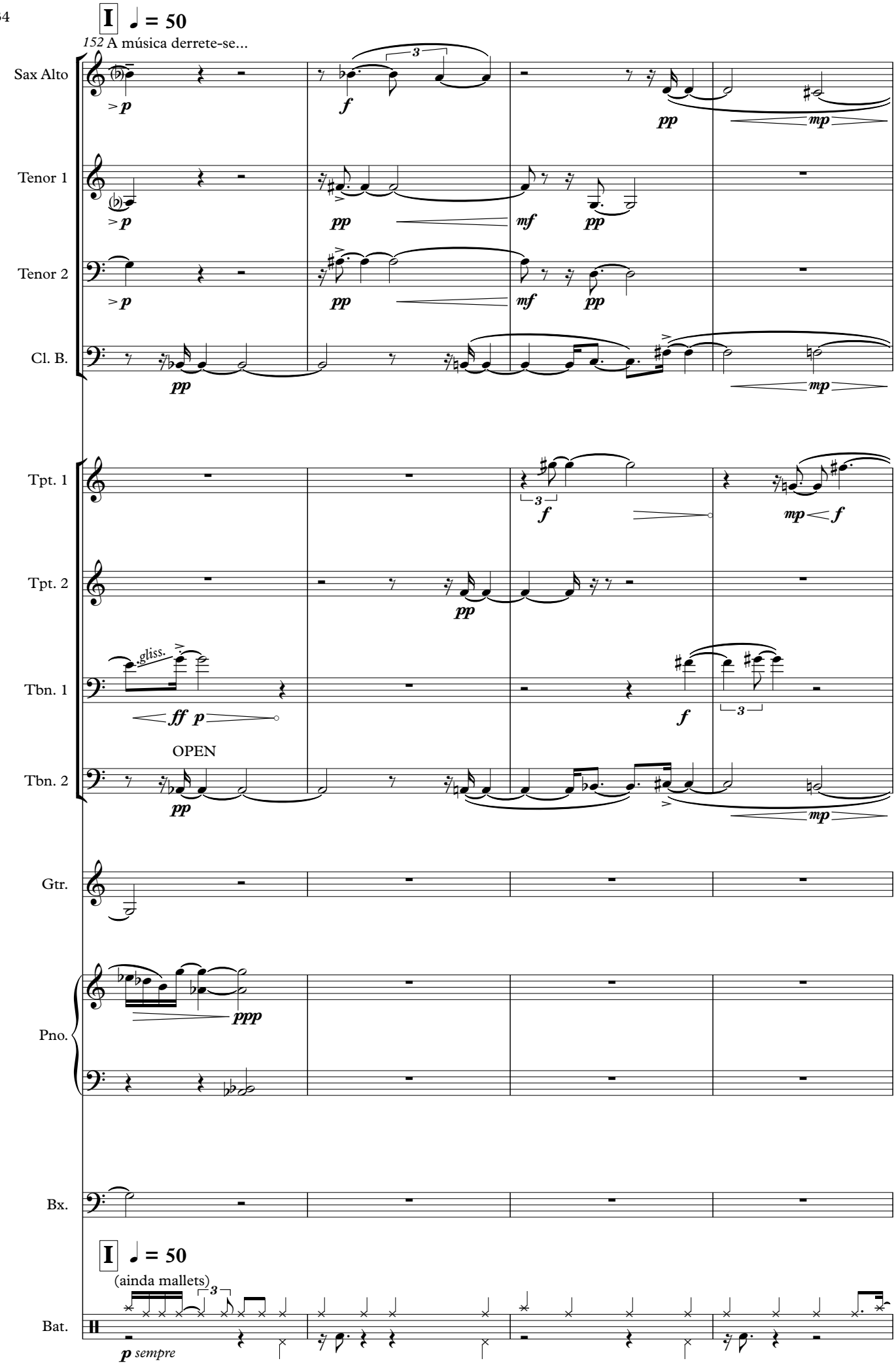

mf
mf
mf

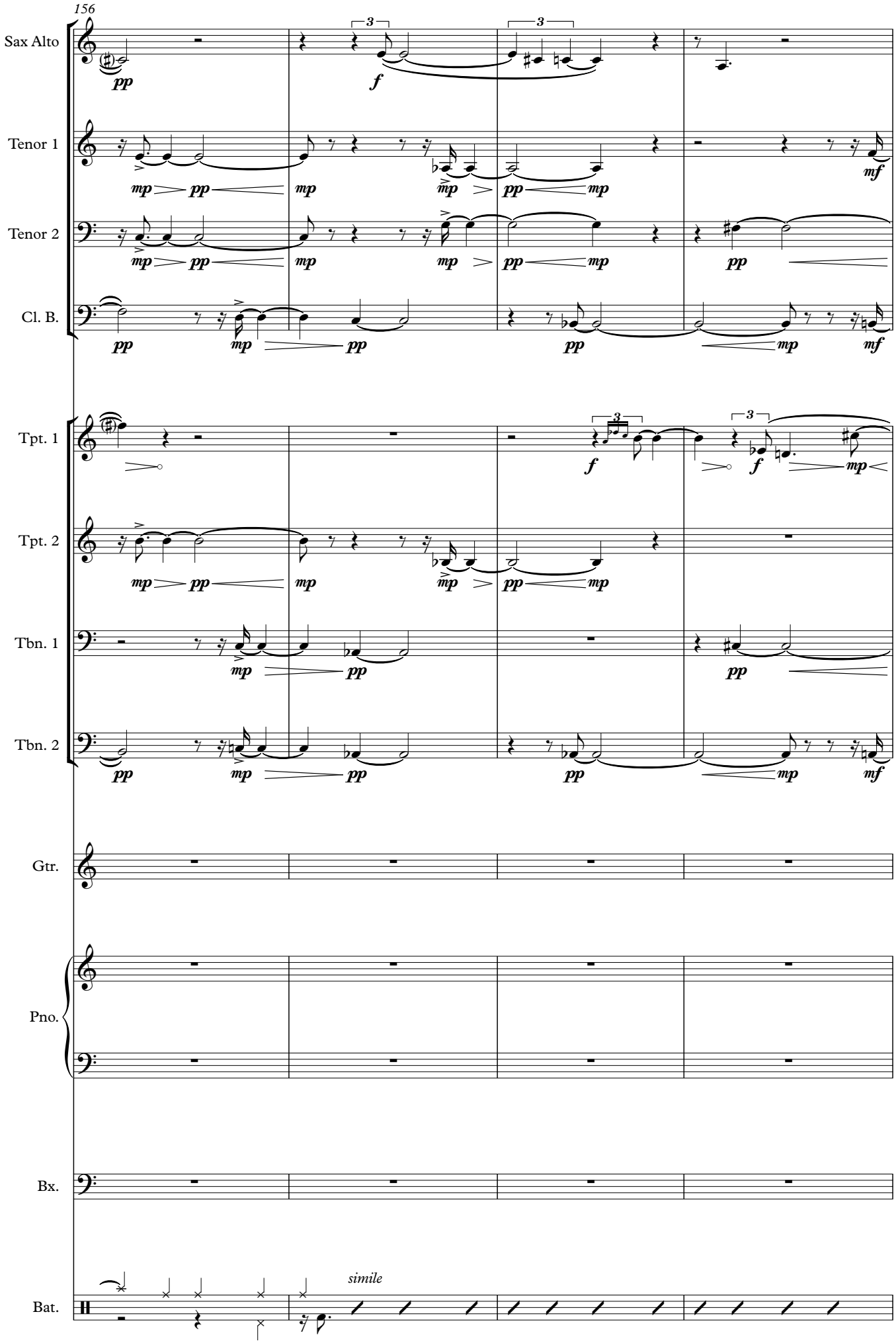

PLUNGER OUT
OPEN
f
PLUNGER OUT

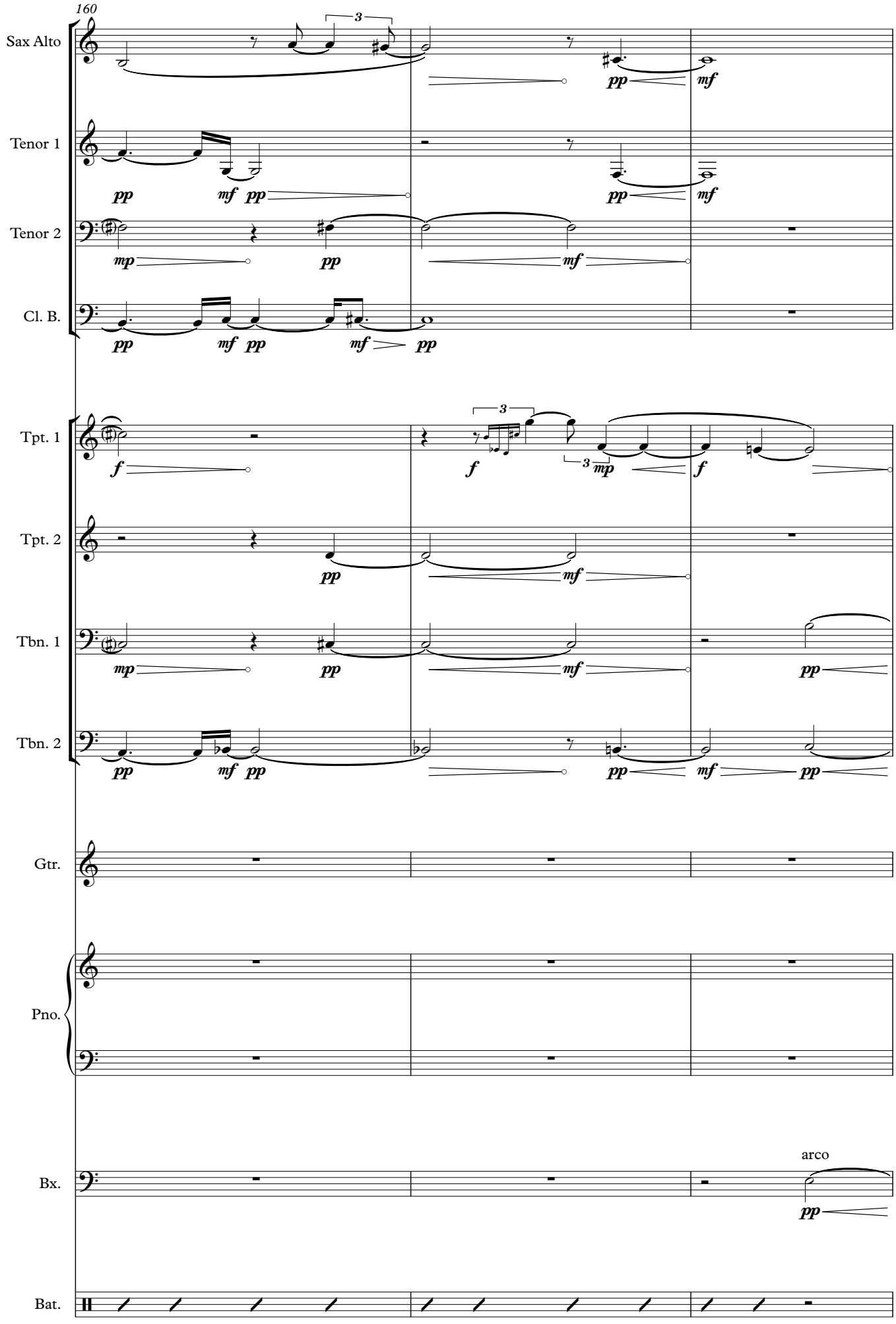

palm mute
ord.
mp
p
mp

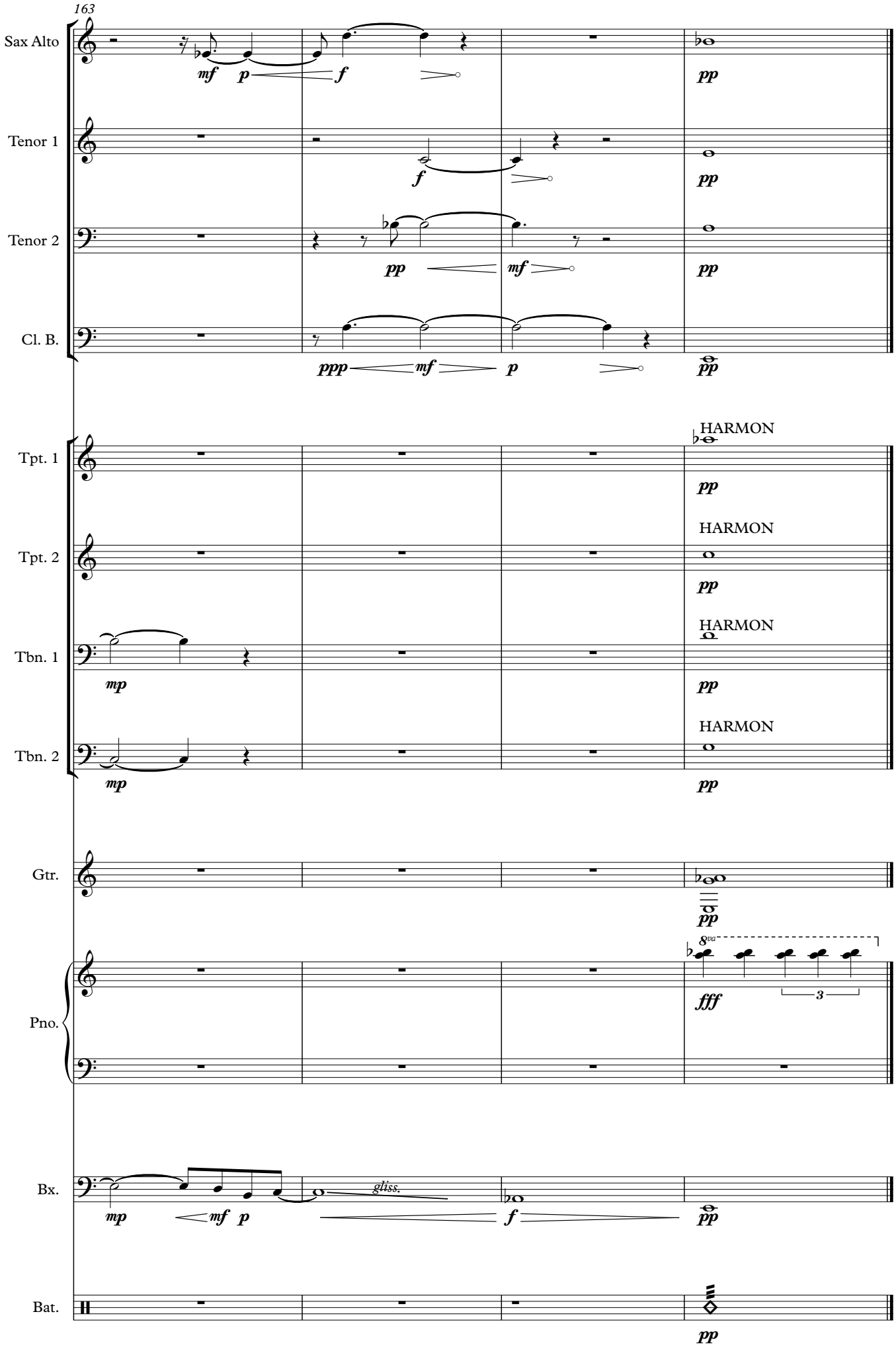

Bx.
mp


← 5-10" →
pp p mp


I ♩ = 50
152 A música derrete-se...
Sax Alto
Tenor 1
Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
OPEN
gliss.
Gtr.
Pno.
Bx.
I ♩ = 50
(ainda mallets)
Bat.
p sempre


156
Sax Alto
Tenor 1
Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
simile

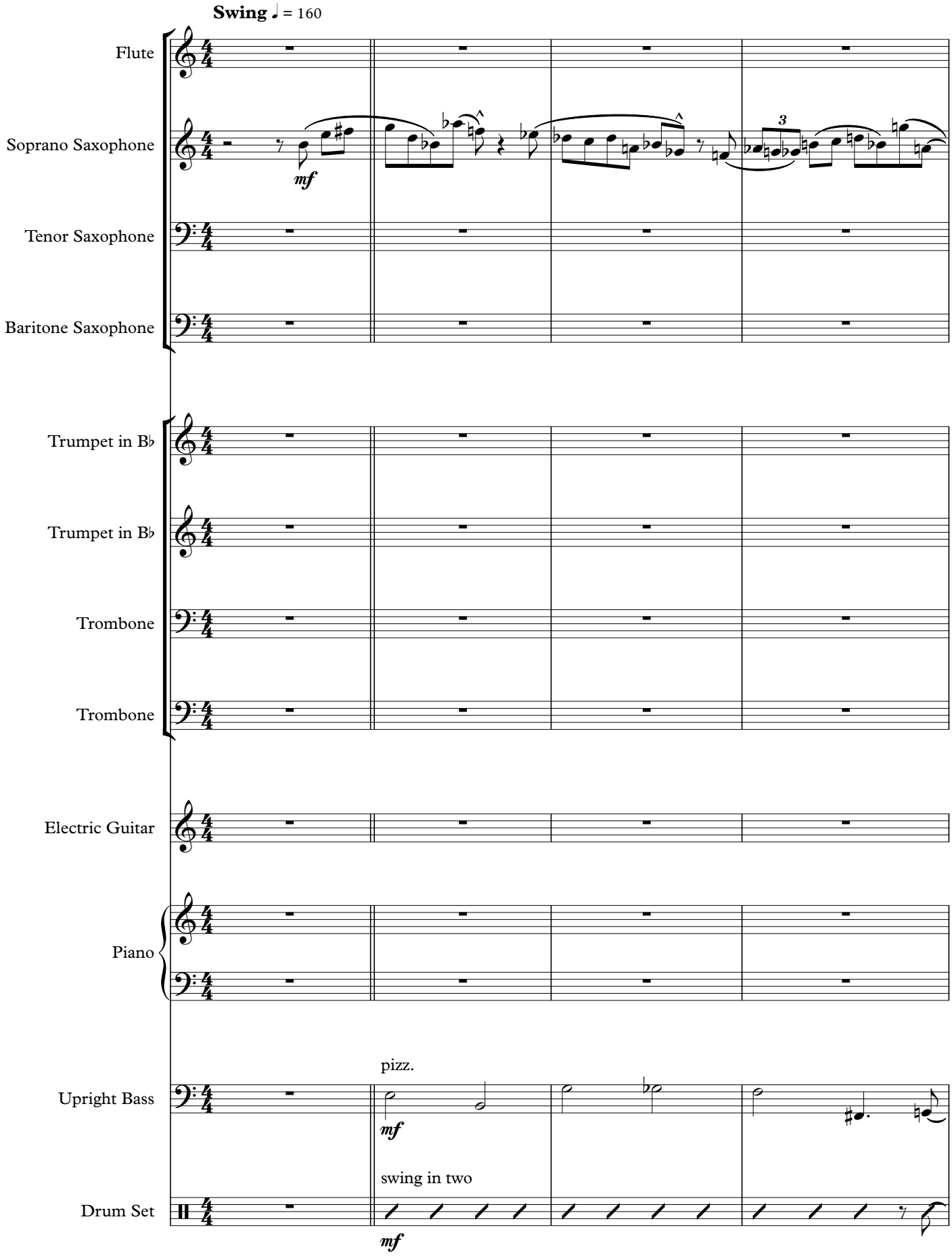

160
Sax Alto
Tenor 1
Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
arco

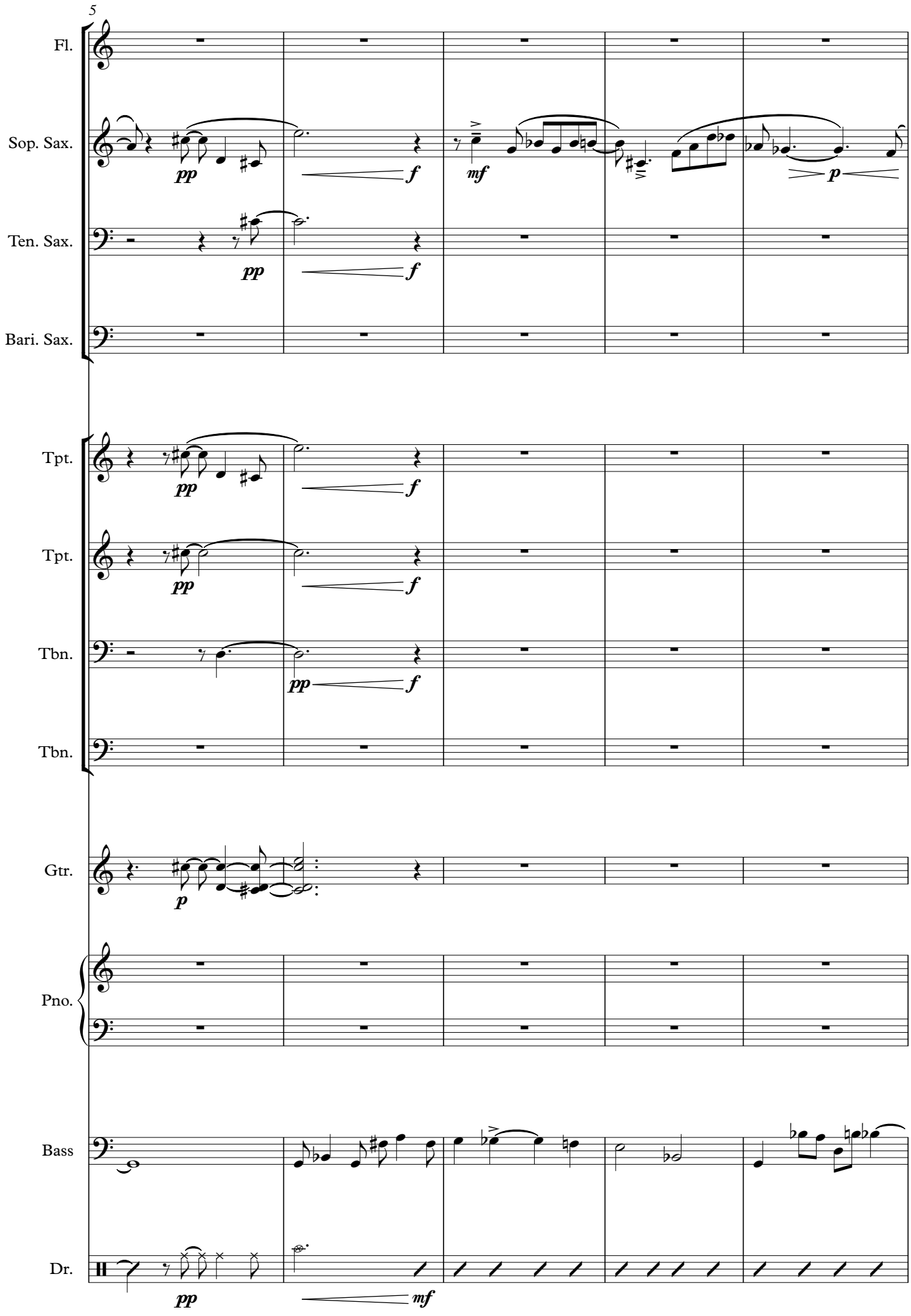

163
Sax Alto
Tenor 1
Tenor 2
Cl. B.
Tpt. 1
Tpt. 2
Tbn. 1
Tbn. 2
Gtr.
Pno.
Bx.
Bat.
HARMON
gliss.
8va
3

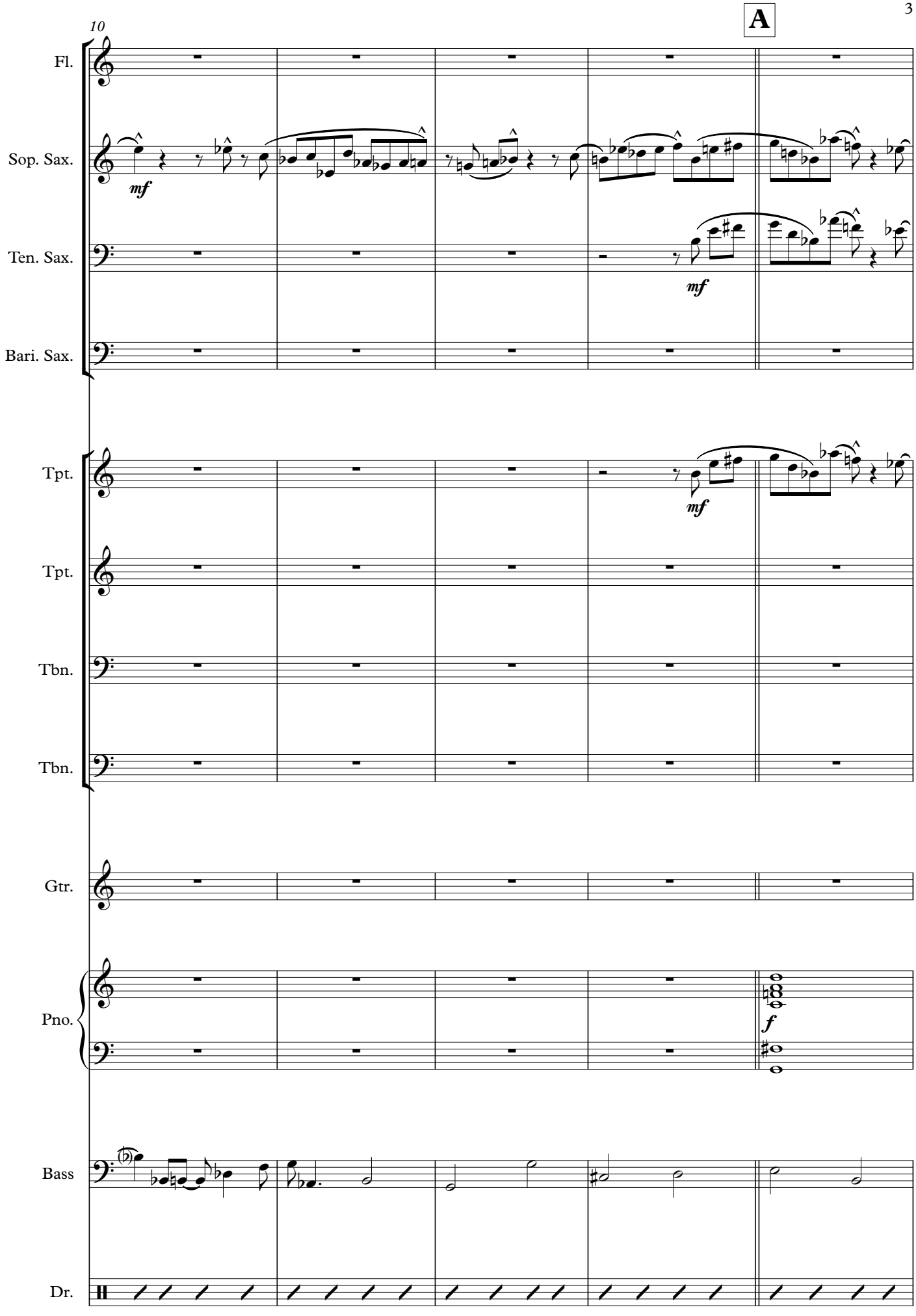
4.3 Três, para ensemble de Jazz

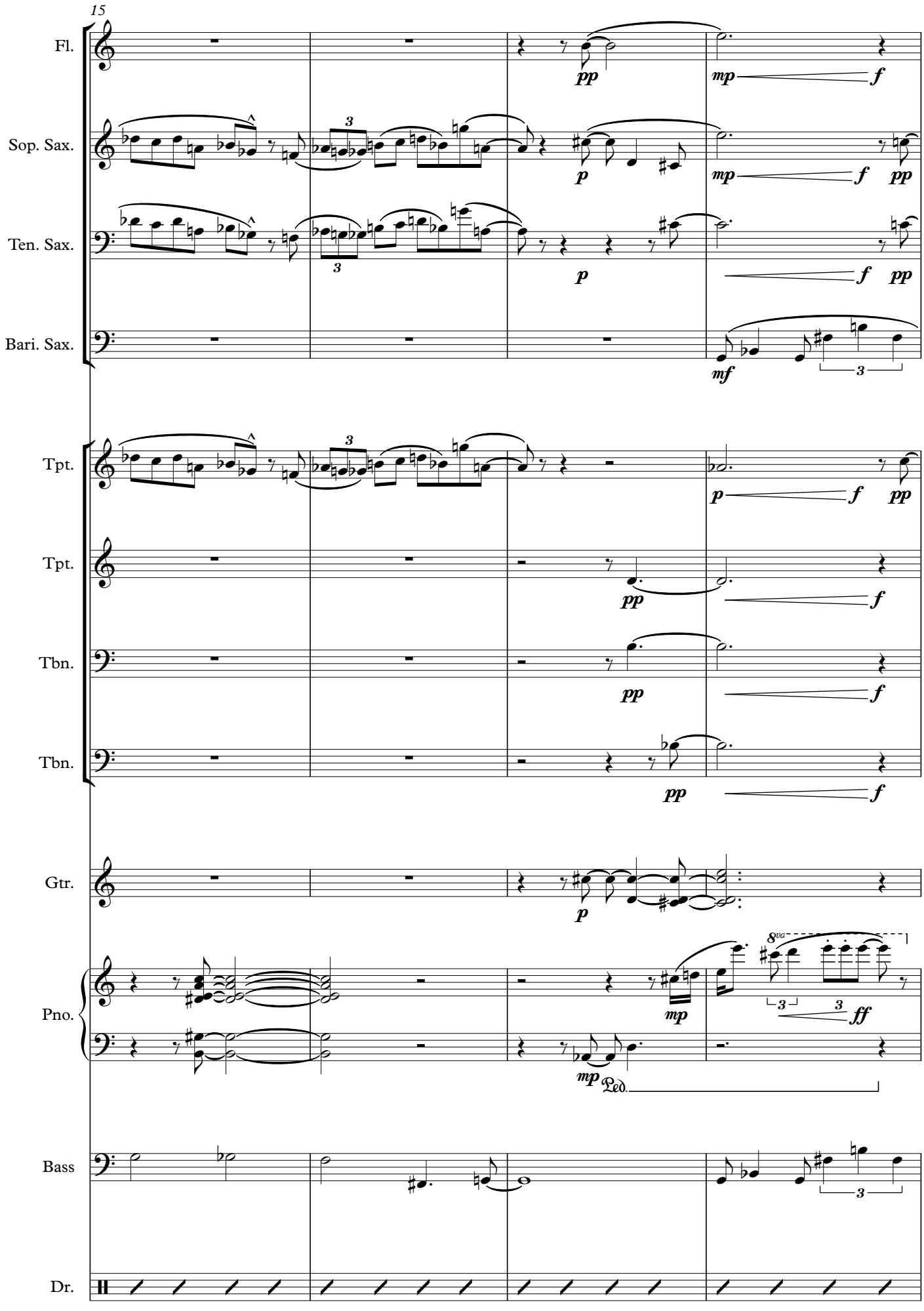
Três

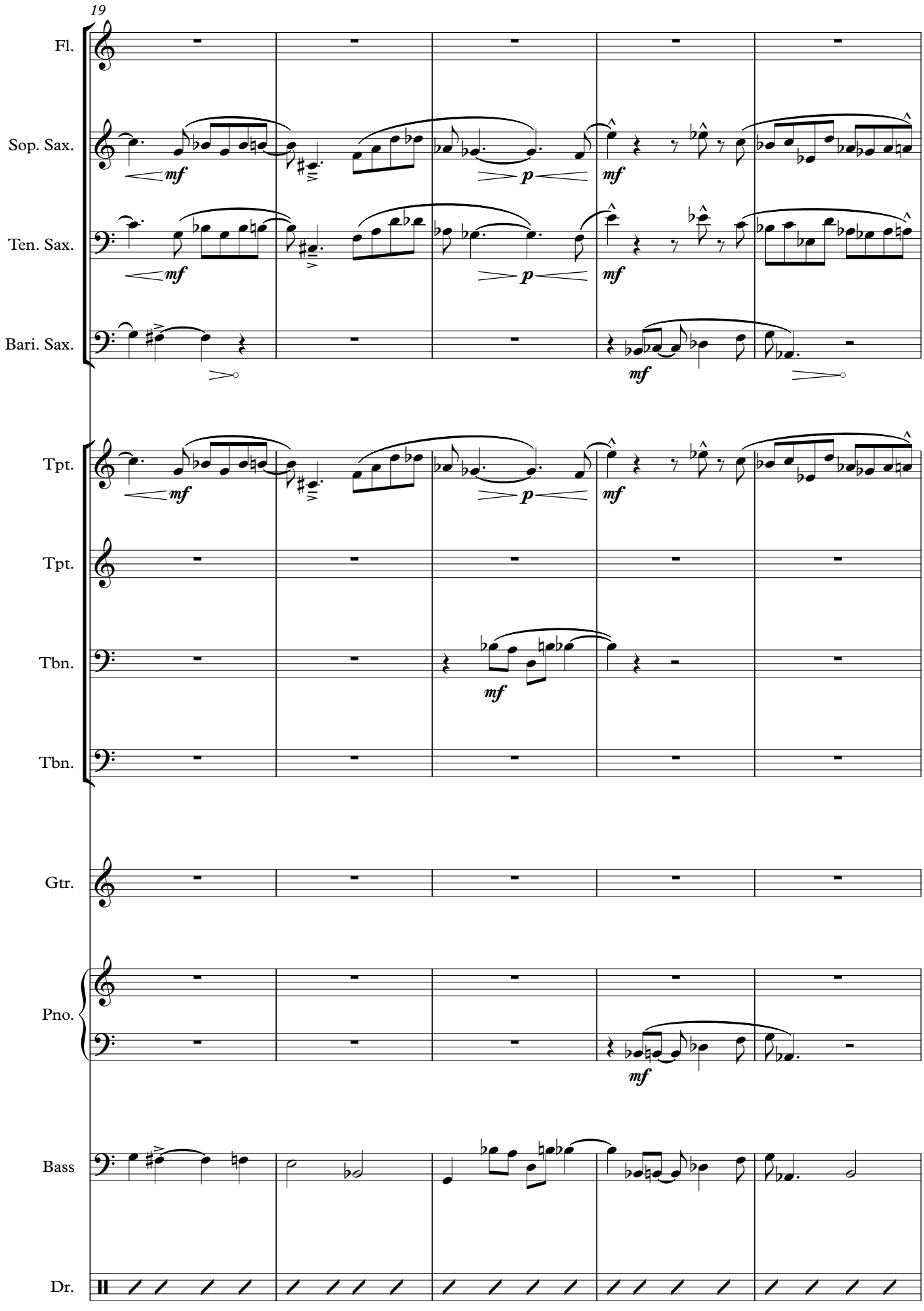
Nuno Trocado

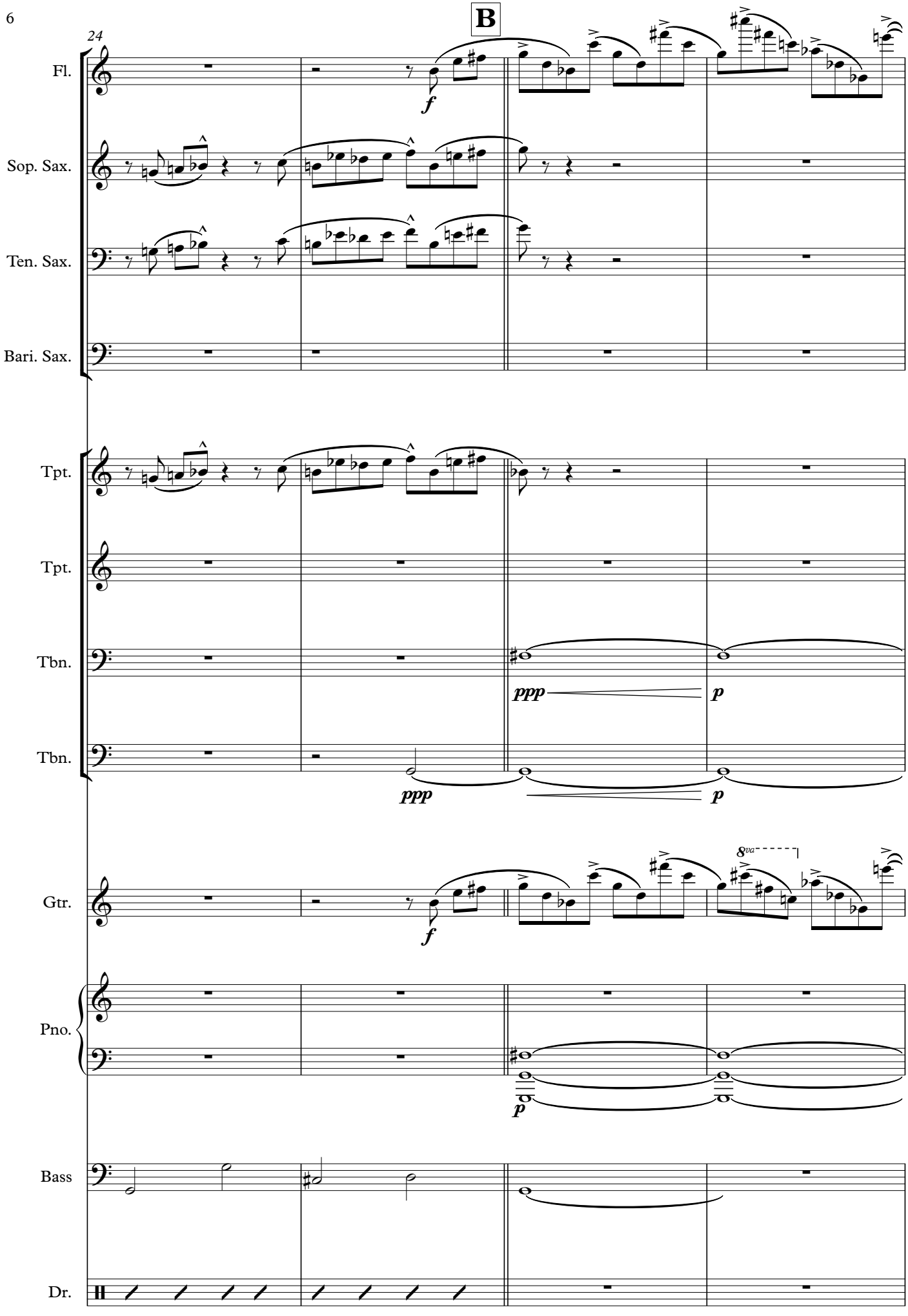


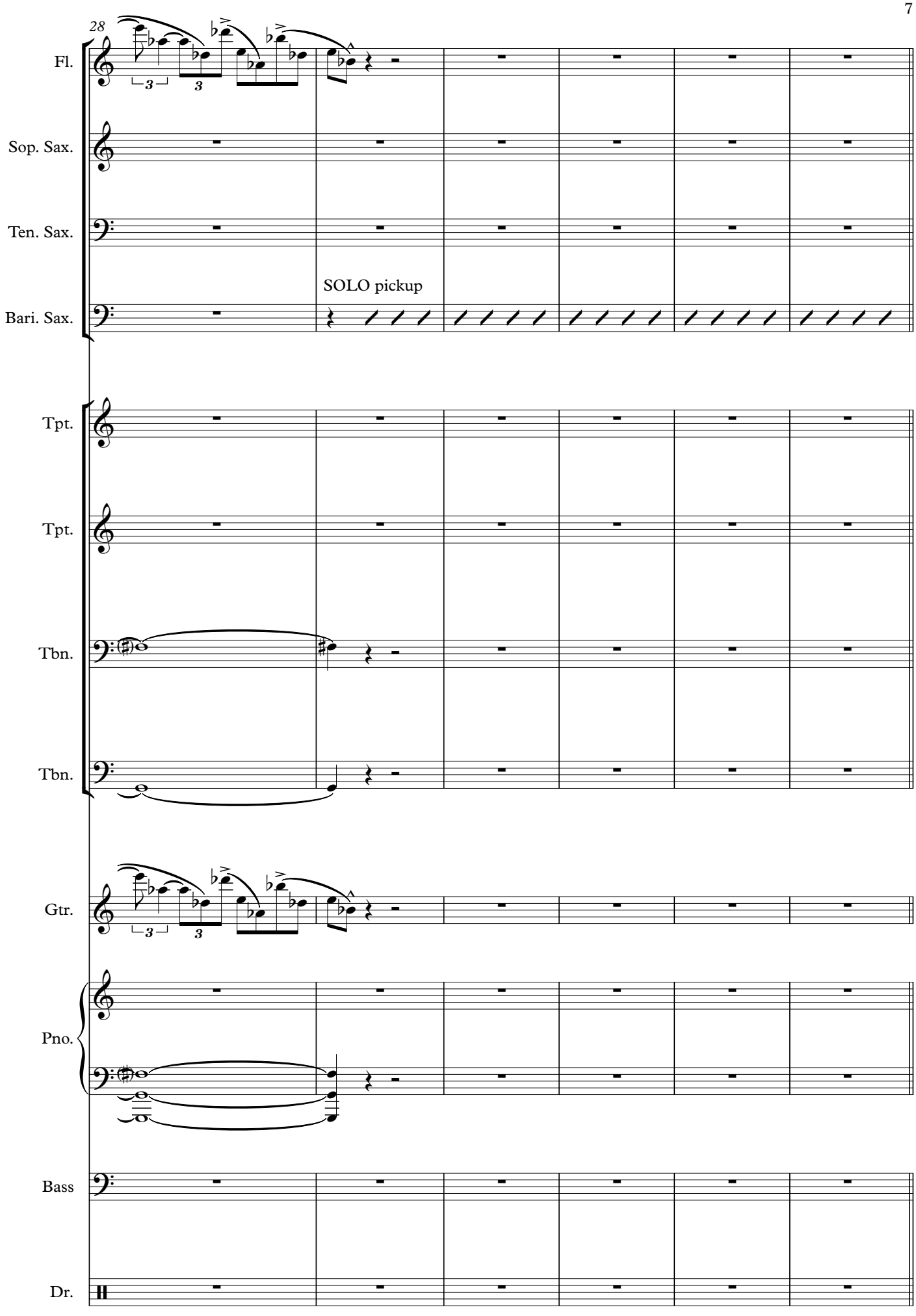

5
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
pp
f
mf
p

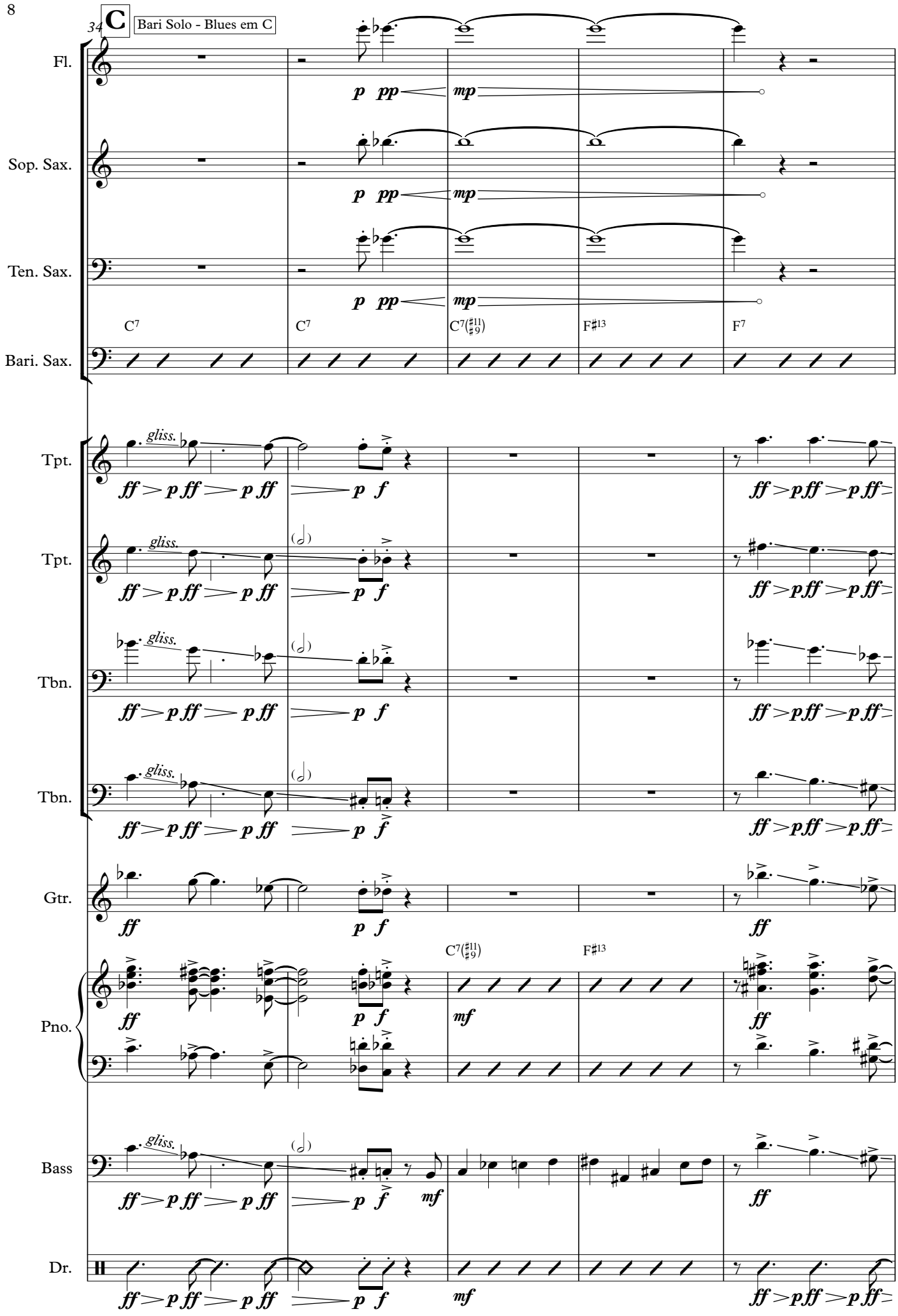

10
A
Fl.
Sop. Sax.
mf
Ten. Sax.
mf
Bari. Sax.
Tpt.
mf
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
f
Bass
Dr.

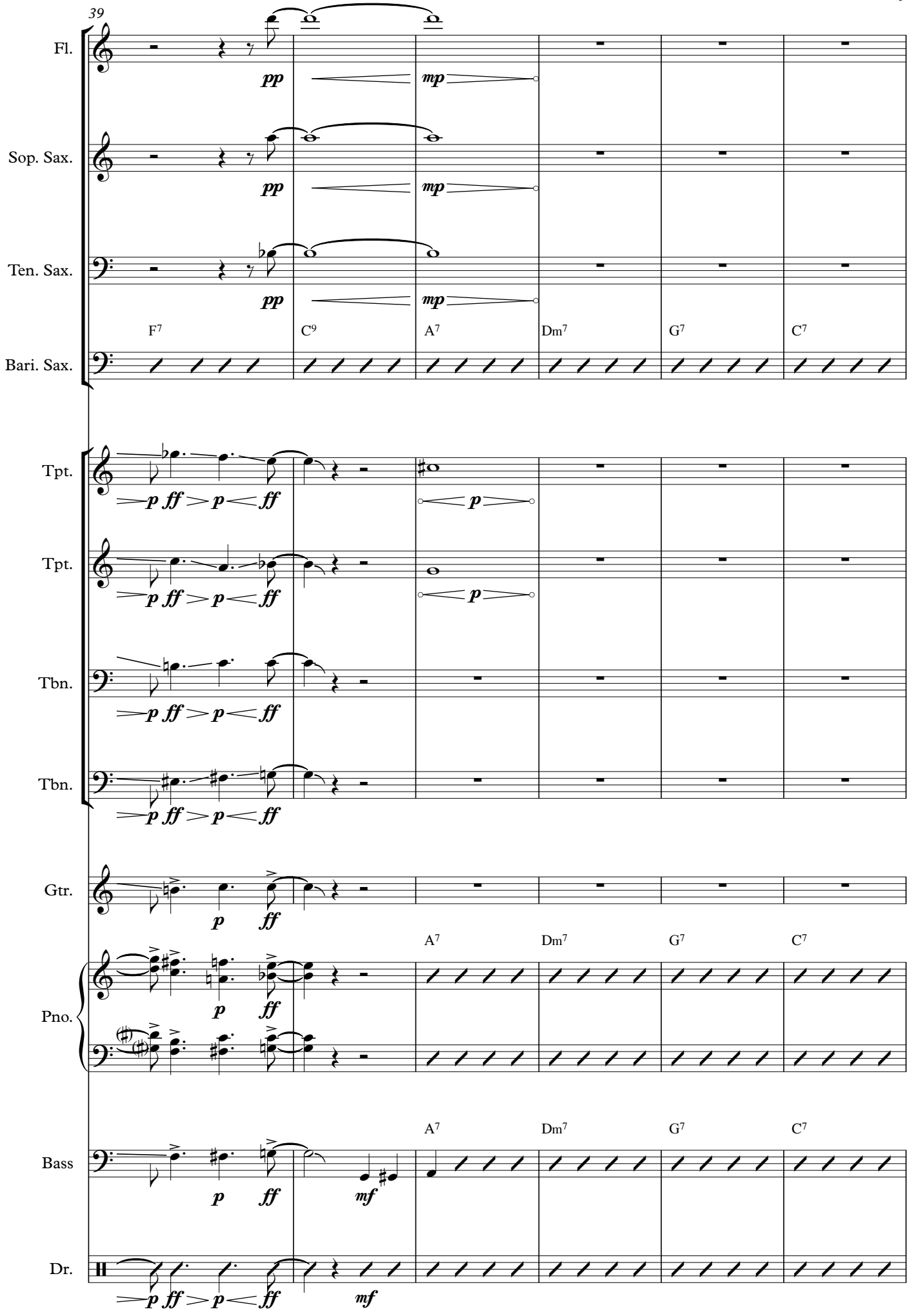

15
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
pp
mp
f
p
mf
ff
8va
Ped.

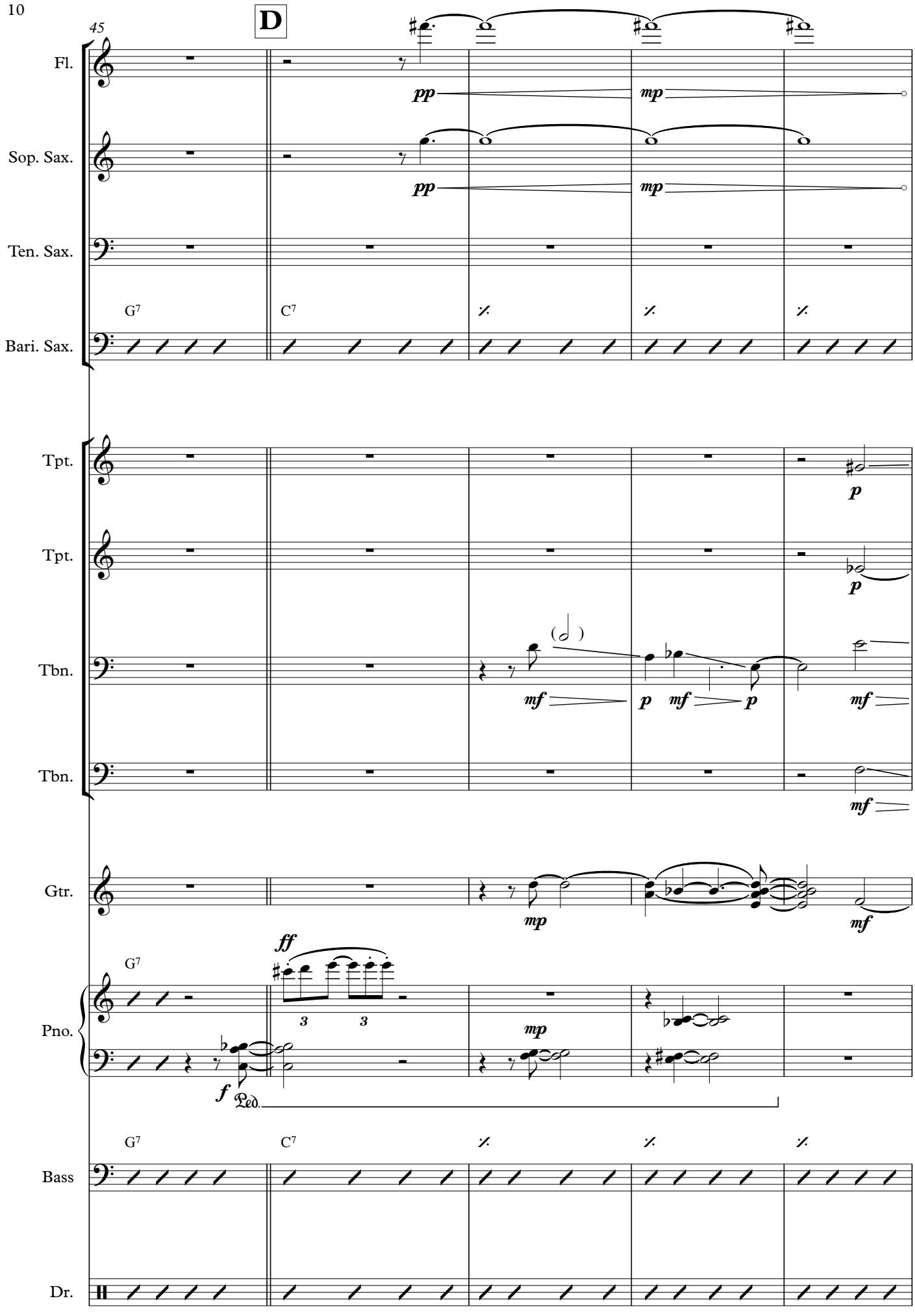

19
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
mf
p
mf

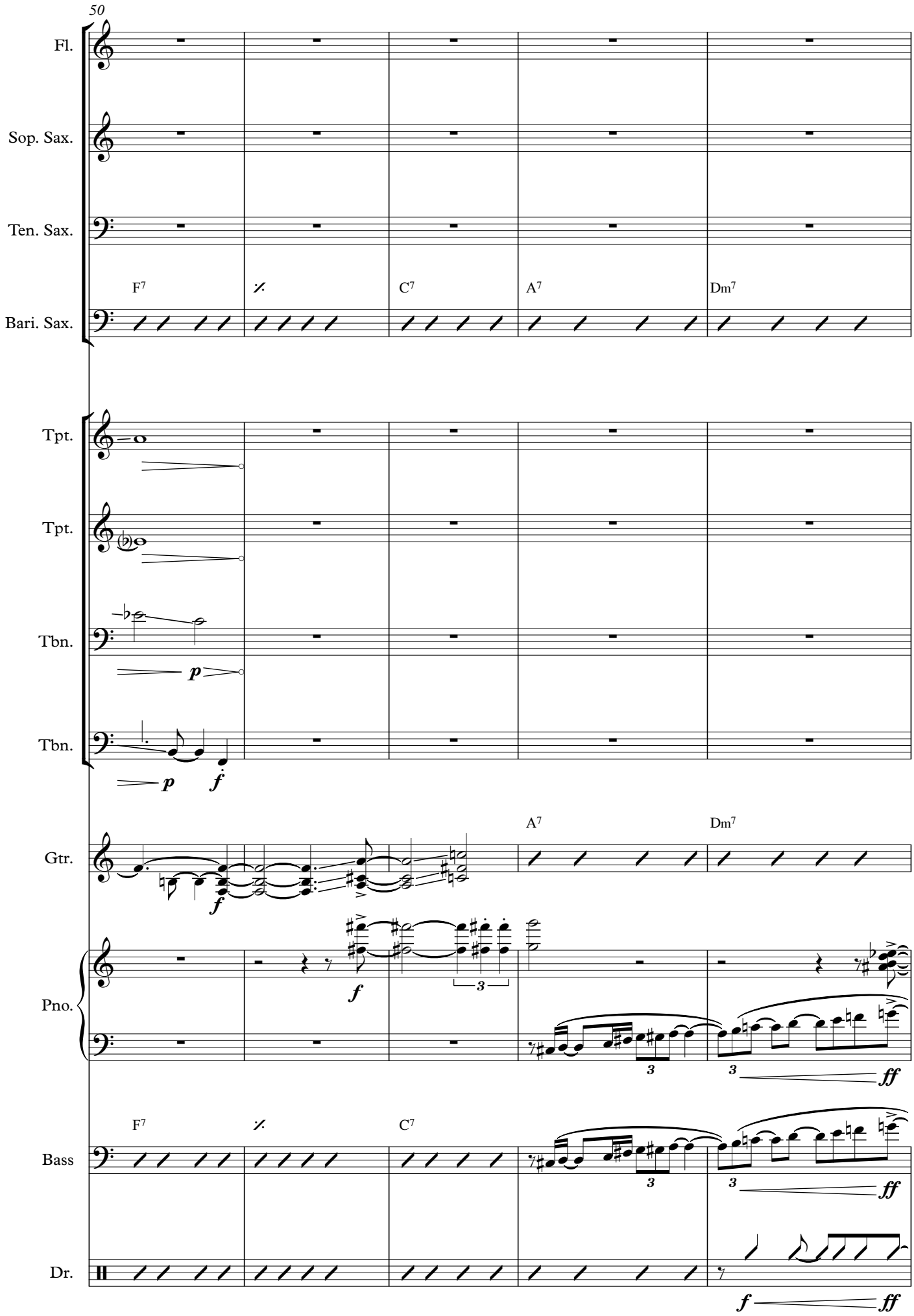

B
24
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
f
ppp
p
8va

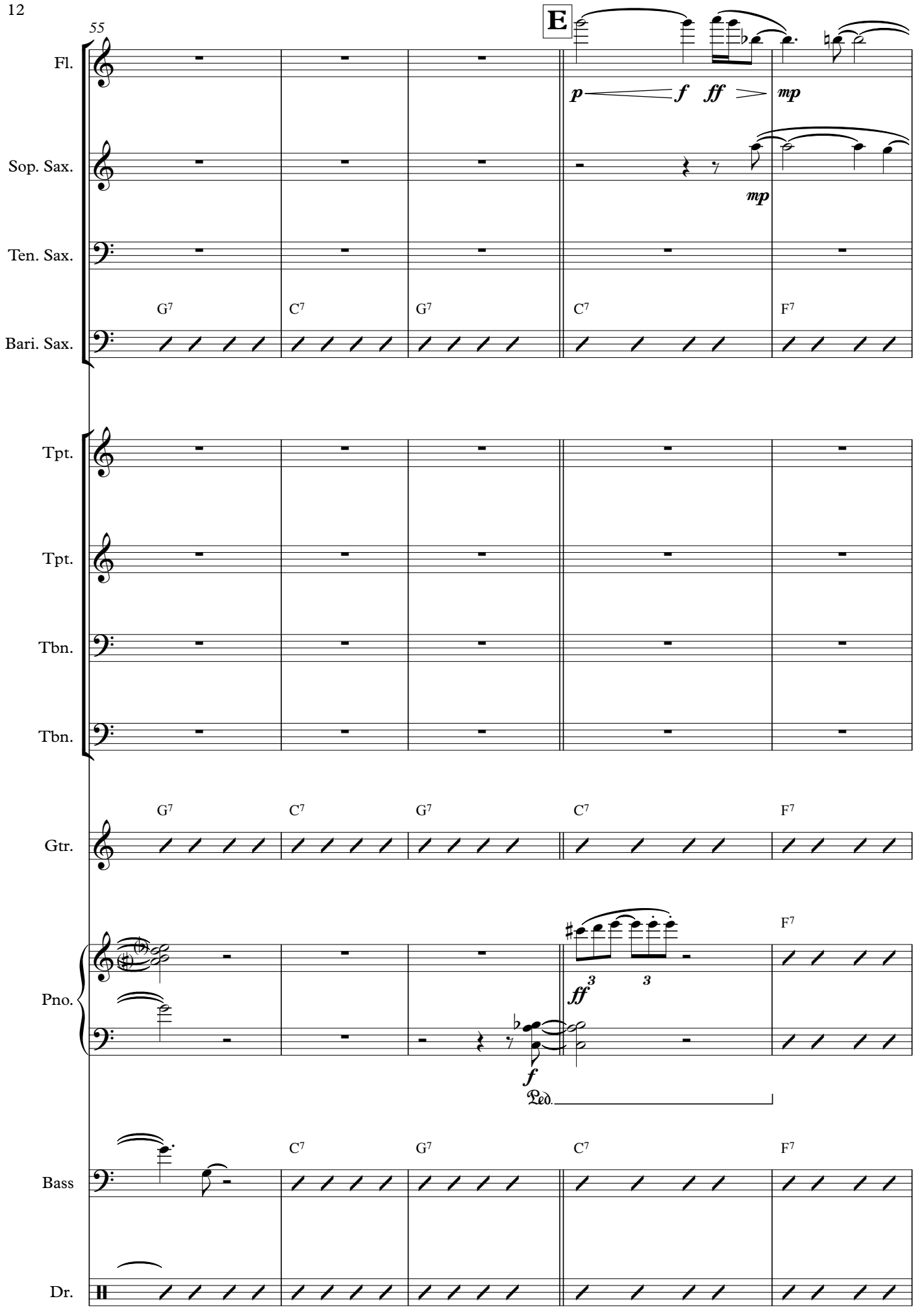

28
Fl.
Sop. Sax.
Ten. Sax.
SOLO pickup
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.

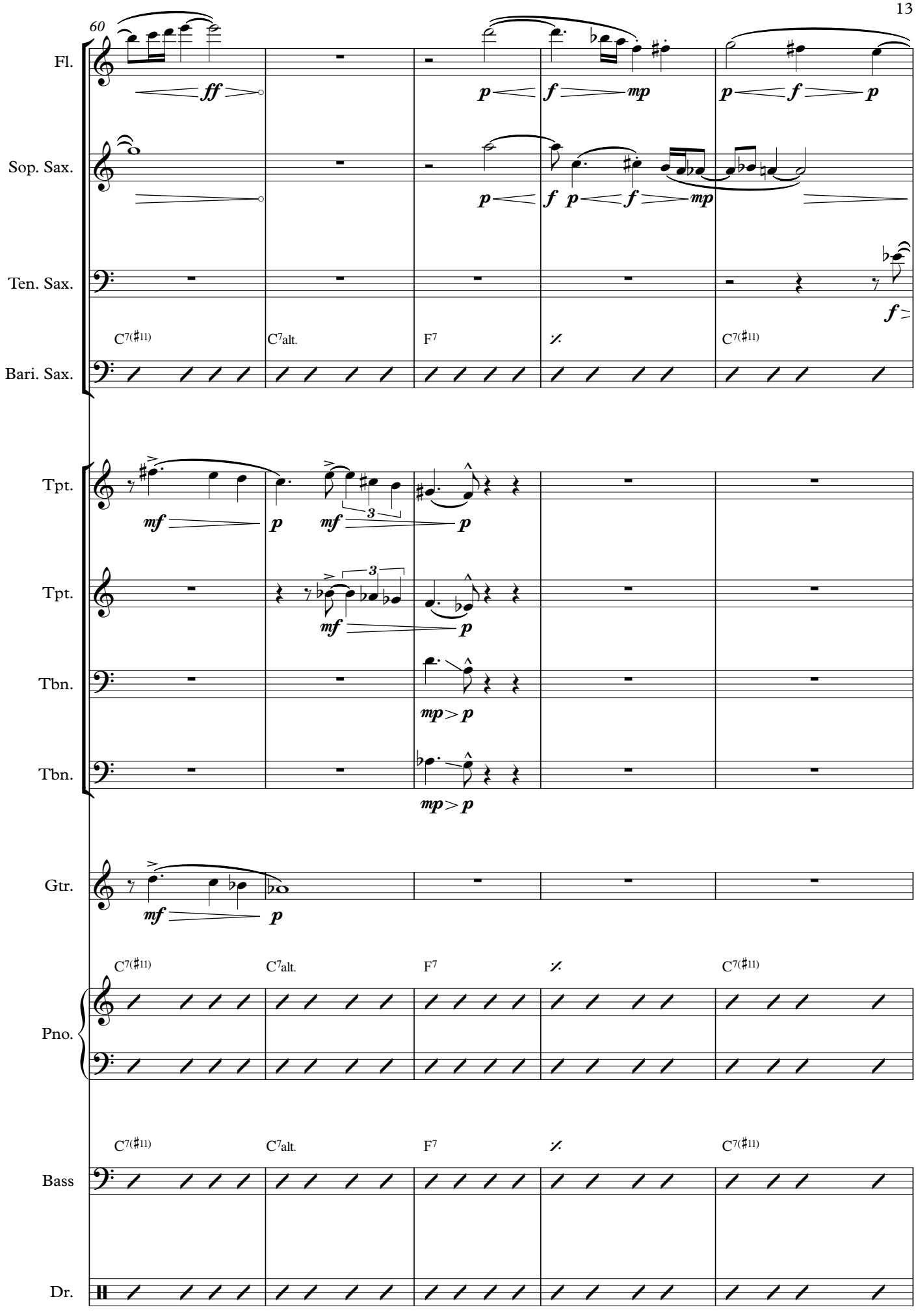

34
C
Bari Solo - Blues em C
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
C7
C7
C7(♯11 ♯9)
F♯13
F7
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
gliss.
p pp
mp
ff > p ff > p ff
p f
mf

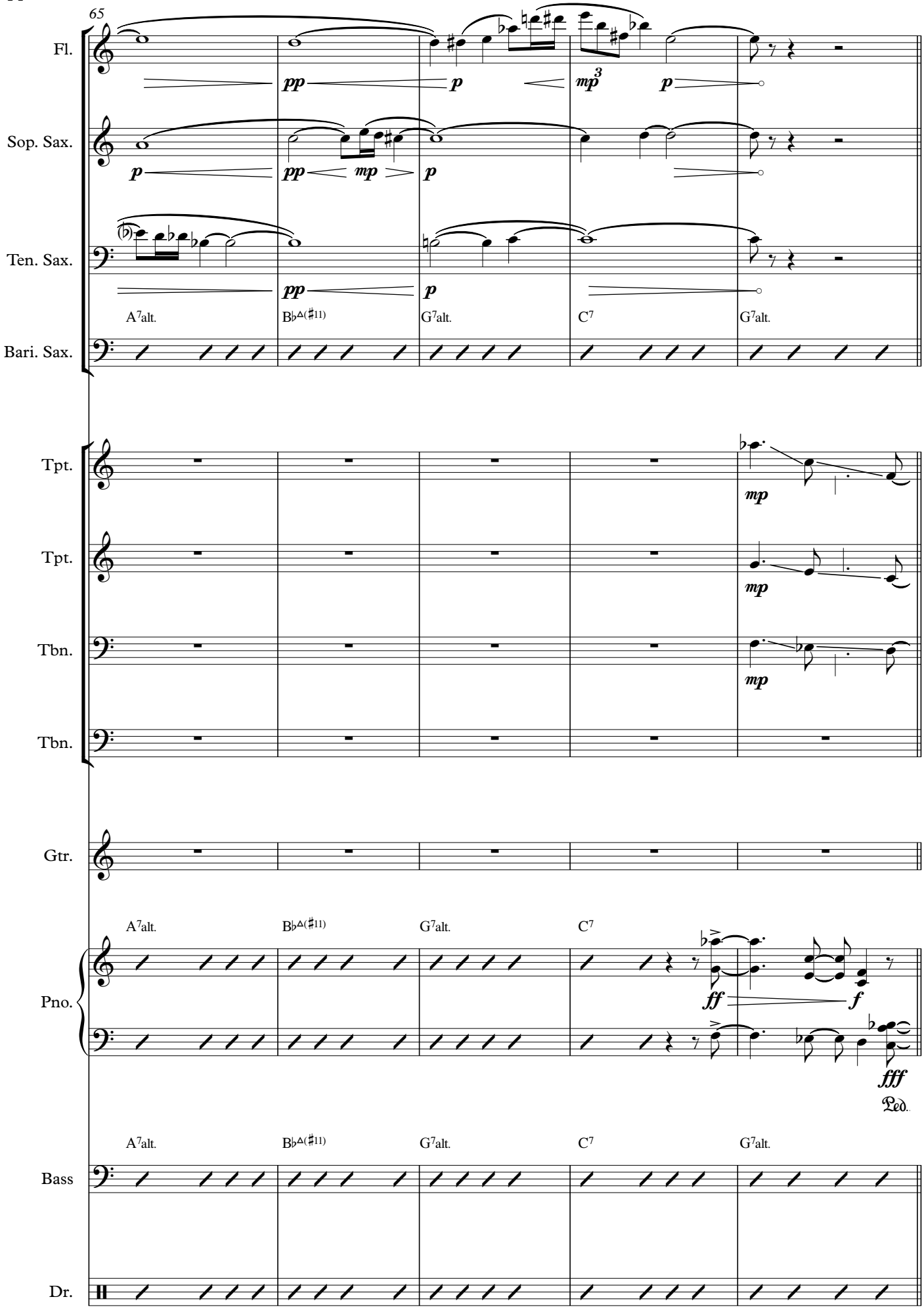

39
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
pp
mp
F7
C9
A7
Dm7
G7
C7
p ff p ff
mf

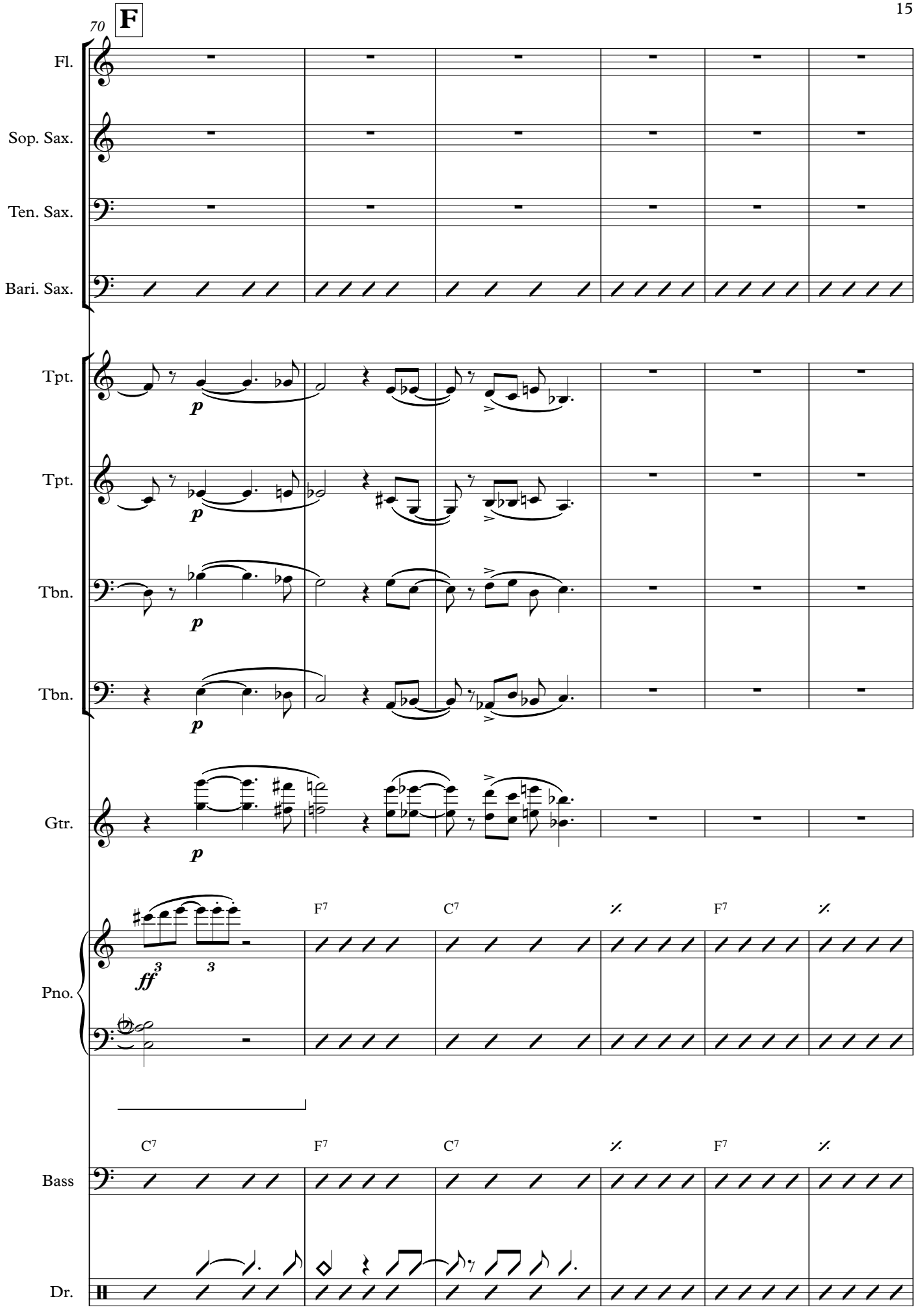

D
45
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
G7
C7
pp
mp
p
mf
ff
f
Ped.
3

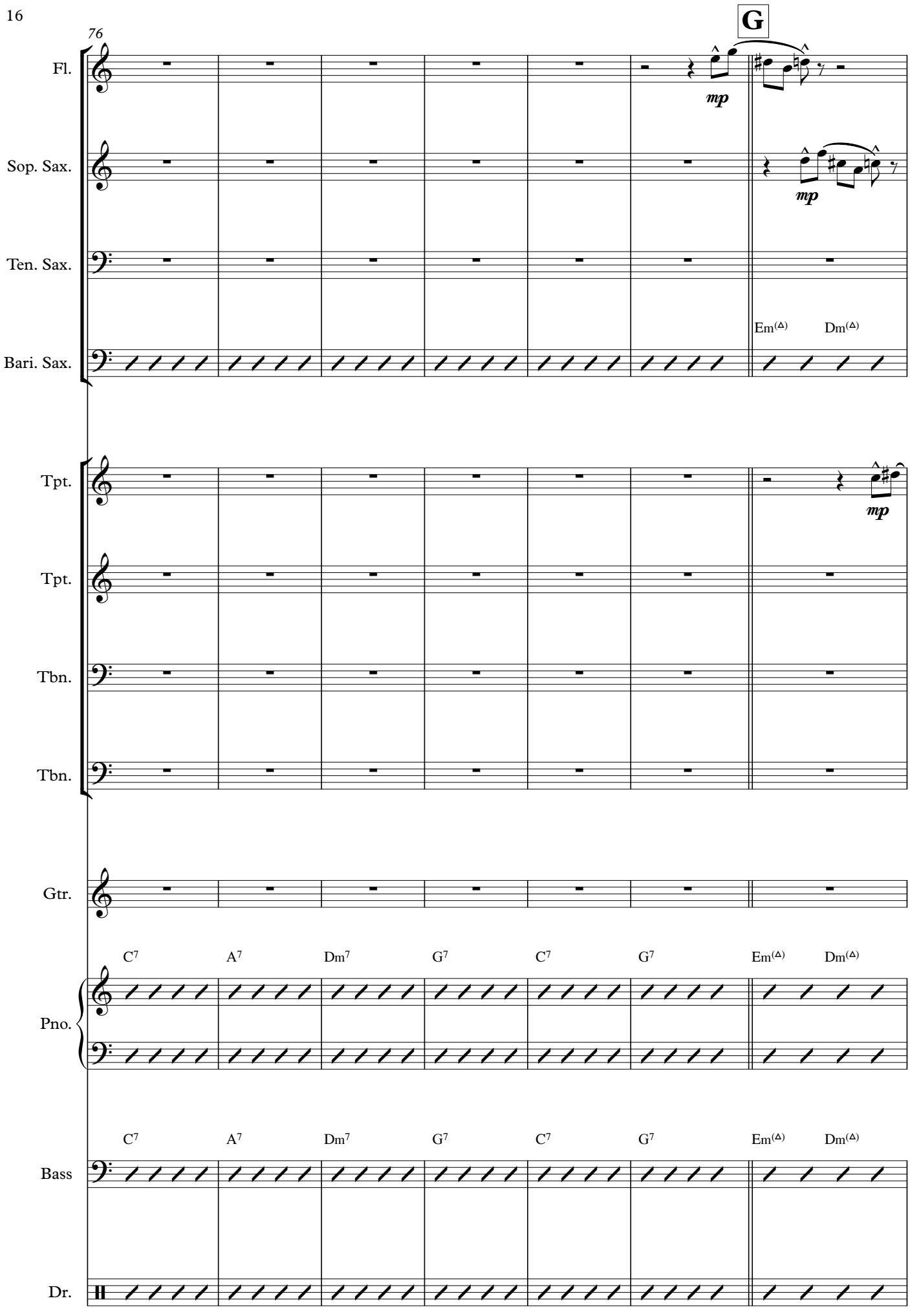

50
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
F7
C7
A7
Dm7
Tpt.
Tpt.
Tbn.
Tbn.
p
f
Gtr.
Pno.
3
ff
Bass
Dr.

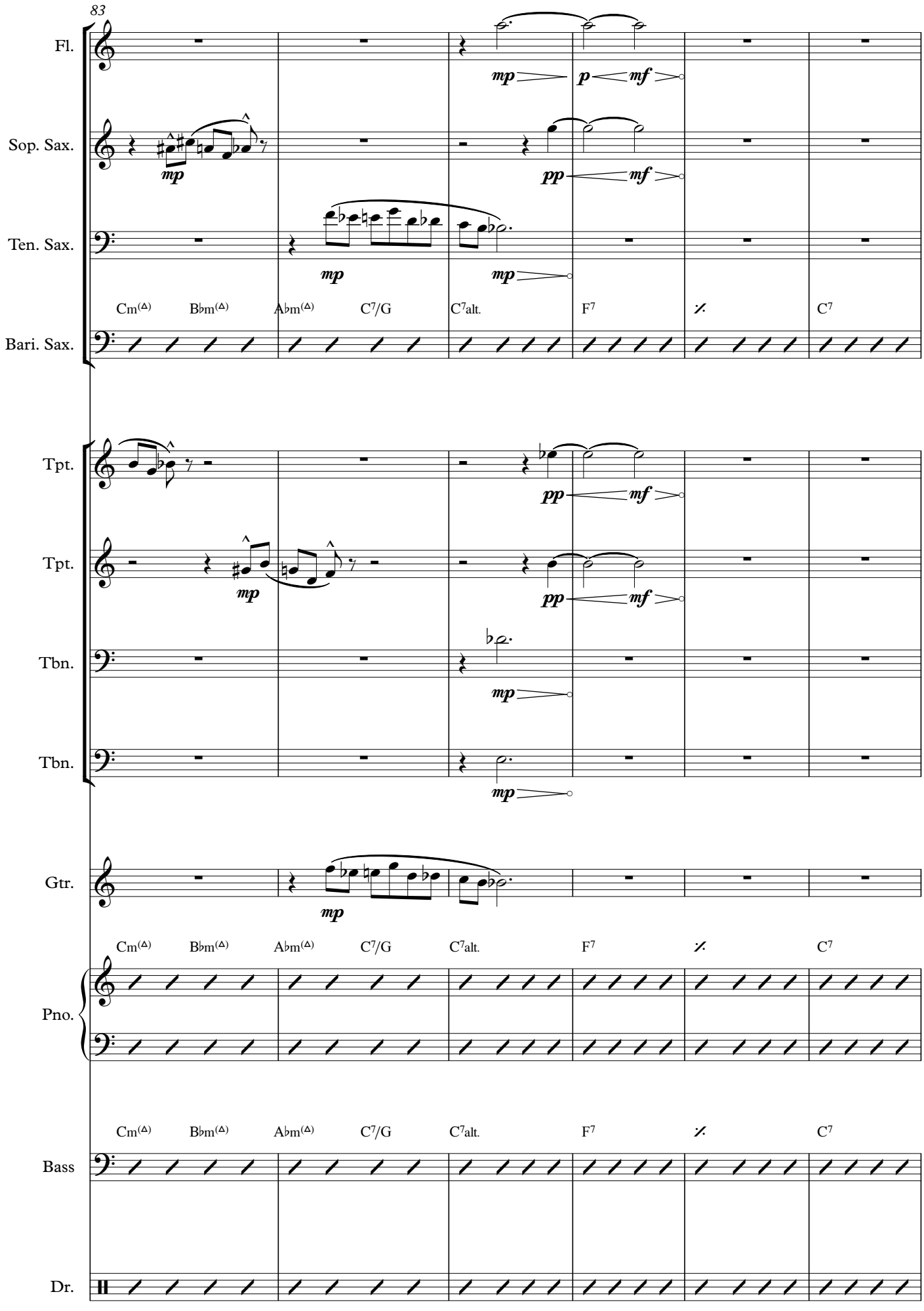

55
E
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
G7
C7
F7
p
f
ff
mp
Ped.

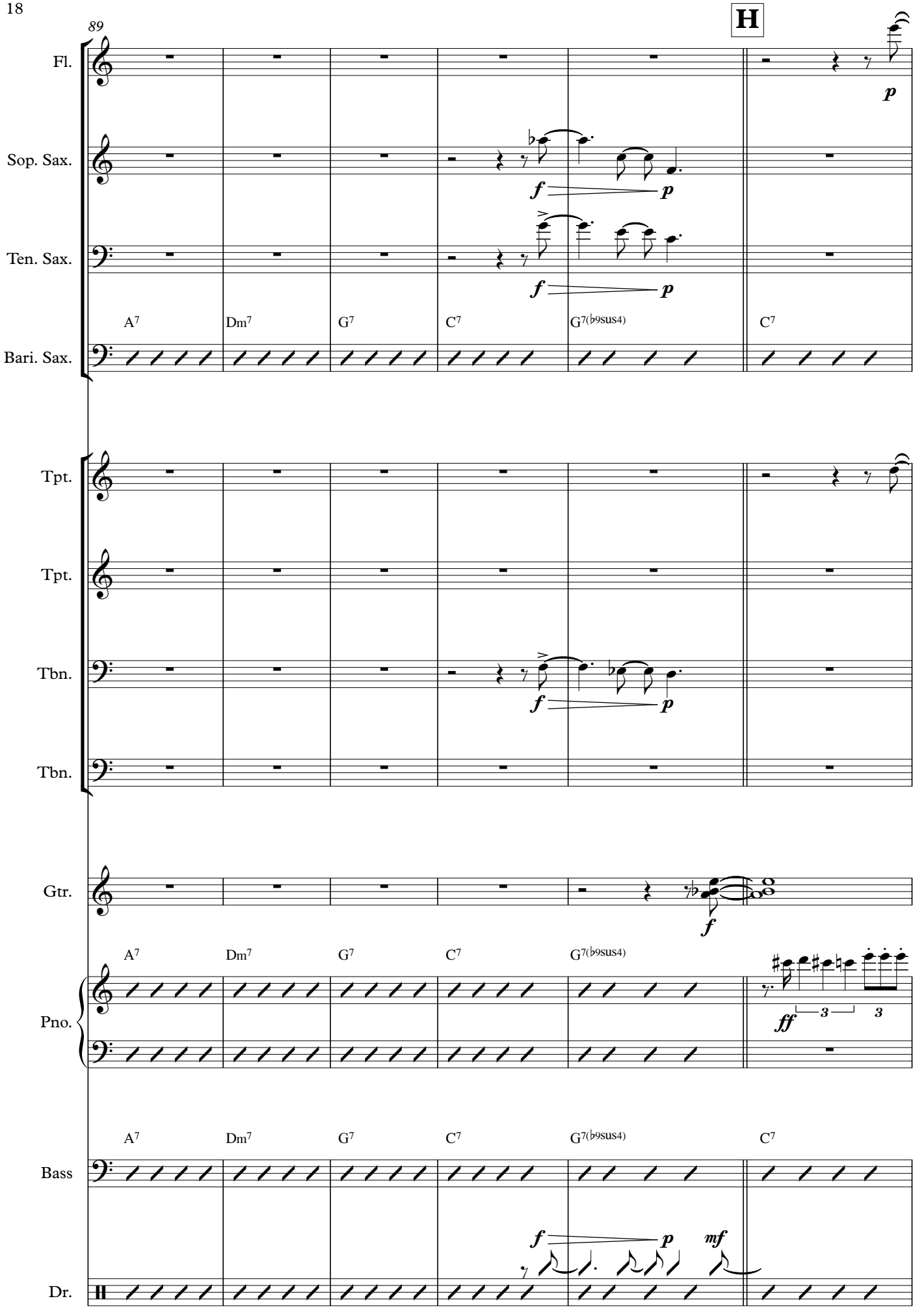

60
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
C7(♯11)
C7alt.
F7
C7(♯11)

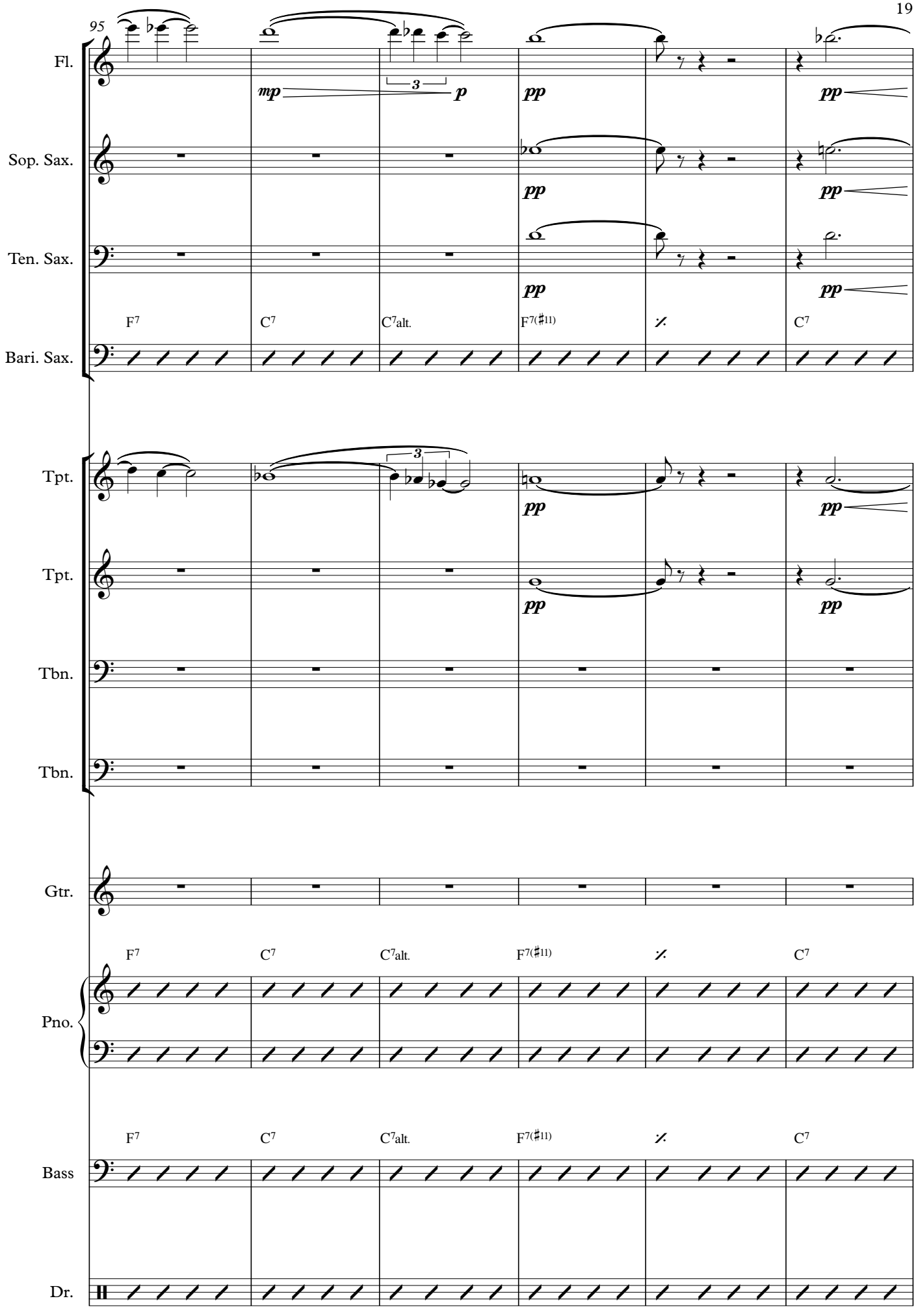

65
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
A7alt.
B♭△(♯11)
G7alt.
C7
G7alt.
pp
p
mp
mp3
ff
f
fff
Ped.

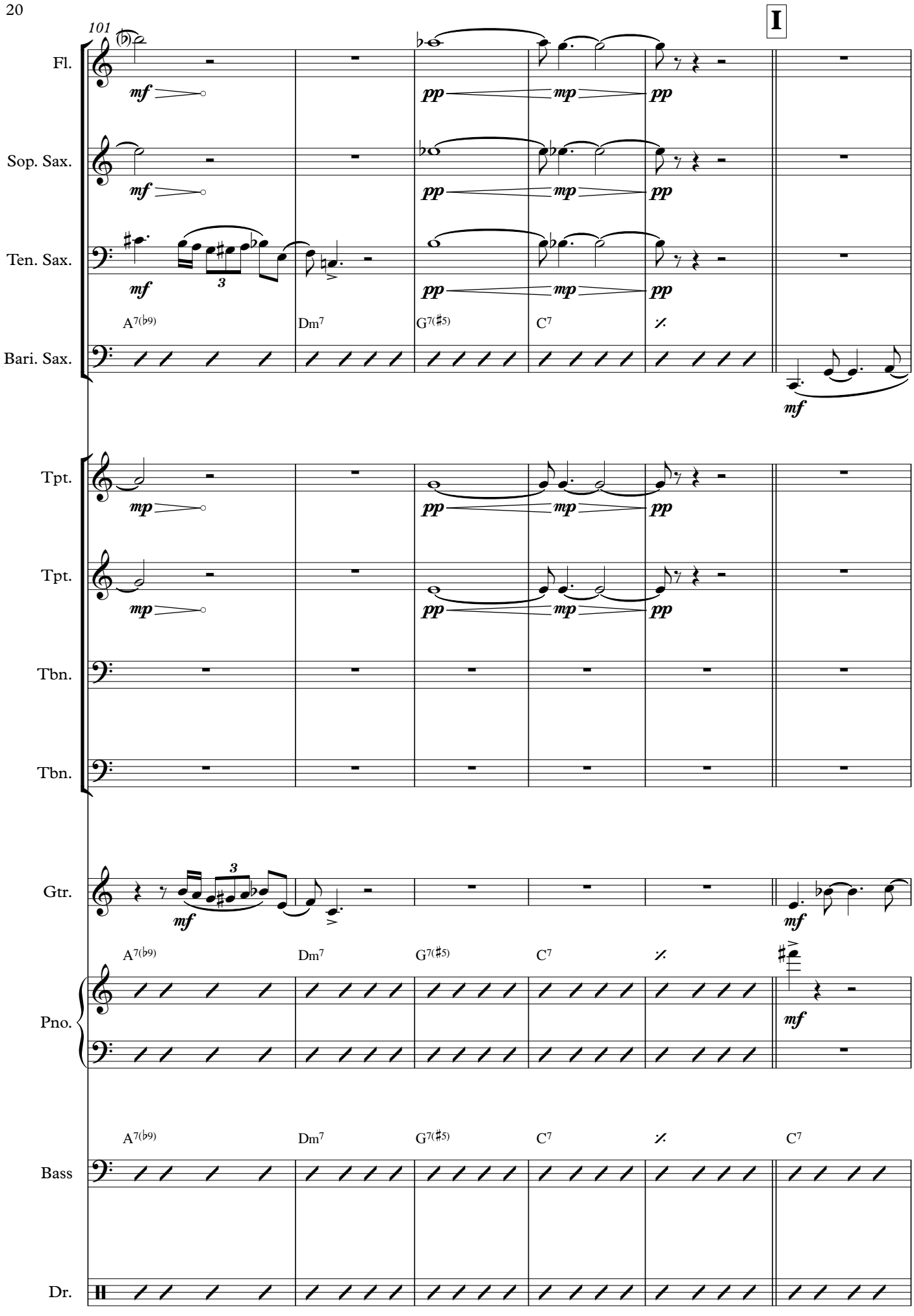

70
F
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
p
ff
F7
C7
C7
F7
C7
F7

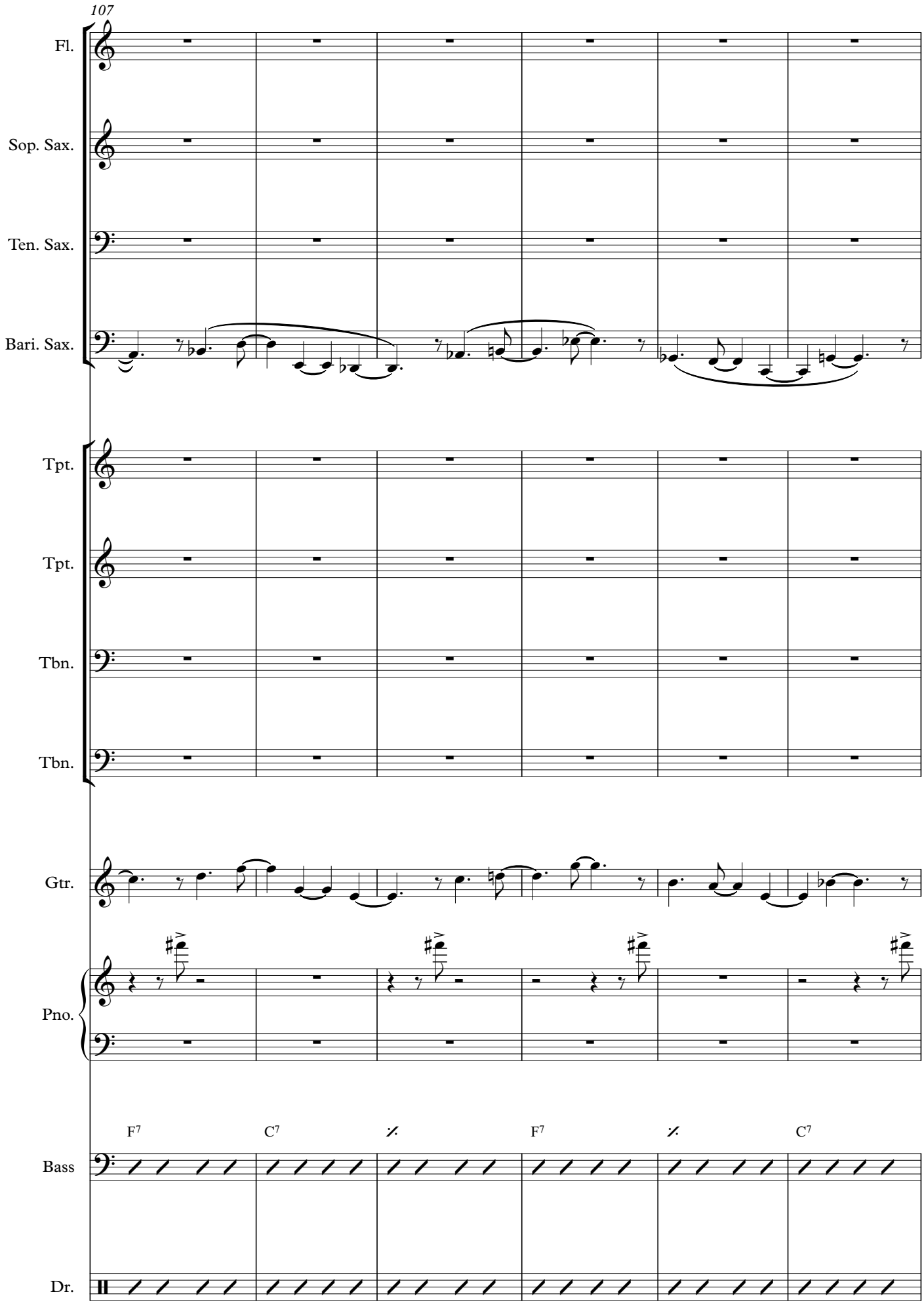

76
G
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
mp
Em(△) Dm(△)
C7 A7 Dm7 G7 C7 G7 Em(△) Dm(△)

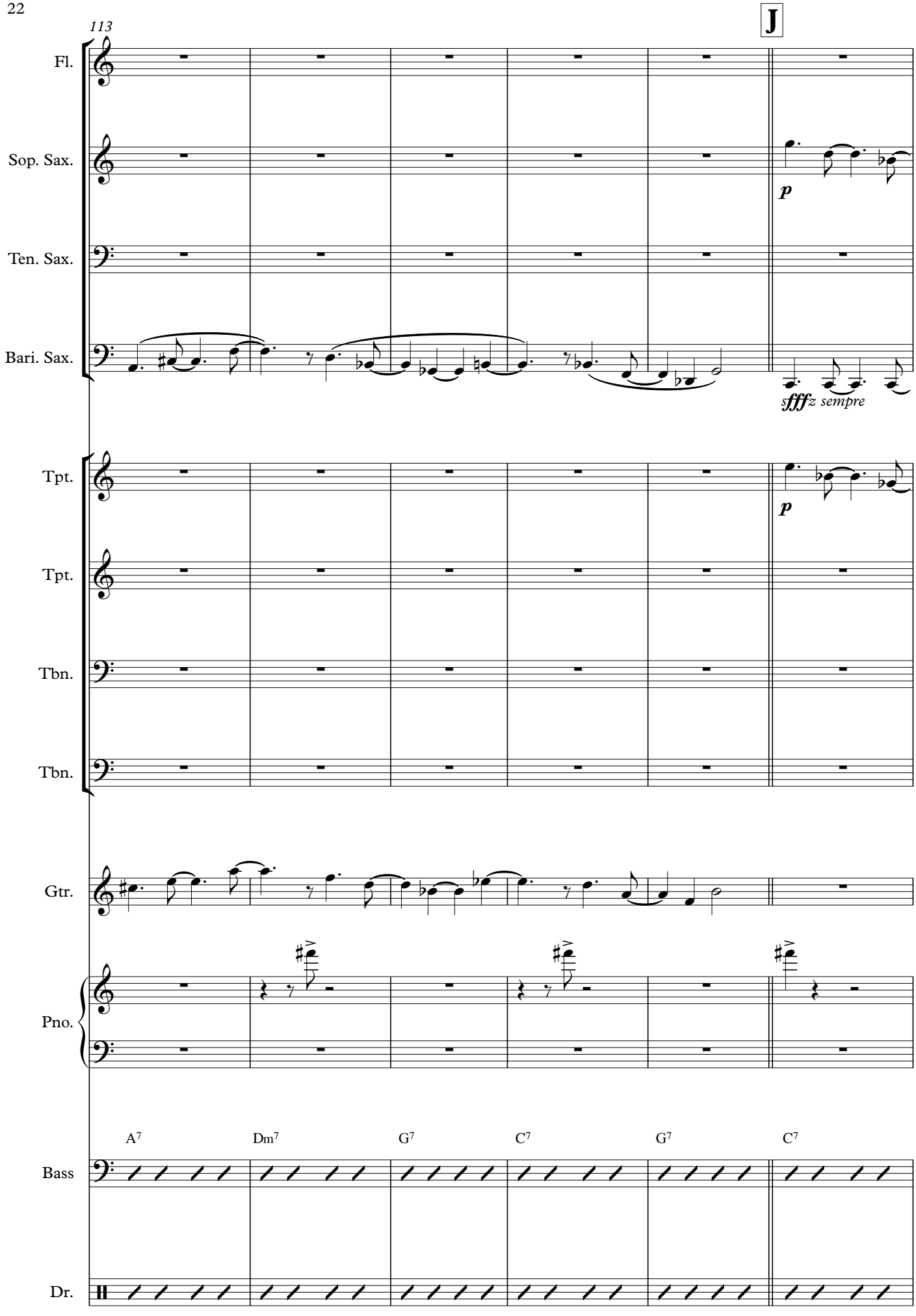

83
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
mp
p
mf
pp
Cm(Δ) B♭m(Δ) A♭m(Δ) C7/G C7alt. F7 C7

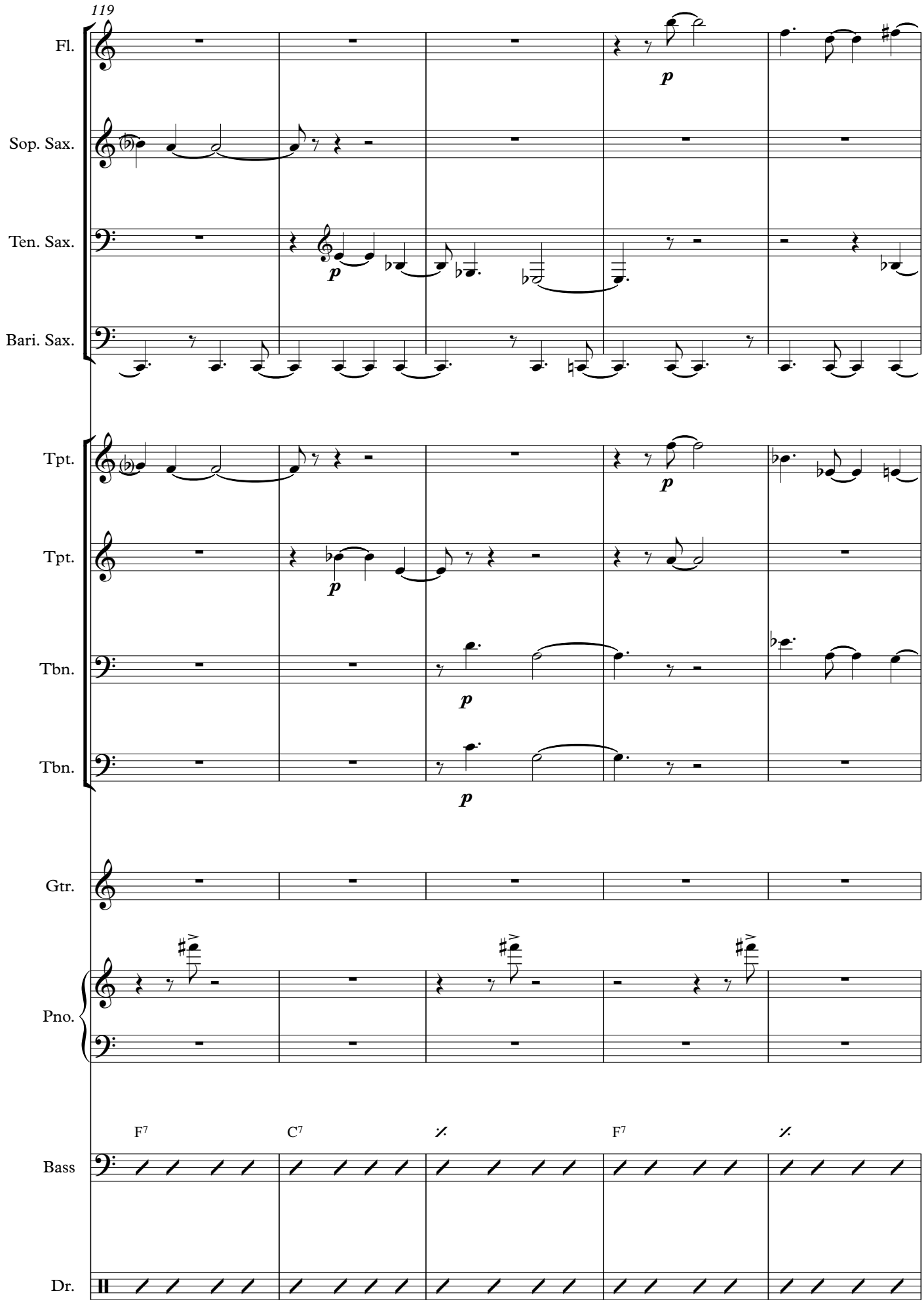

89
H
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
A7
Dm7
G7
C7
G7(♭9sus4)
C7
p
f
ff
mf
3

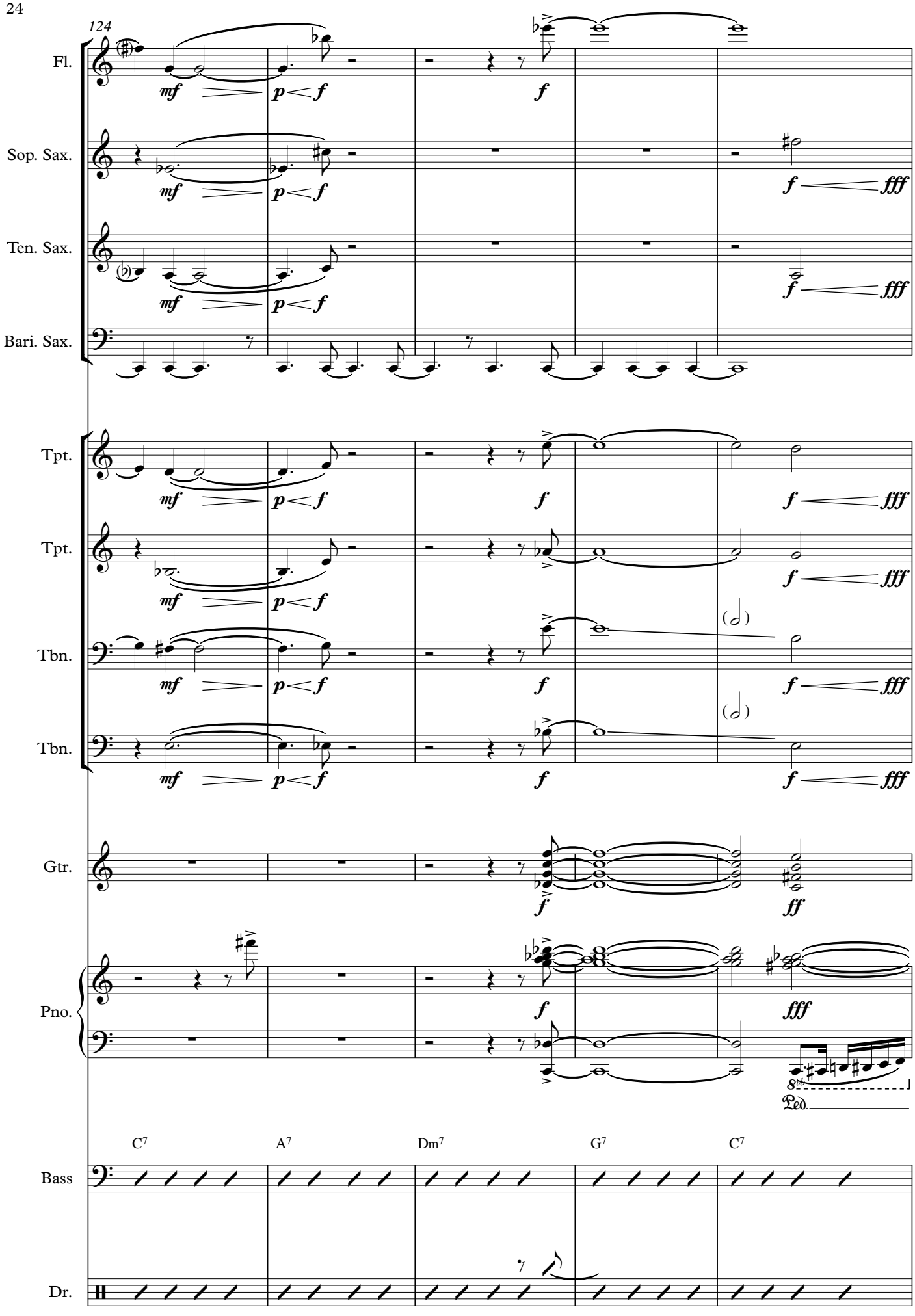

95
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
mp
p
pp
3
F7
C7
C7alt.
F7(♯11)
C7


101
I
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
A7(♭9)
Dm7
G7(♯5)
C7
mf
mp
pp


107
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
F7
C7
F7
C7
Bass
Dr.


113
J
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
p
sffz sempre
Tpt.
Tpt.
Tbn.
Tbn.
p
Gtr.
Pno.
A7
Dm7
G7
C7
G7
C7
Bass
Dr.


119
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
F7
C7
F7
Bass
Dr.


124
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
C7
A7
Dm7
G7
C7
mf
p
f
fff
ff
Ped.

ON CUE

129

rest until piano resonance decays

K

♩ = 120 **Swing**

Fl.

Sop. Sax.

Ten. Sax.

Bari. Sax.

Tpt.

Tpt.

Tbn.

Tbn.

Gtr.

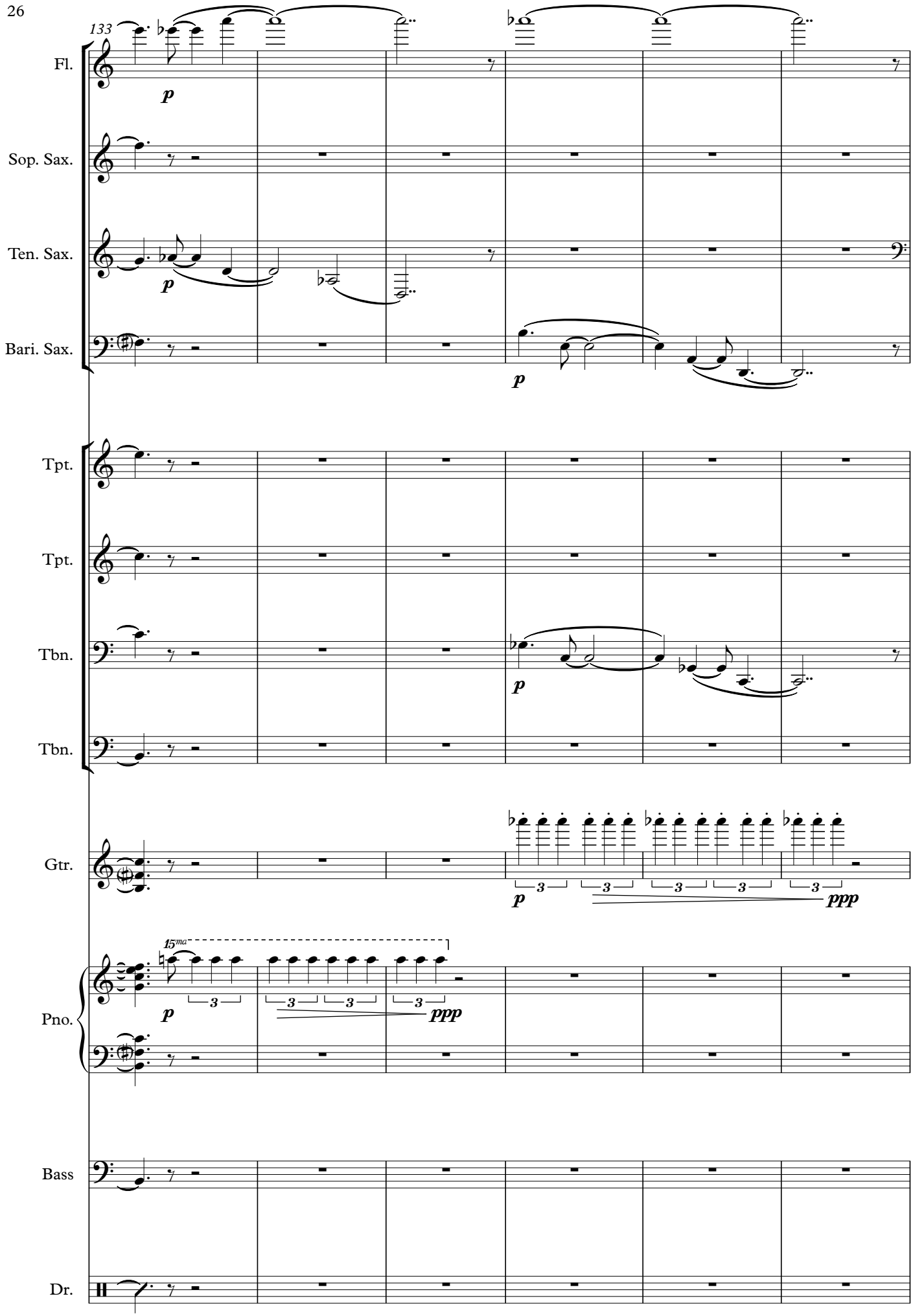
Pno.

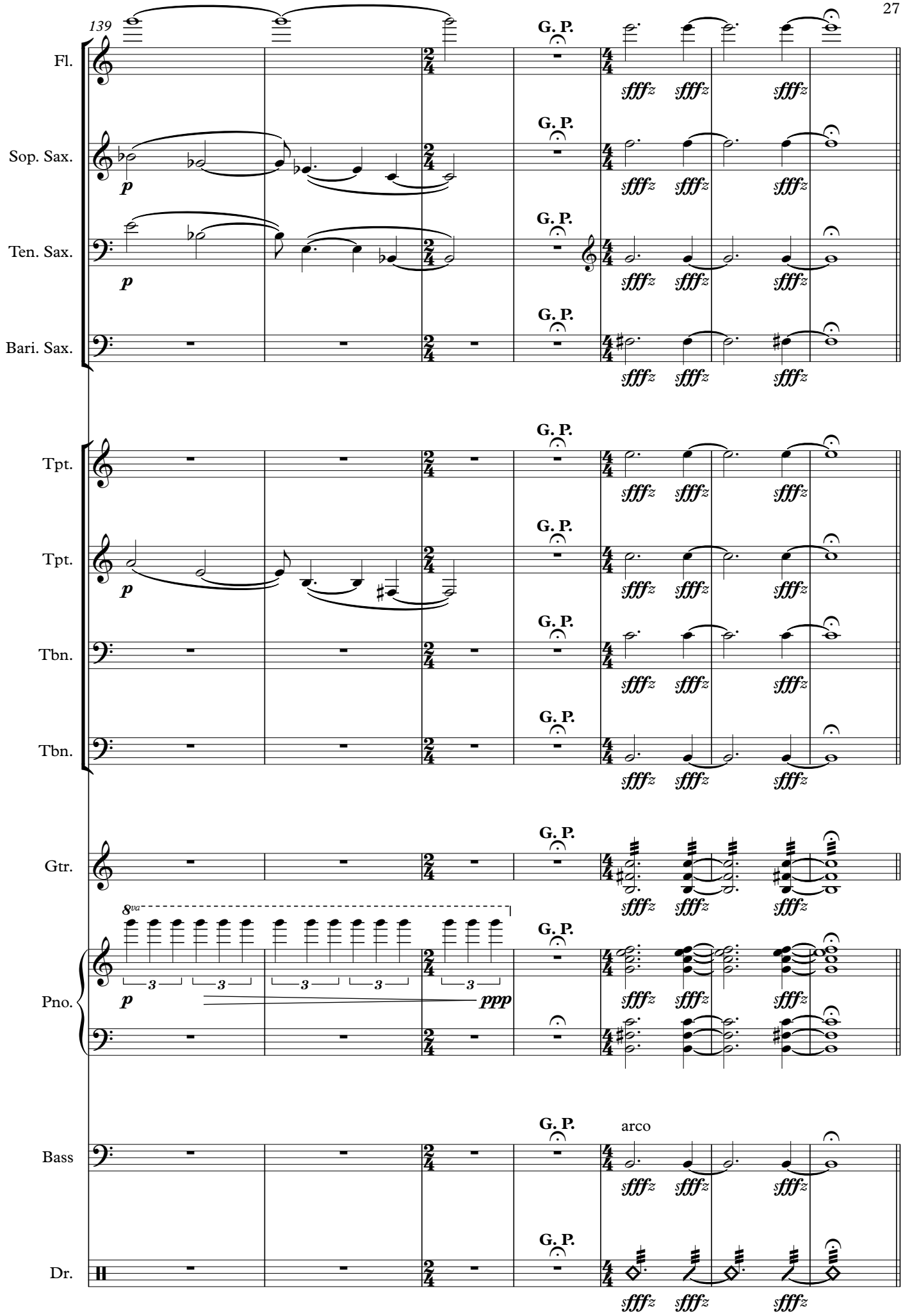
Bass

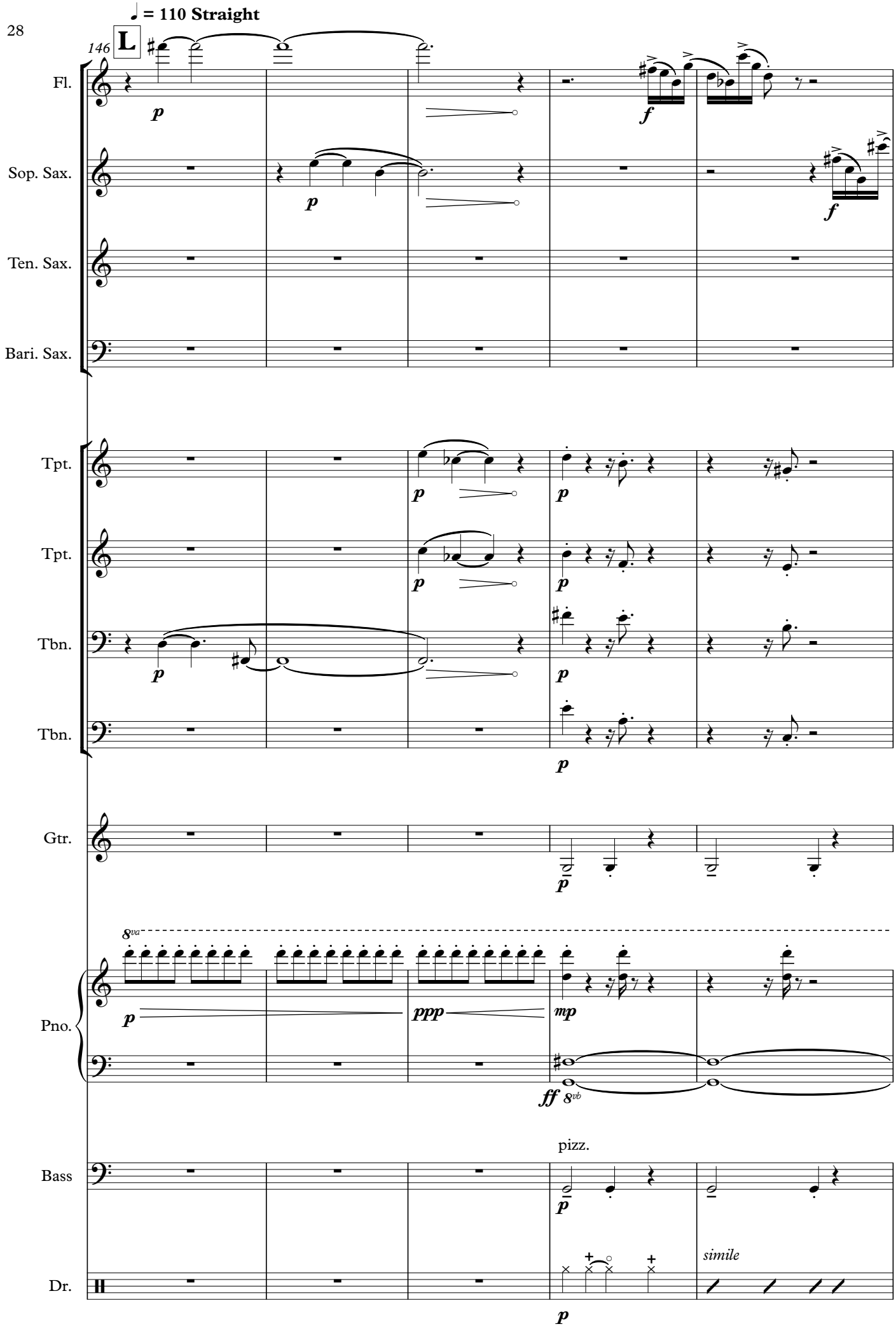
choke cymbals

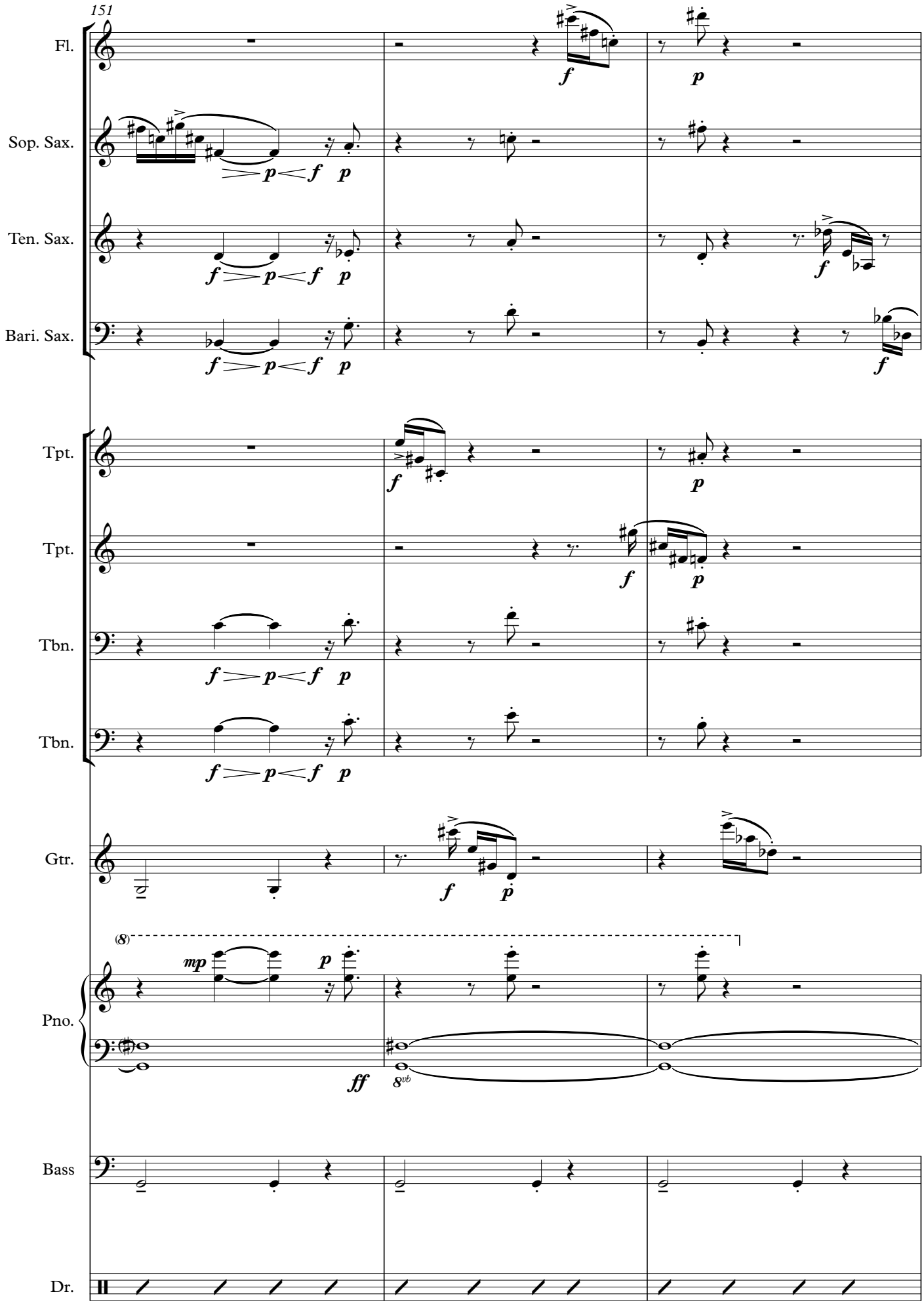
Dr.

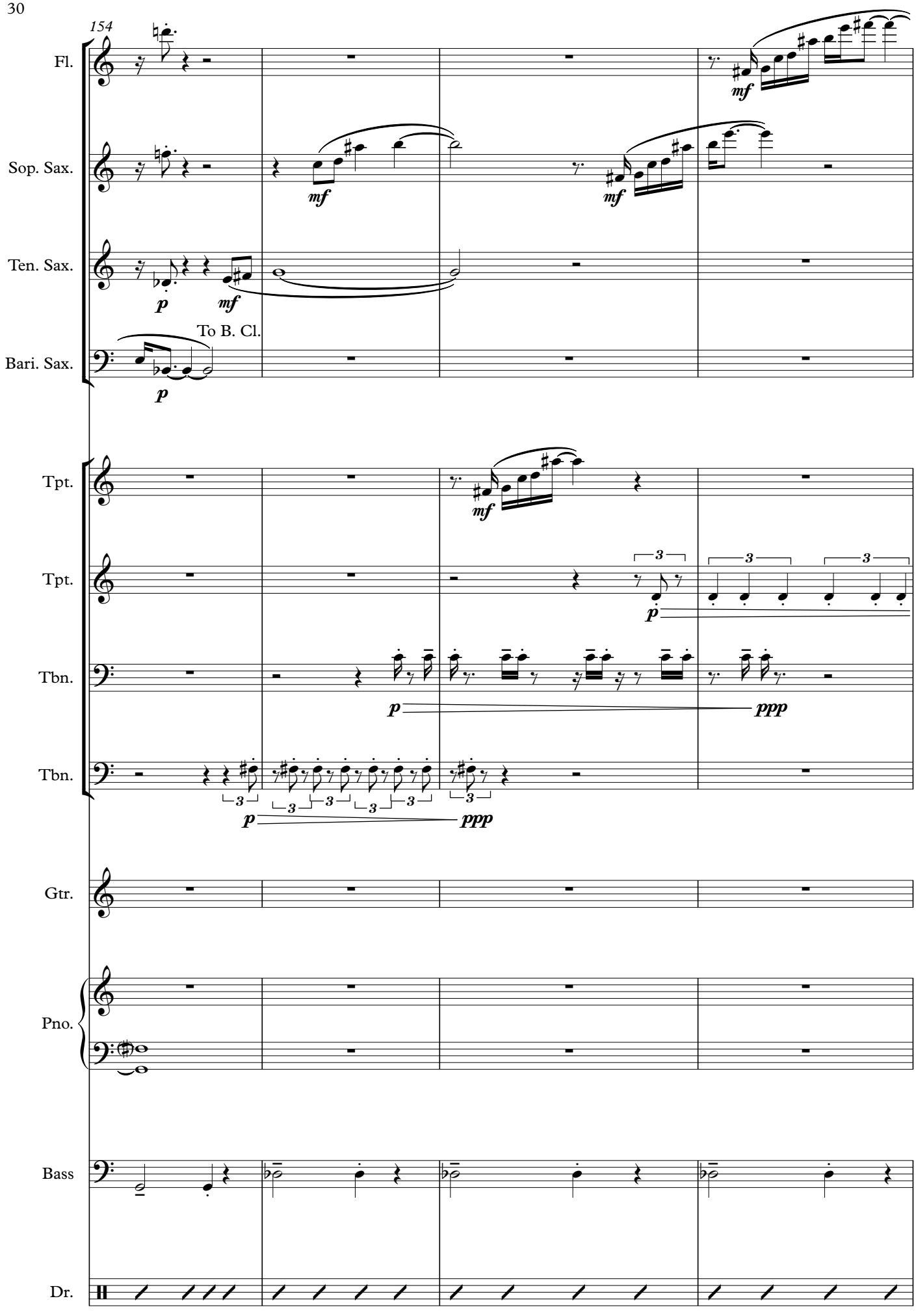
ff

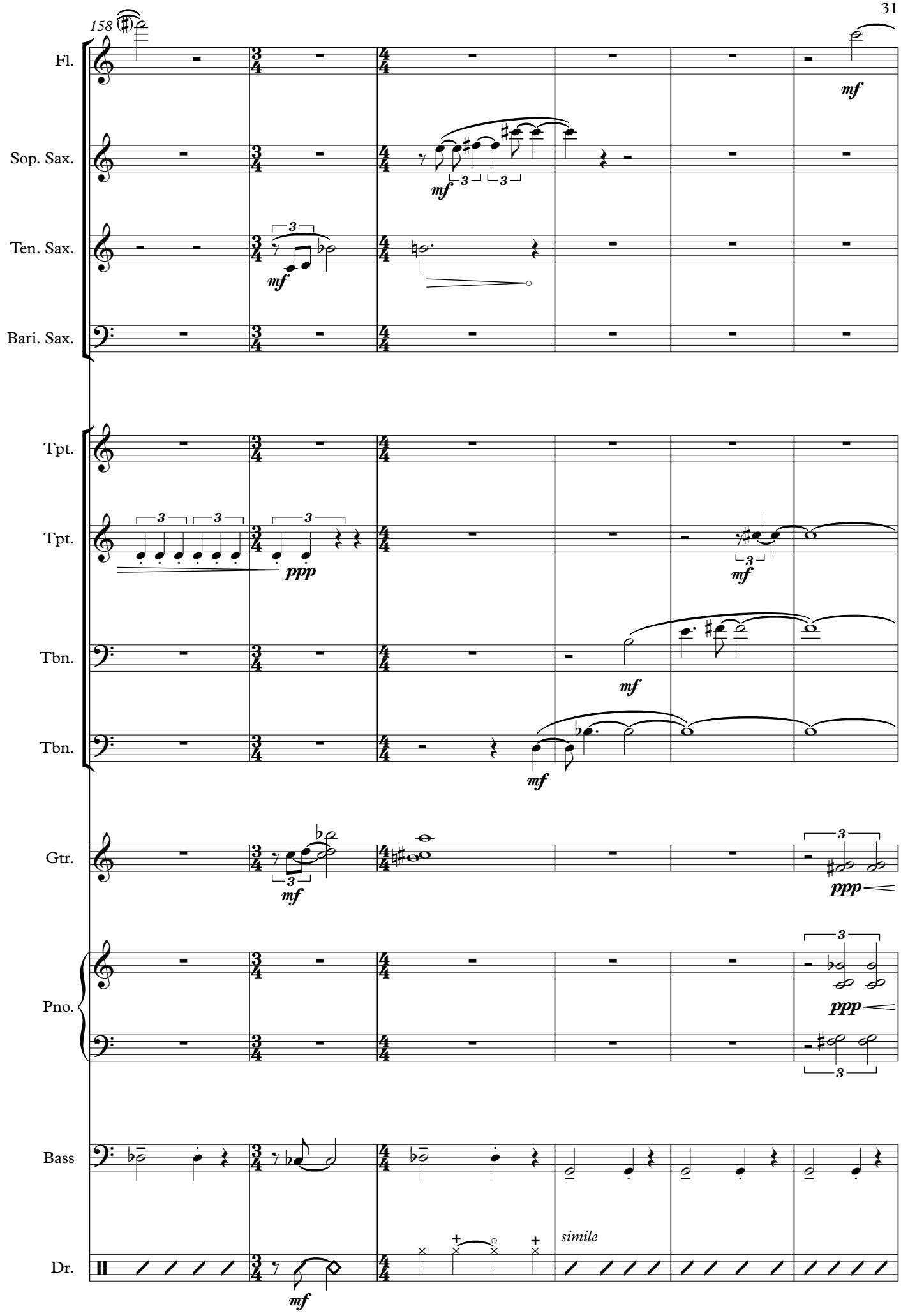

133
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
p
ppp
15ma
3

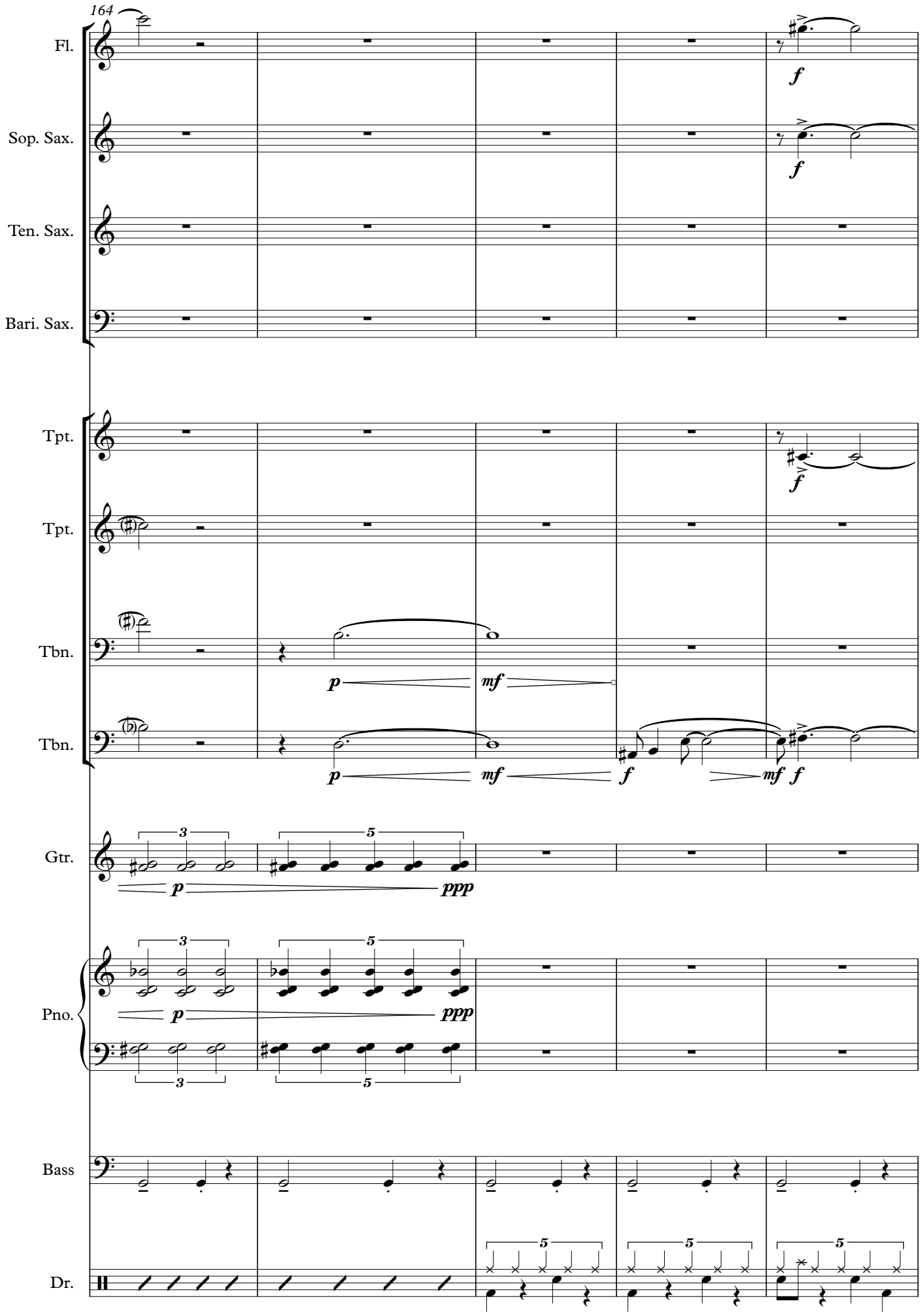

139
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
G. P.
p
ppp
8va
arco
sfffz

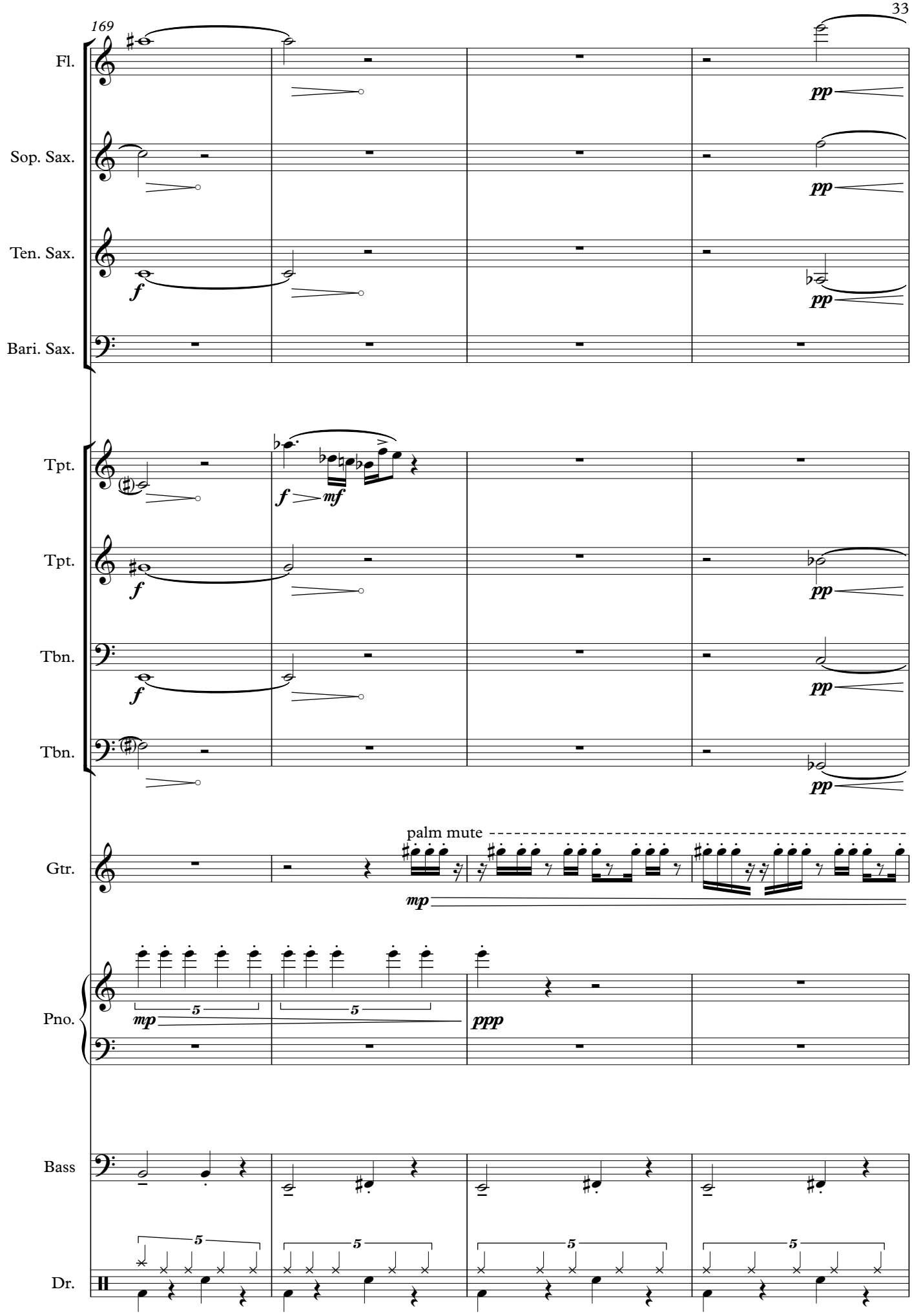

♩ = 110 Straight
146
L
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
8va
8vb
pizz.
simile
p
f
ppp
mp
ff

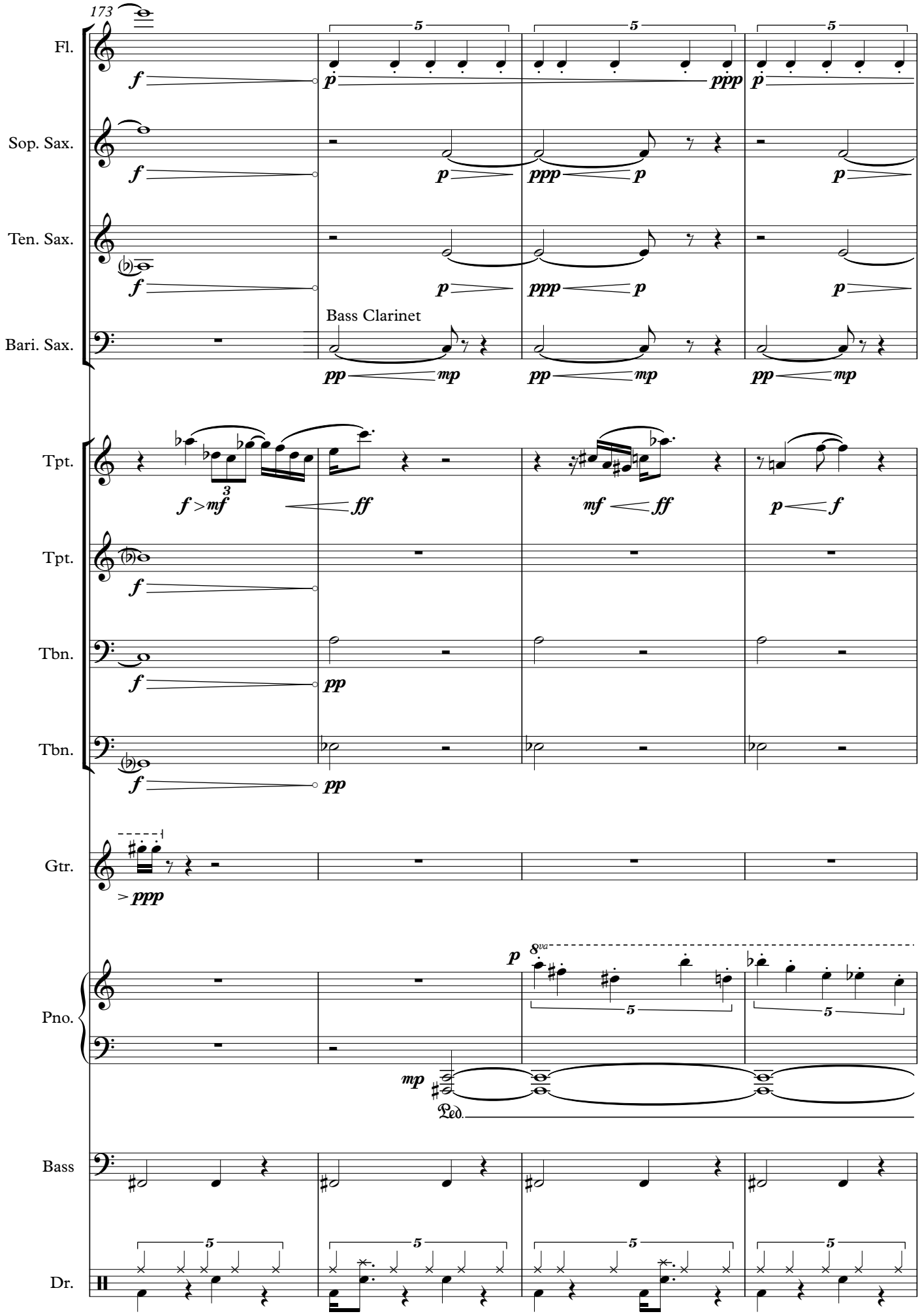

151
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
(8)
8vb

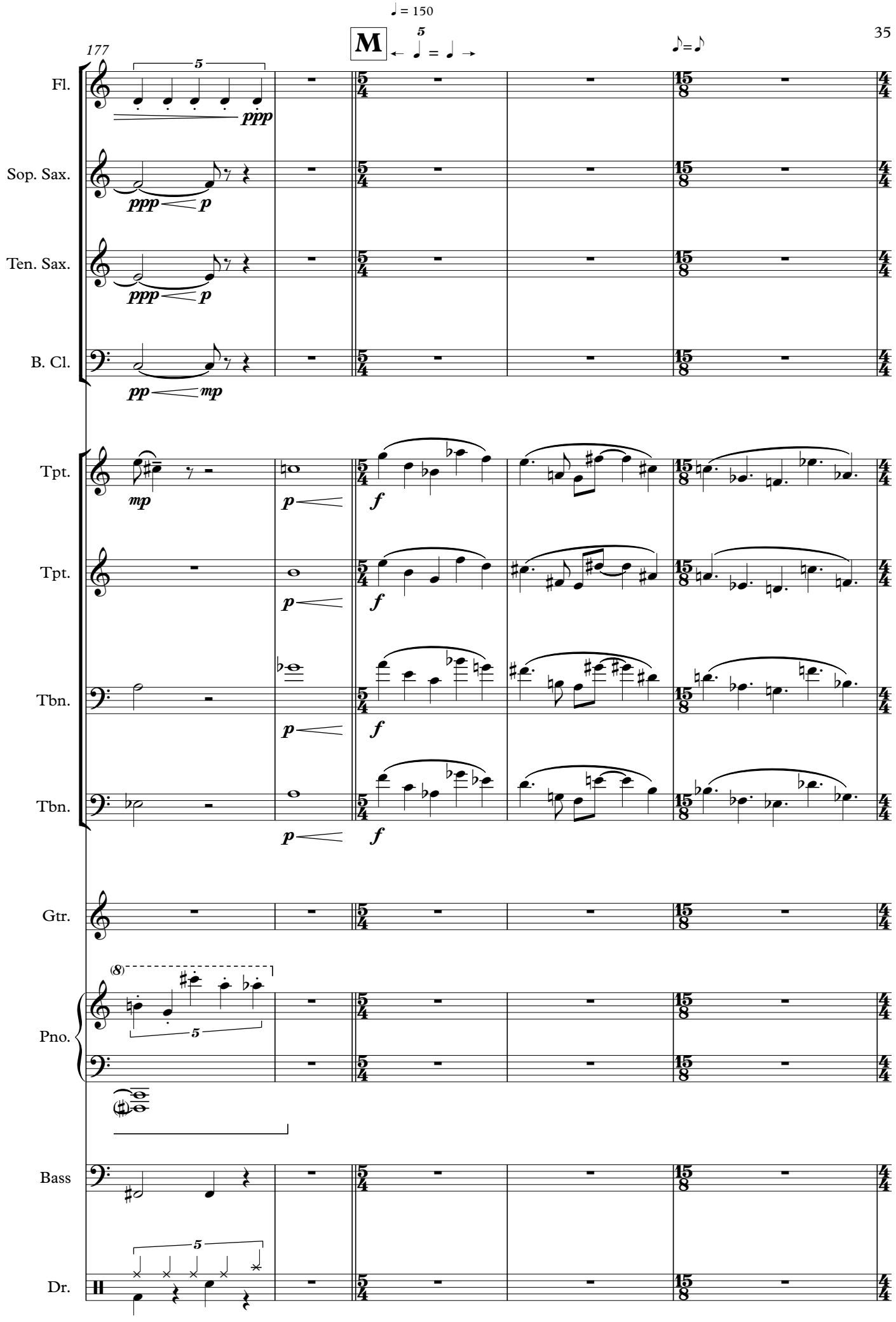

154
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
To B. Cl.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
mf
p
ppp

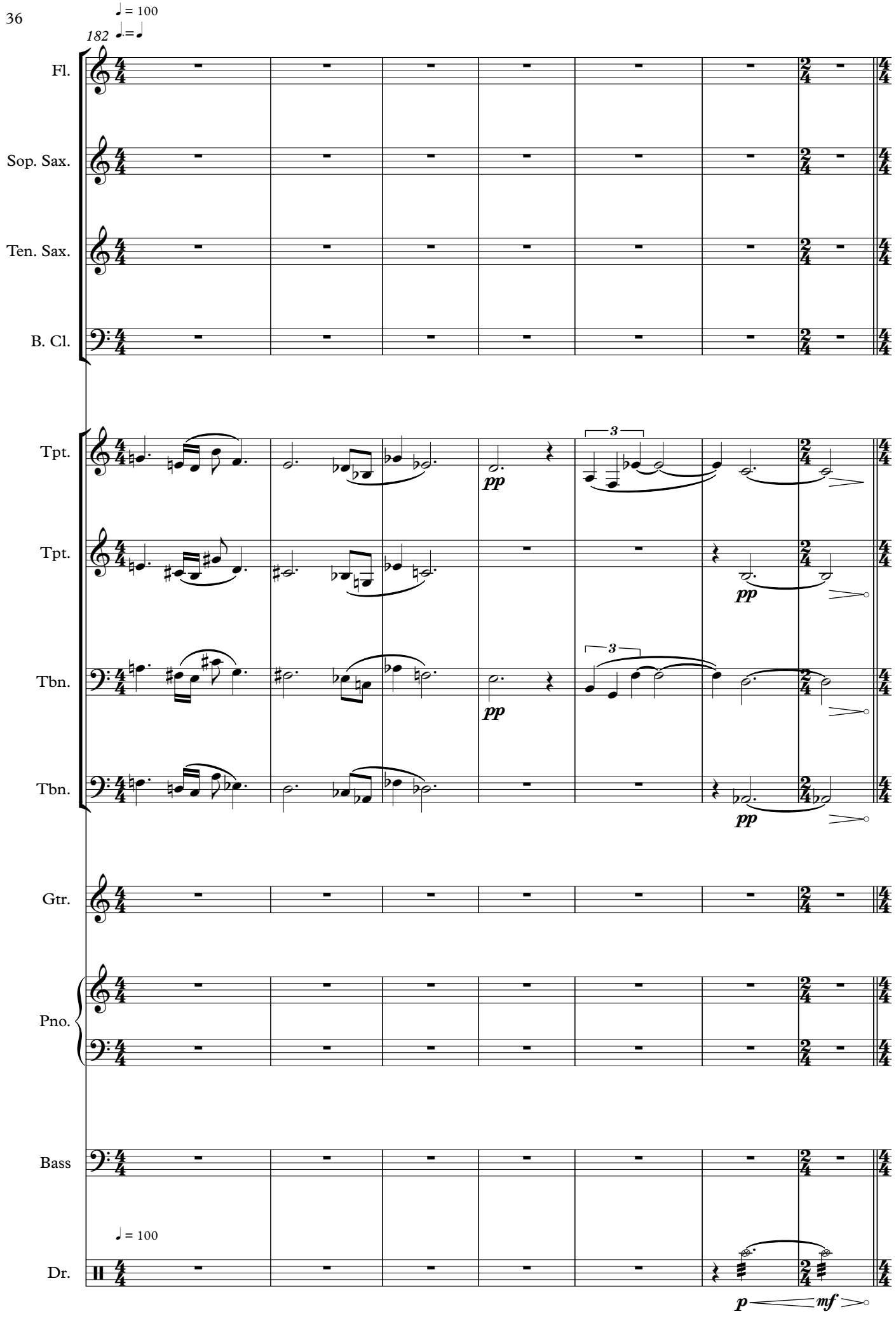

158
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
mf
ppp
simile

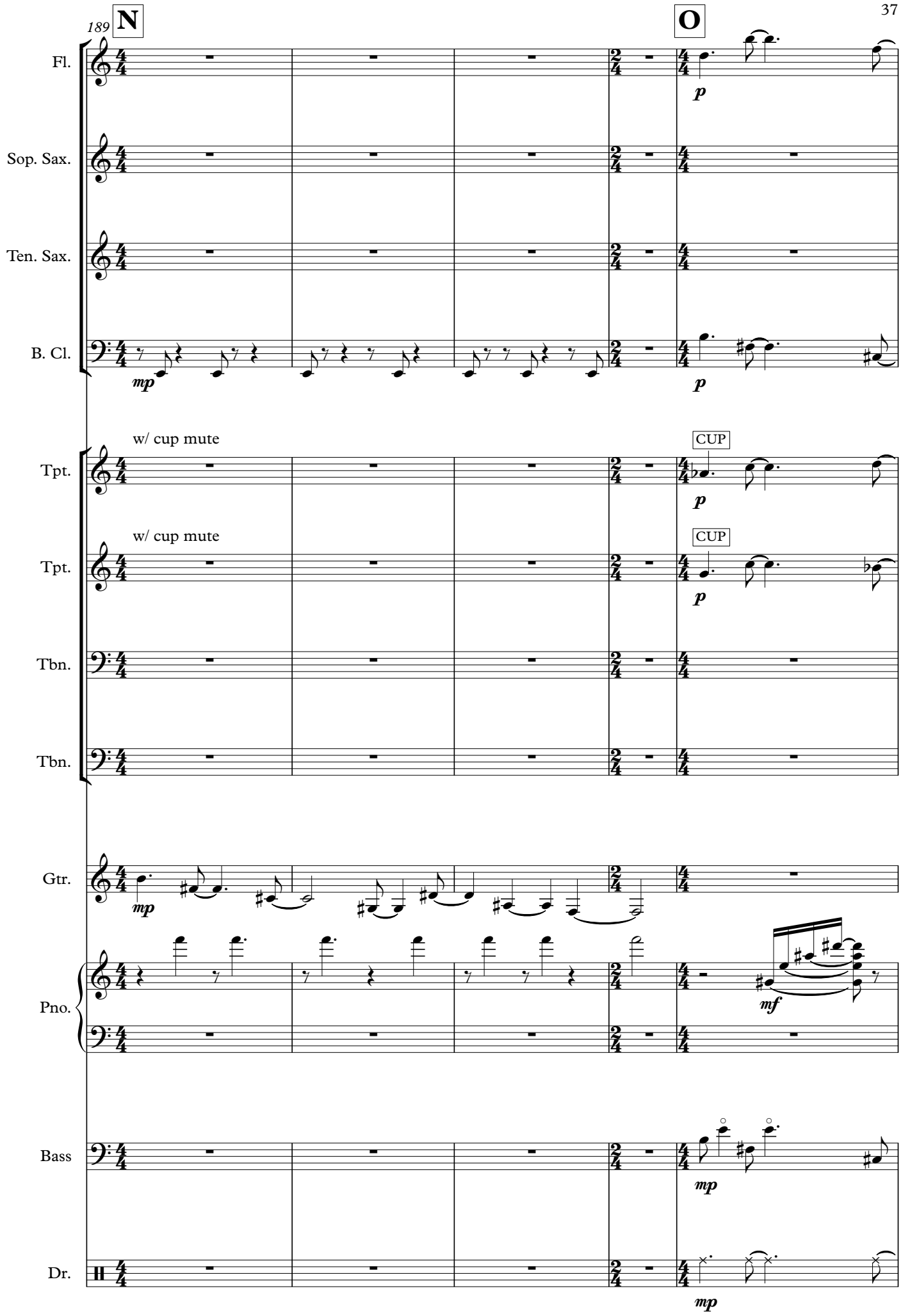

164
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
f
p
mf
ppp
3
5

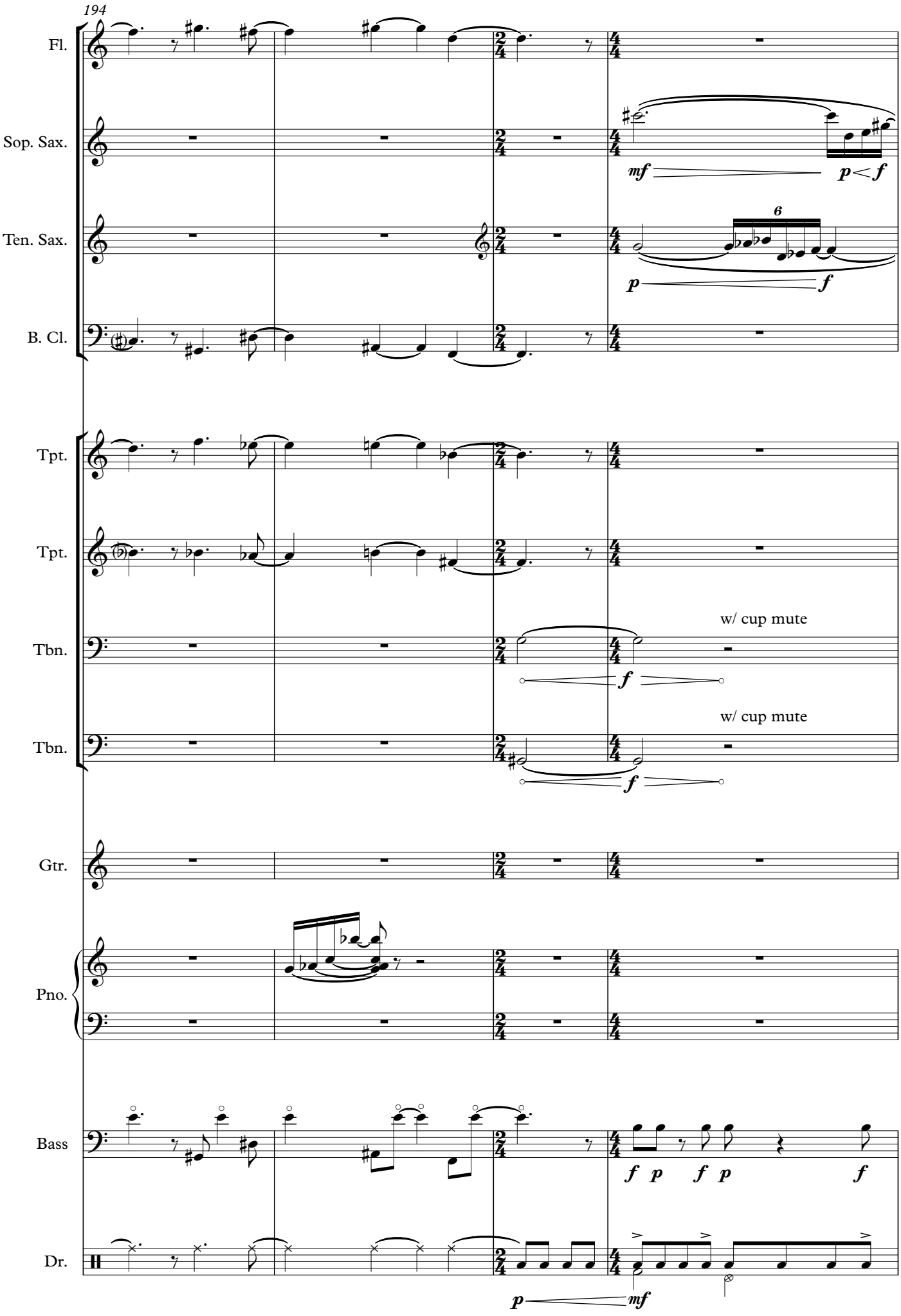

169
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
palm mute
pp
f
mf
mp
ppp
5

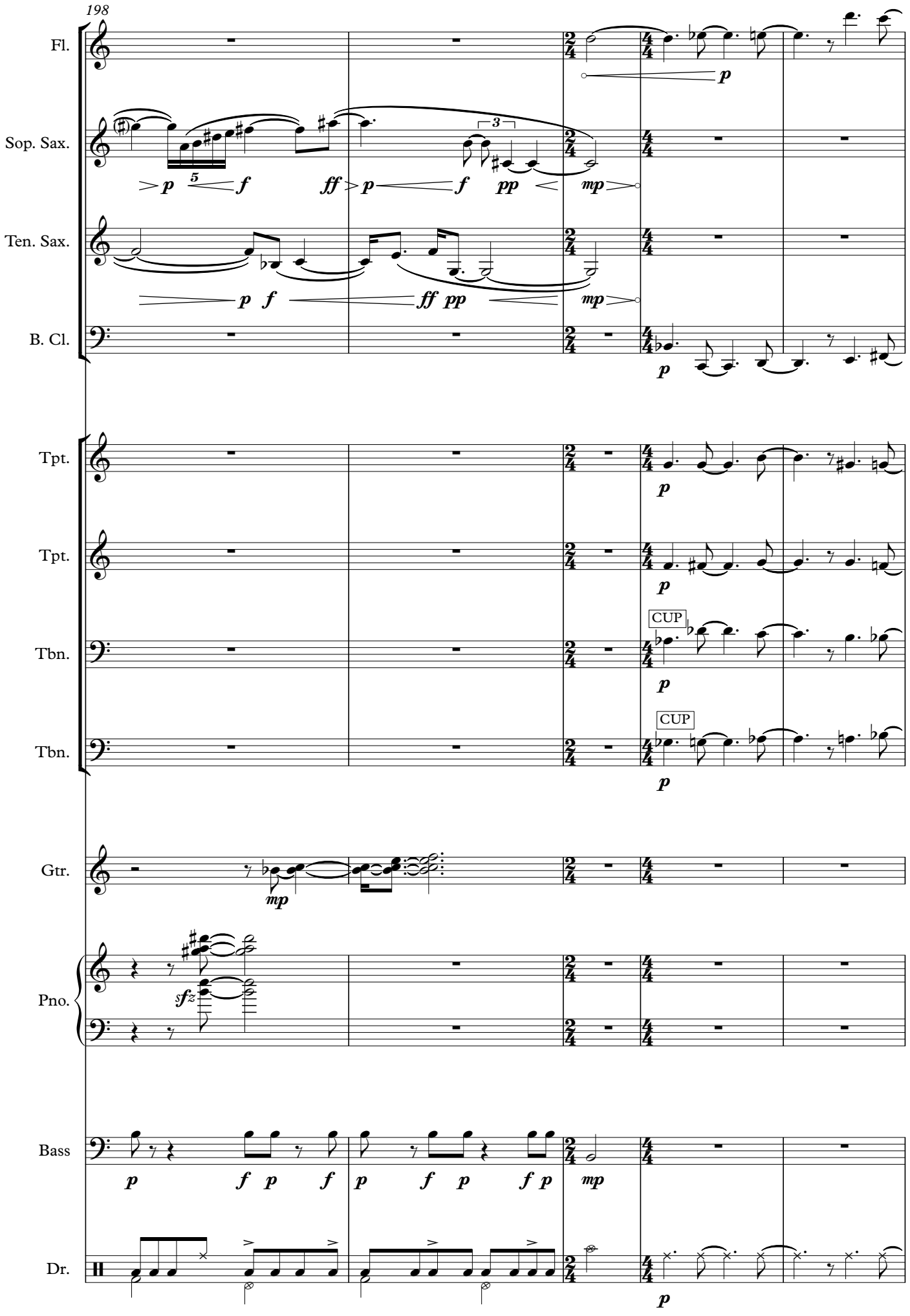

173
Fl.
Sop. Sax.
Ten. Sax.
Bari. Sax.
Bass Clarinet
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
8va
Ped.
Bass
Dr.

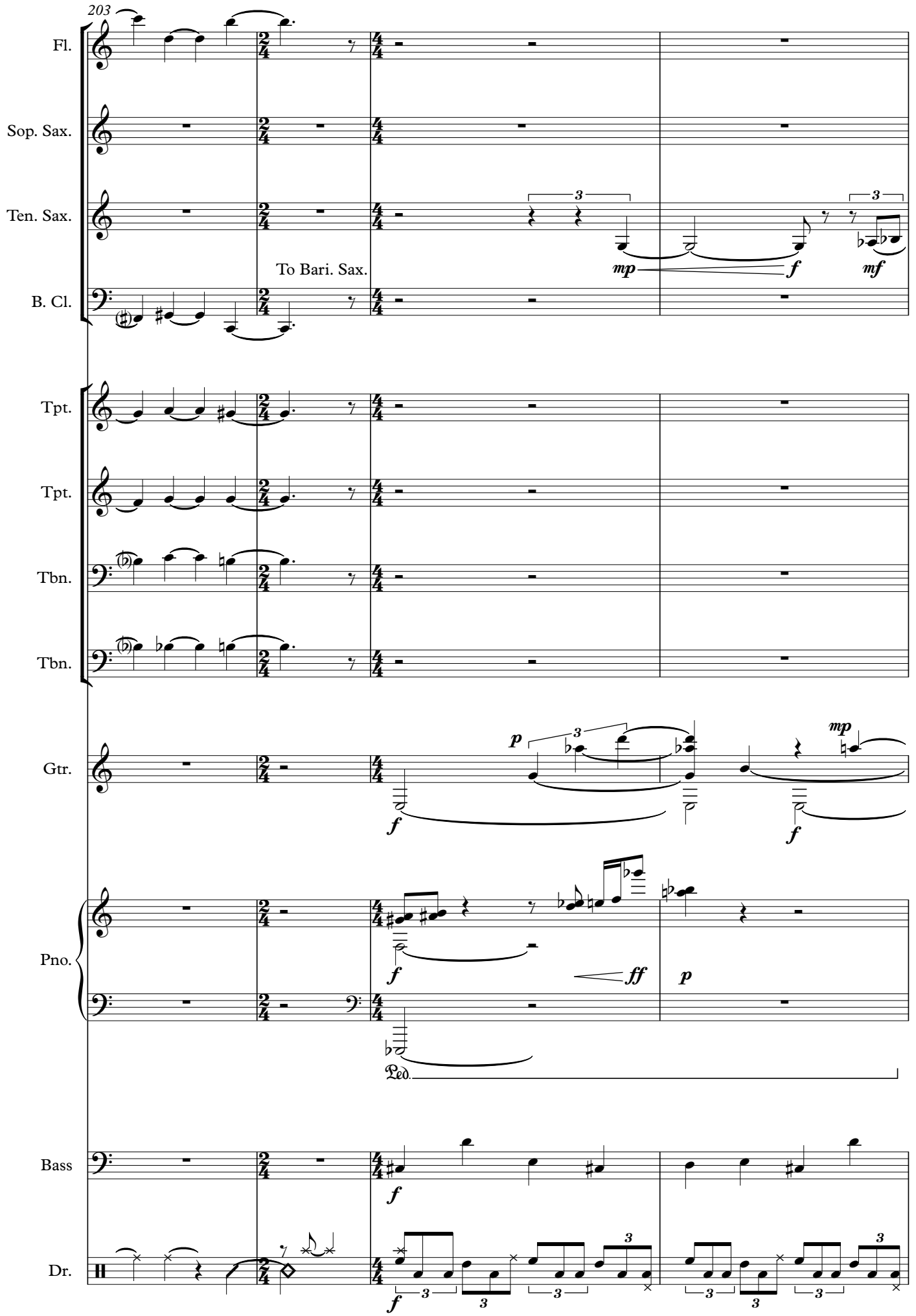

♩ = 150
M
177
Fl.
Sop. Sax.
Ten. Sax.
B. Cl.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
ppp
p
pp
mp
f

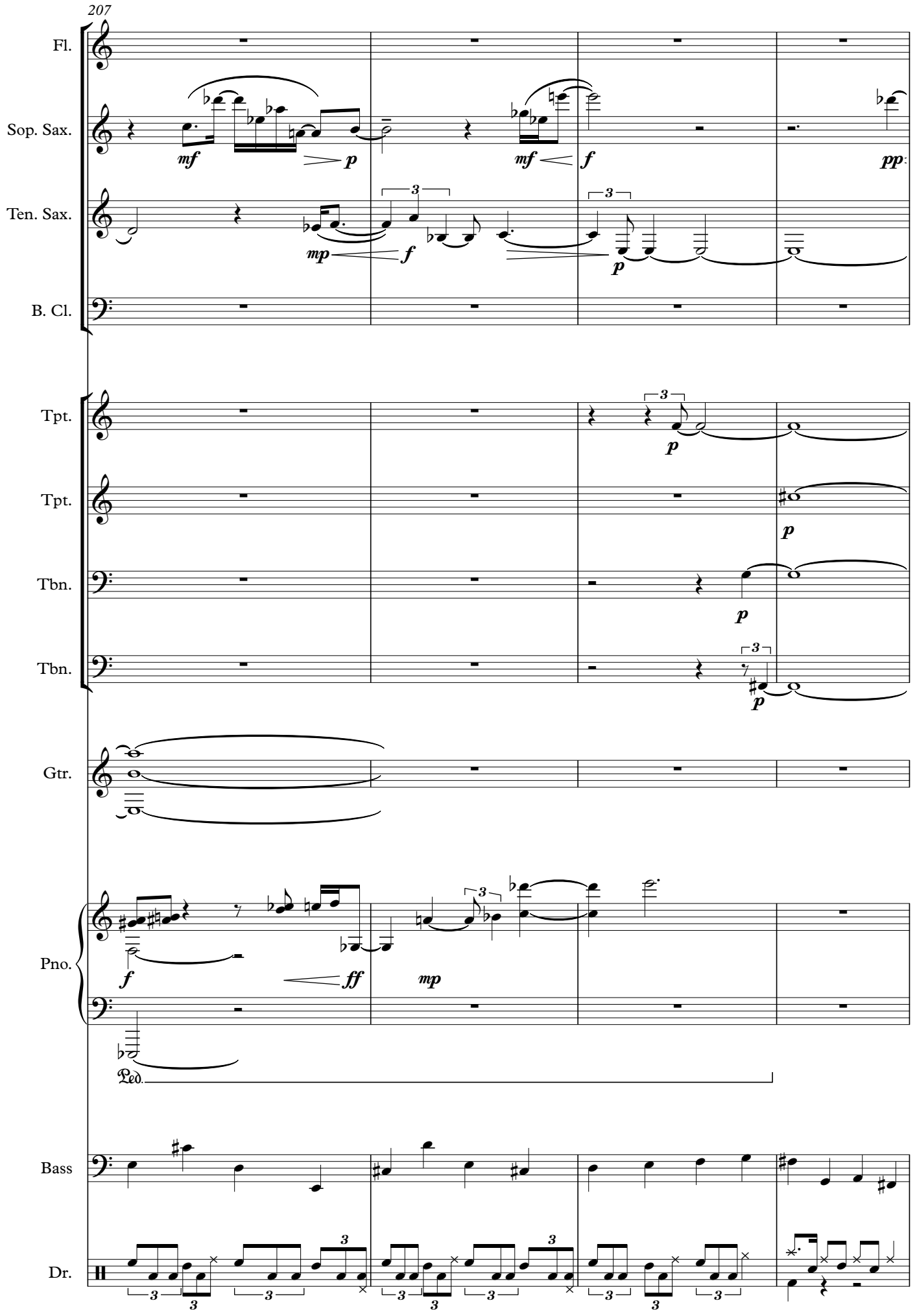

182
♩ = 100
♩.=♩
Fl.
Sop. Sax.
Ten. Sax.
B. Cl.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
3
pp
3
pp
pp
p
mf

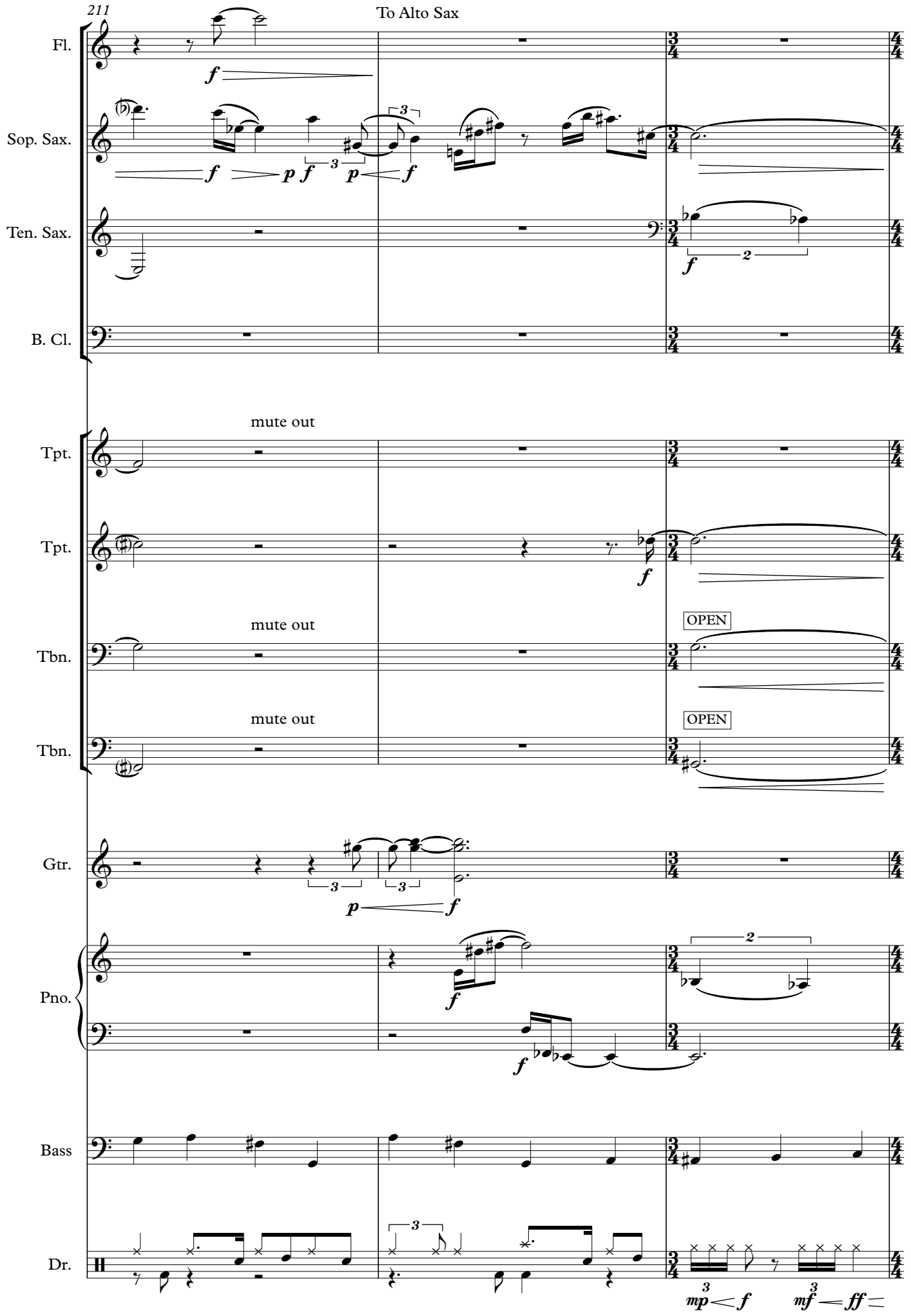

189
N
O
Fl.
Sop. Sax.
Ten. Sax.
B. Cl.
mp
p
w/ cup mute
CUP
Tpt.
Tbn.
Gtr.
Pno.
mf
Bass
Dr.

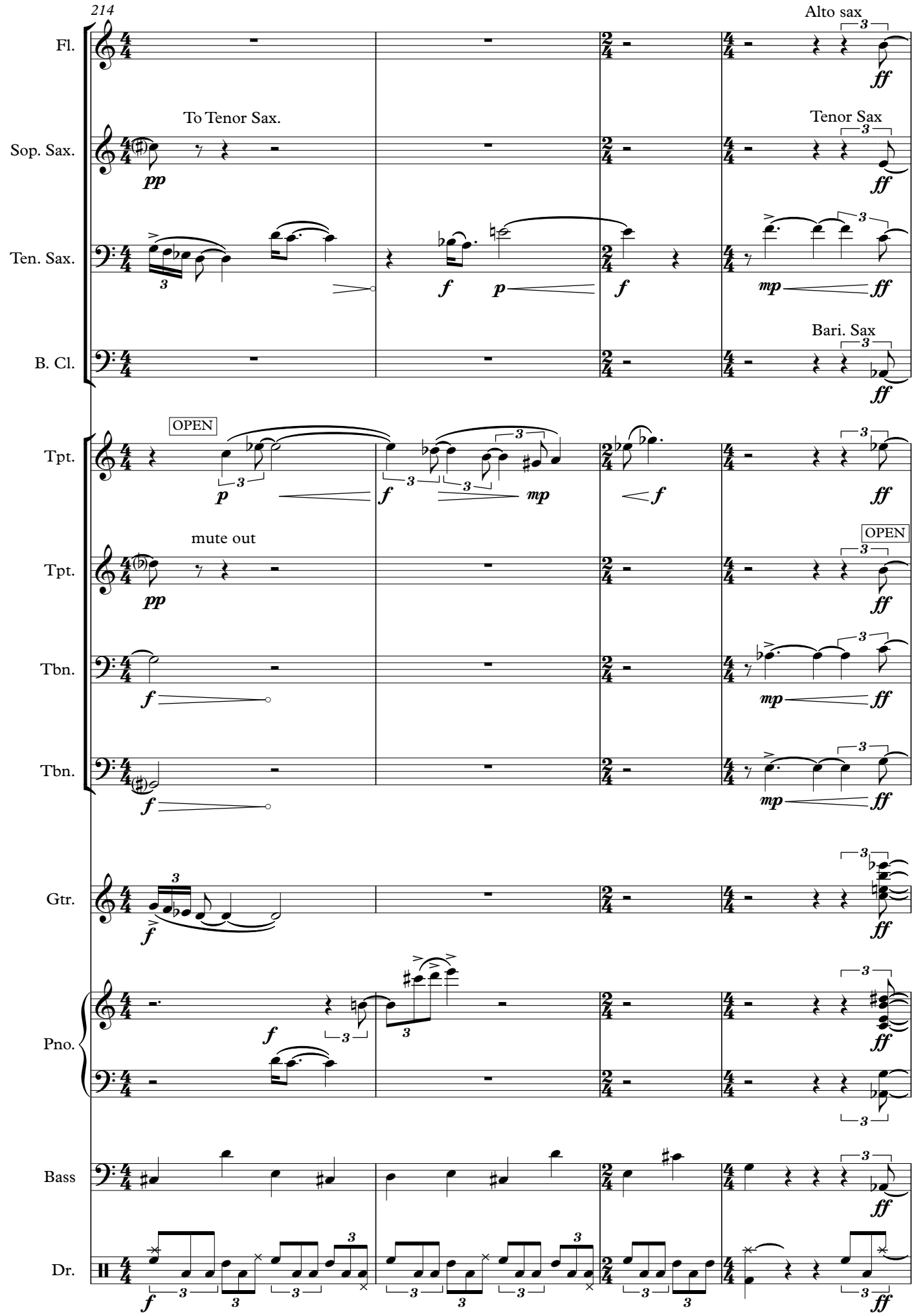

194
Fl.
Sop. Sax.
Ten. Sax.
B. Cl.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
mf
p < f
6
p
f
w/ cup mute
f
w/ cup mute
f
f p f p f
p
mf

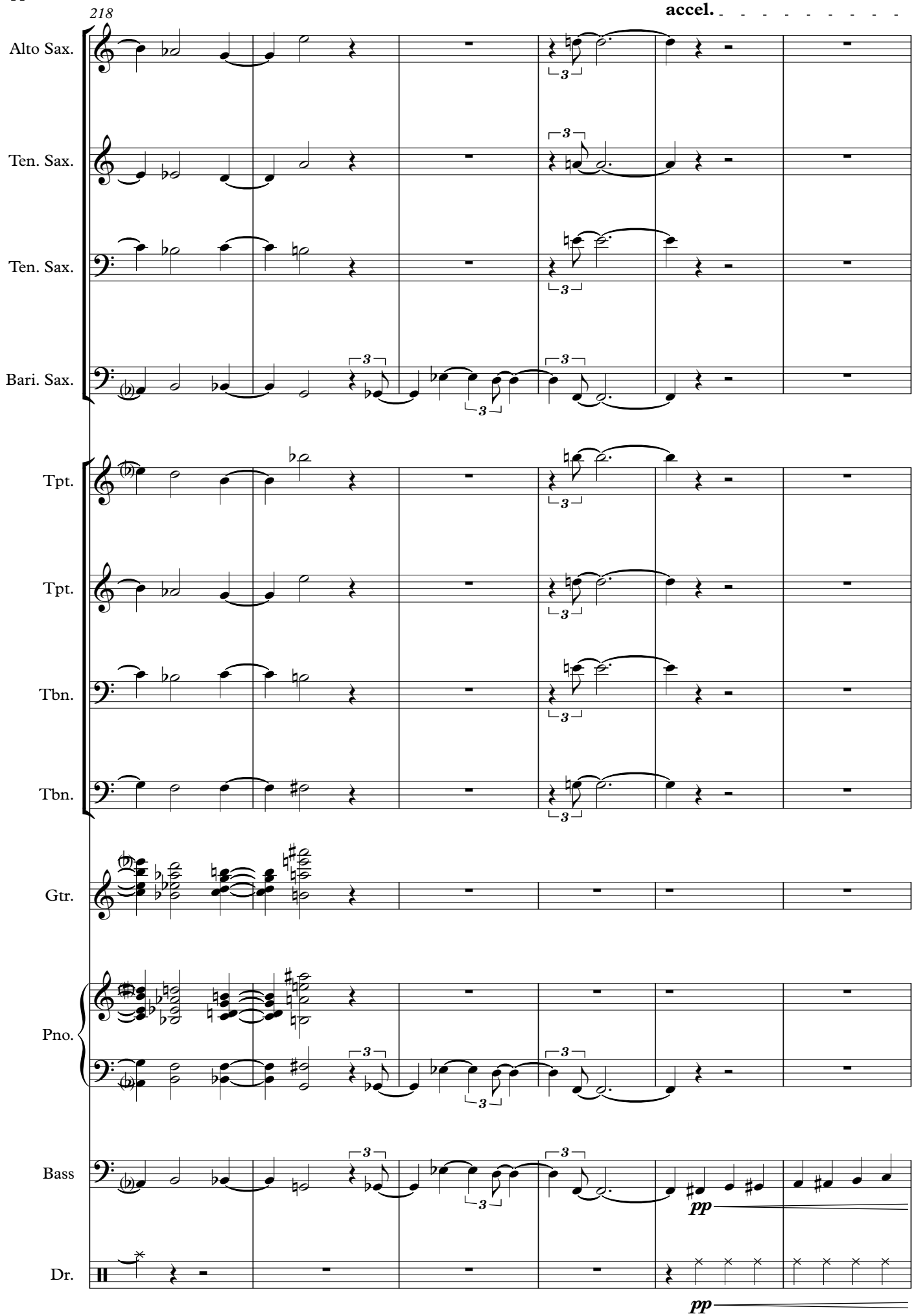

198
Fl.
Sop. Sax.
Ten. Sax.
B. Cl.
Tpt.
Tpt.
Tbn.
Tbn.
CUP
CUP
Gtr.
Pno.
Bass
Dr.

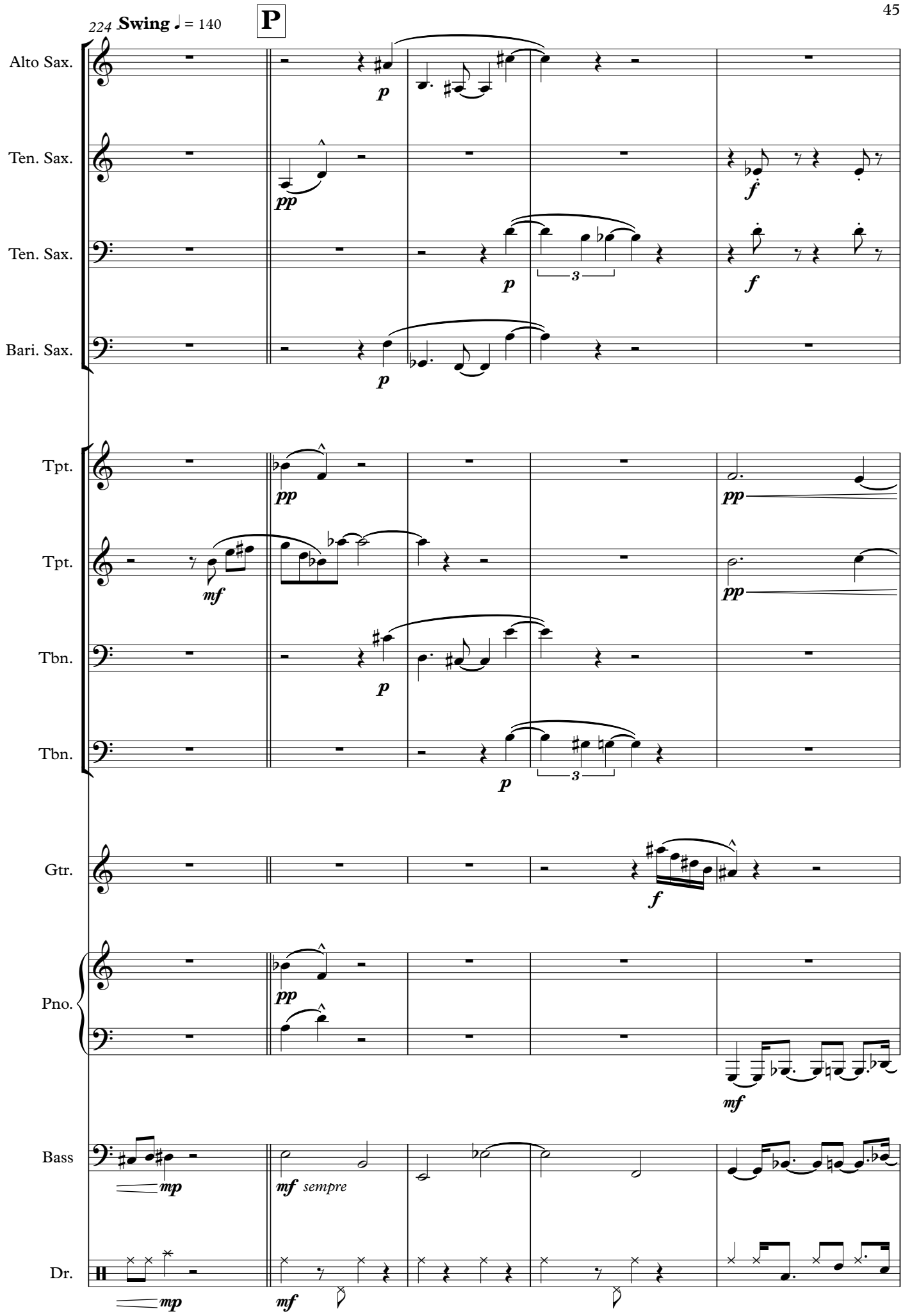

203
Fl.
Sop. Sax.
Ten. Sax.
To Bari. Sax.
B. Cl.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
mp
f
mf
p
ff
Ped.

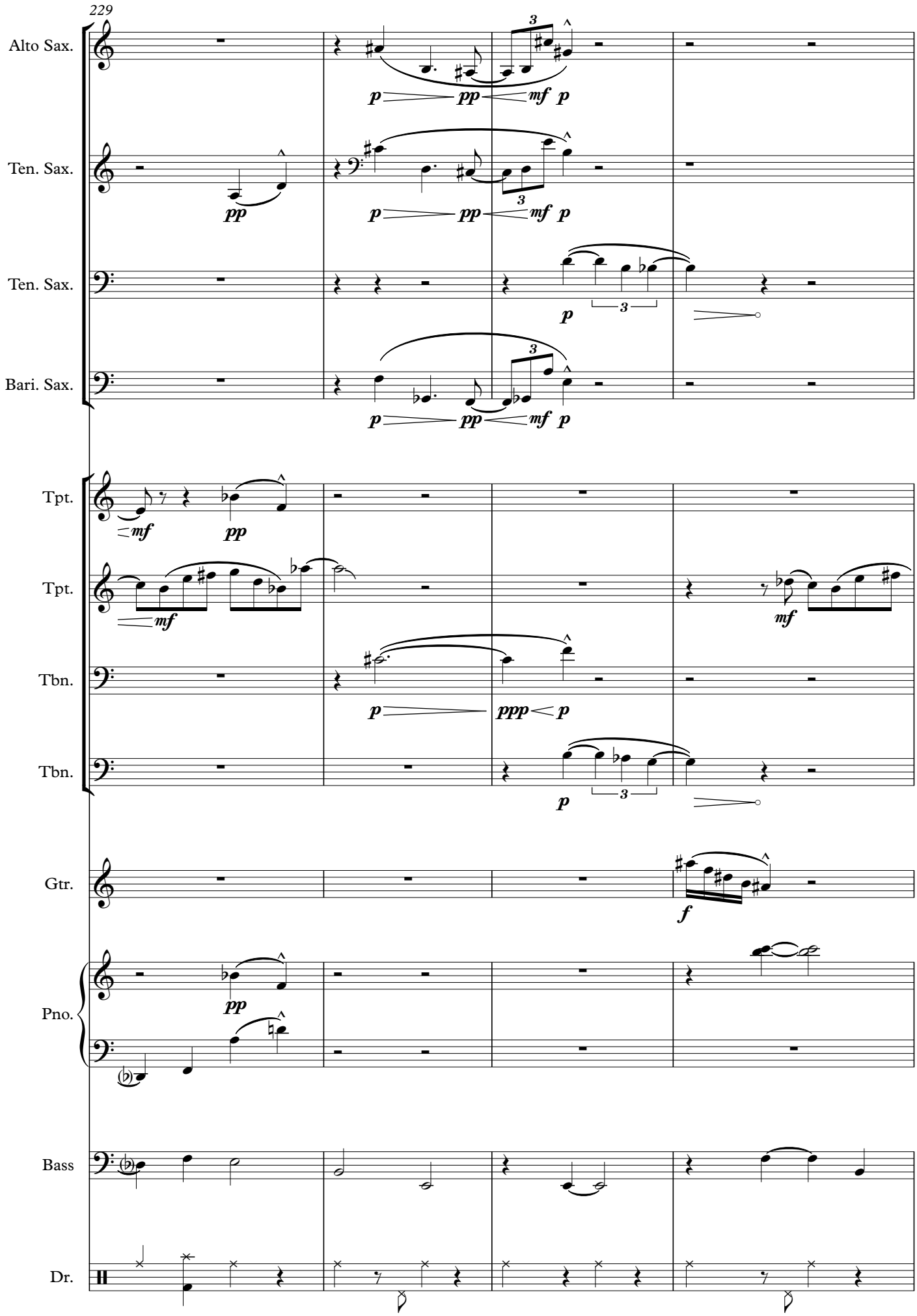

207
Fl.
Sop. Sax.
Ten. Sax.
B. Cl.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
mf
p
mf
f
pp
mp
f
p
p
p
p
f
ff
mp
Ped.

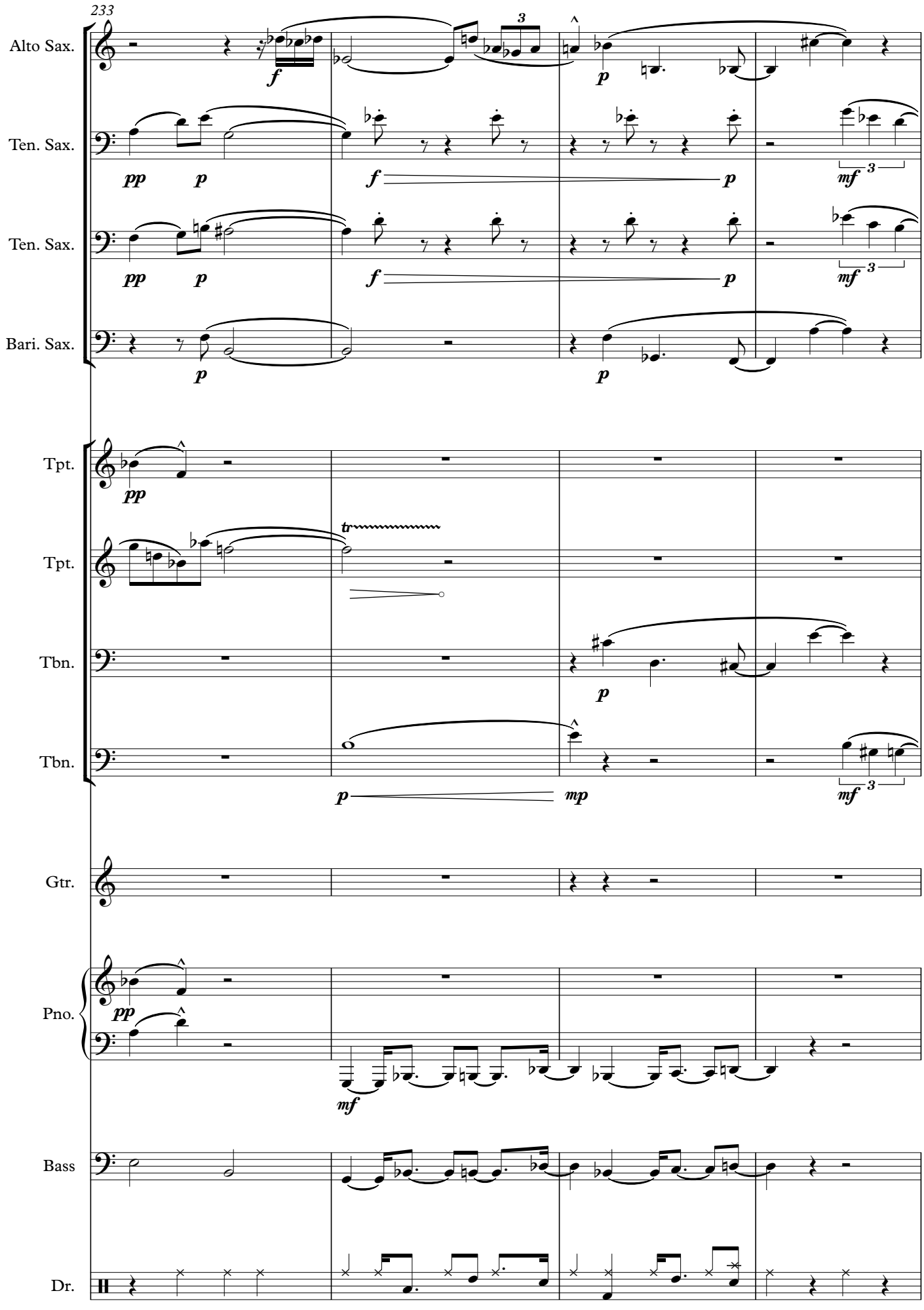

211
To Alto Sax
Fl.
Sop. Sax.
Ten. Sax.
B. Cl.
mute out
Tpt.
Tpt.
mute out
OPEN
Tbn.
mute out
OPEN
Tbn.
Gtr.
Pno.
Bass
Dr.

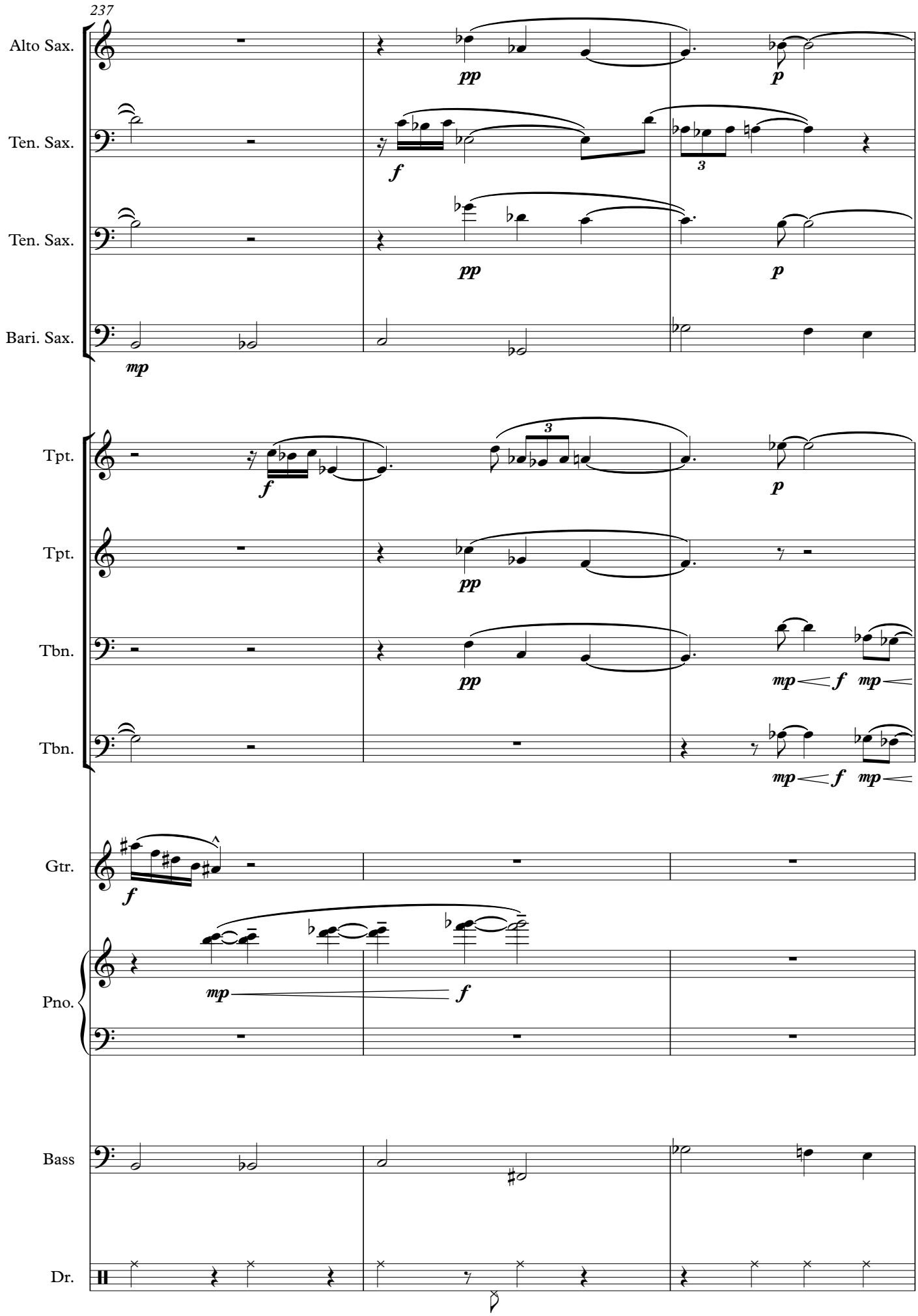

214
Fl.
Alto sax
Sop. Sax.
To Tenor Sax.
Tenor Sax
Ten. Sax.
B. Cl.
Bari. Sax
Tpt.
OPEN
mute out
Tbn.
Gtr.
Pno.
Bass
Dr.

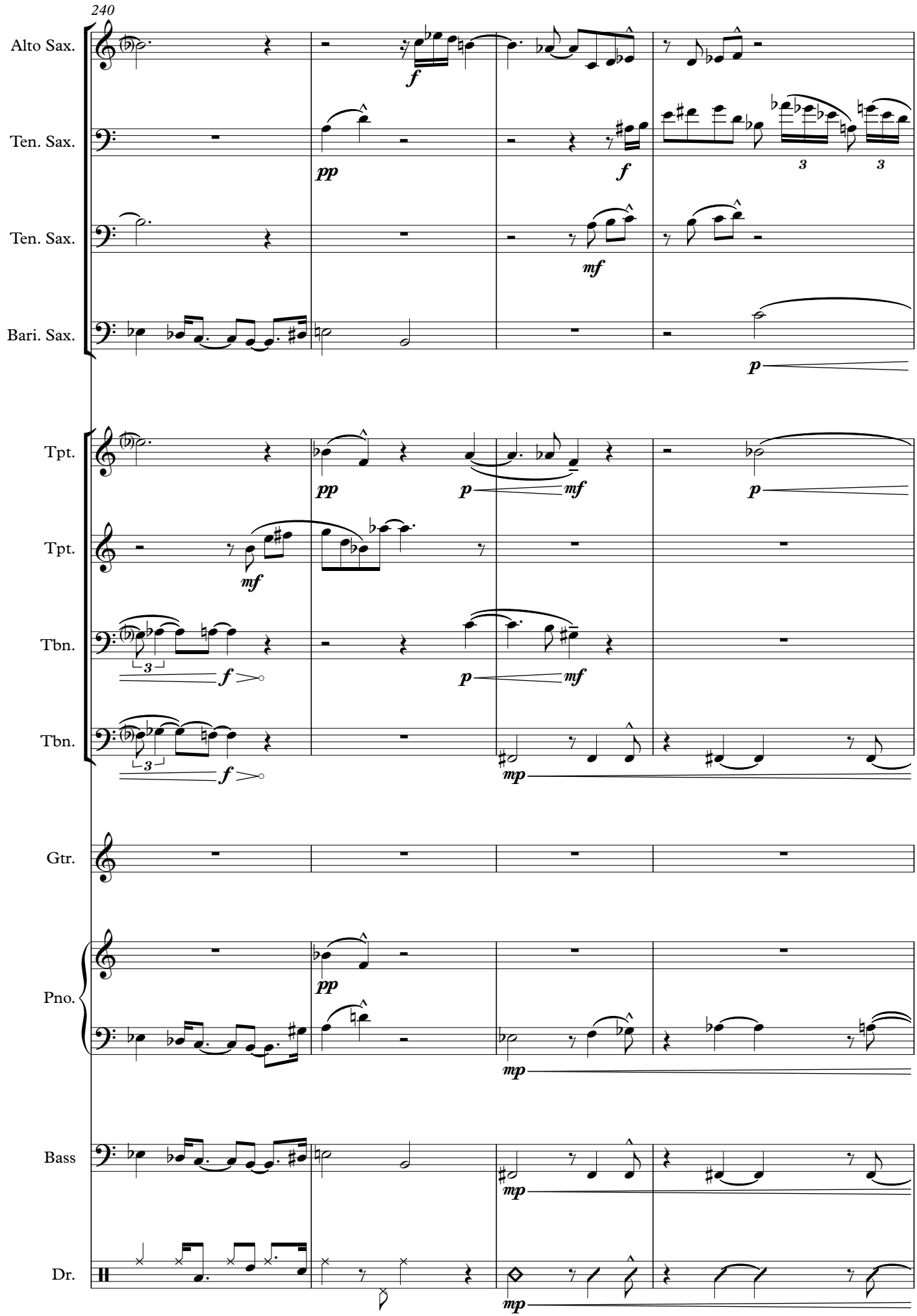

218
accel.
Alto Sax.
Ten. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
pp
pp

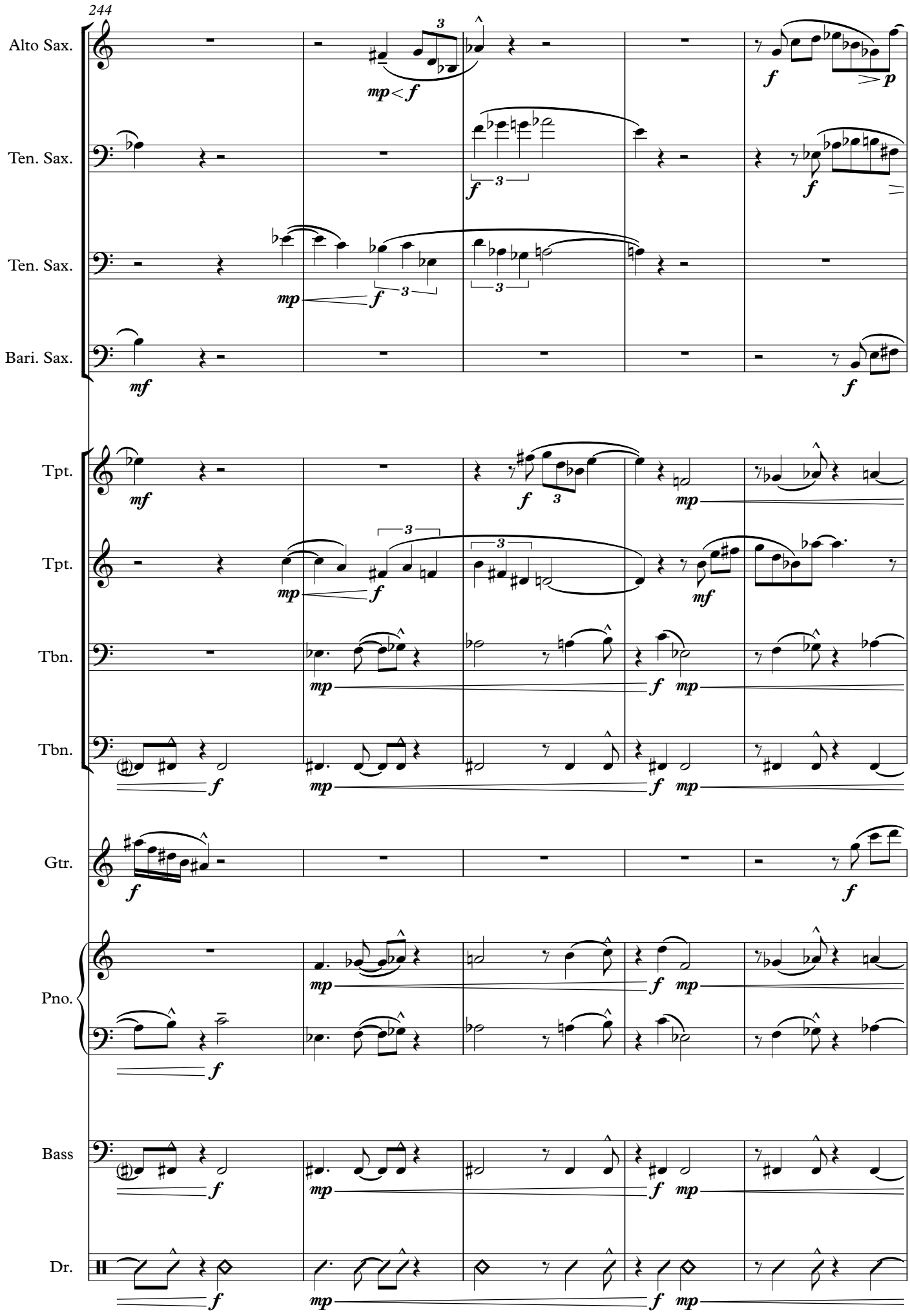

224
Swing ♩ = 140
P
Alto Sax.
Ten. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
sempre

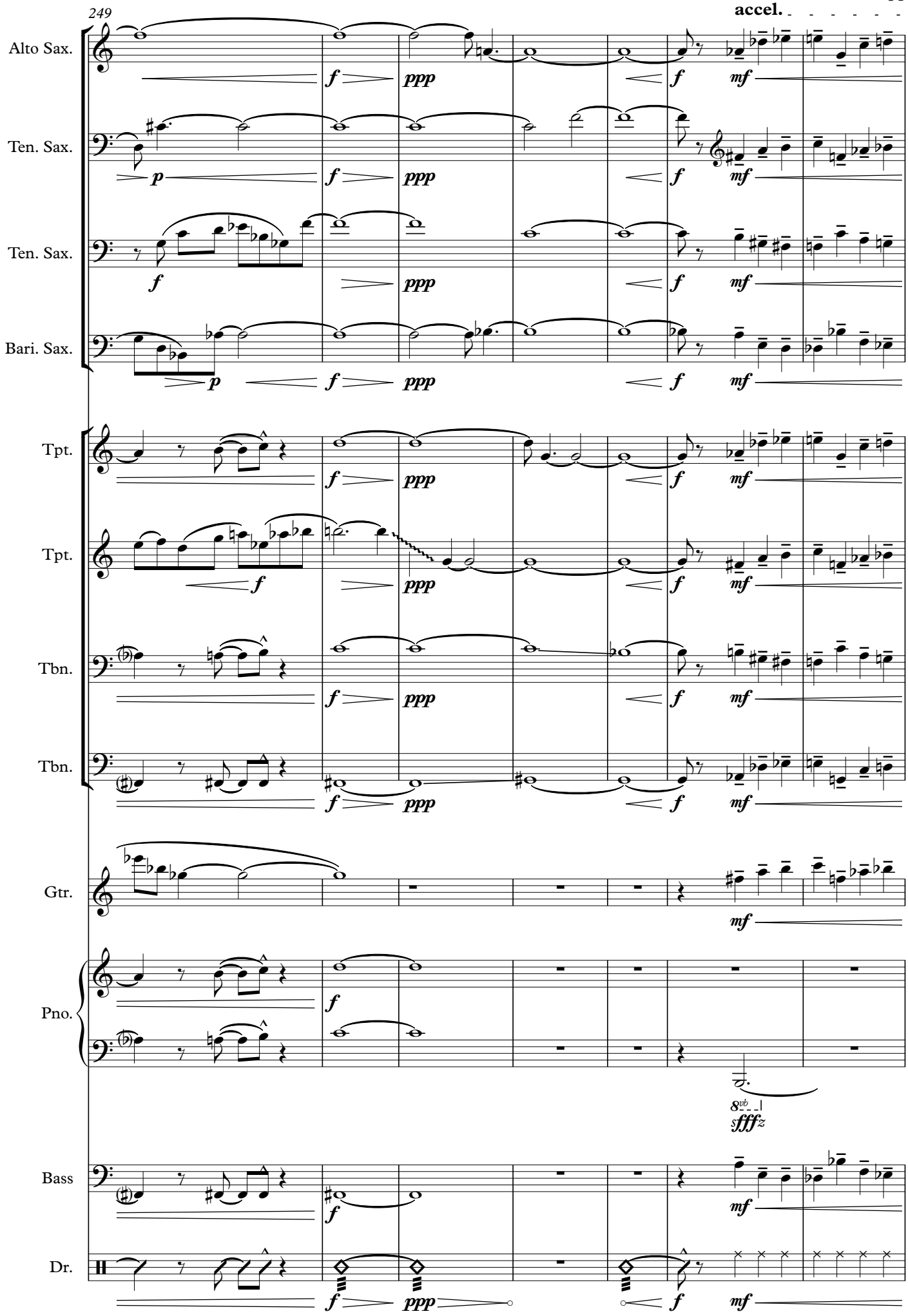

229
Alto Sax.
Ten. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.

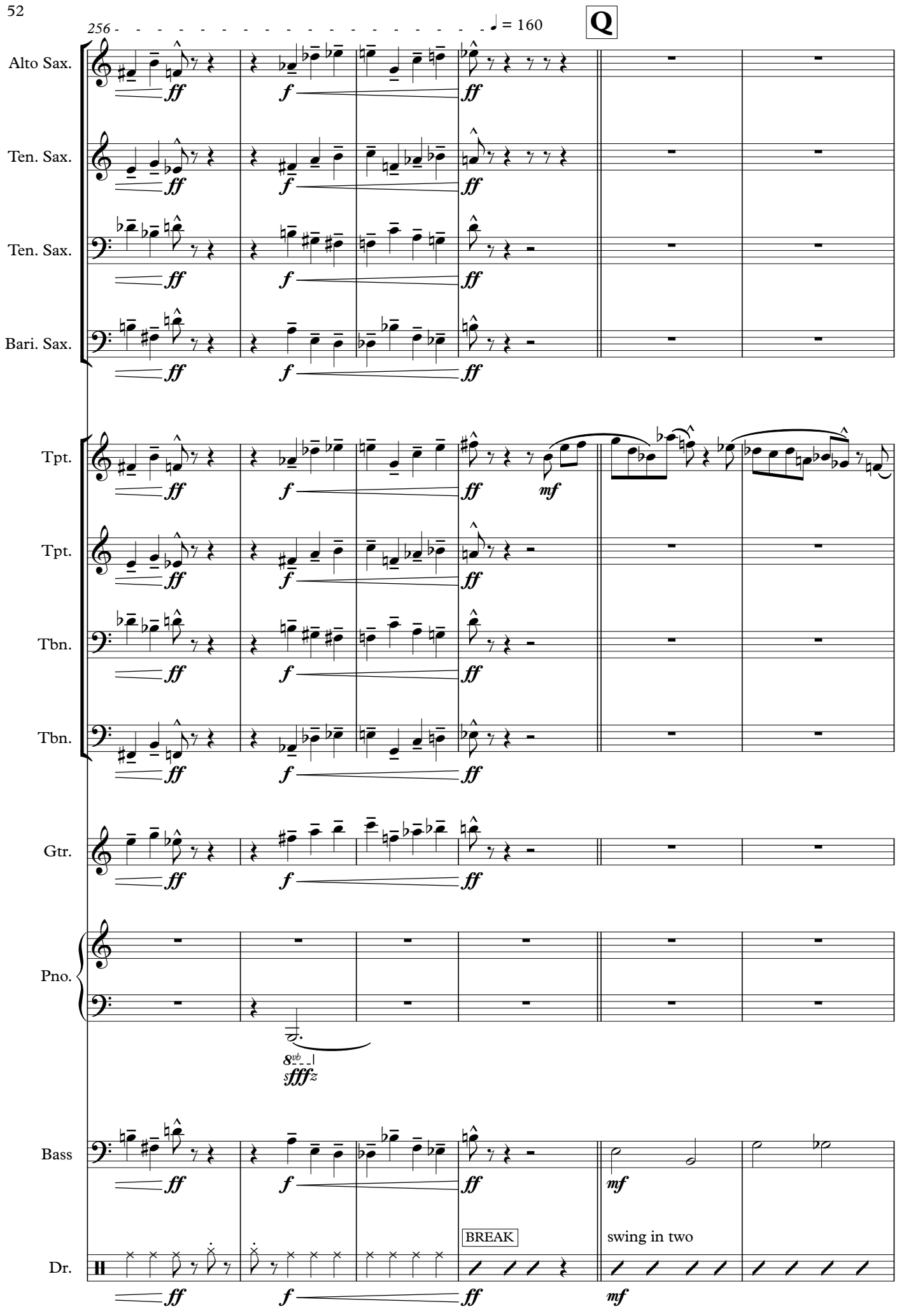

233
Alto Sax.
Ten. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
f
p
pp
mf
mp
tr
3

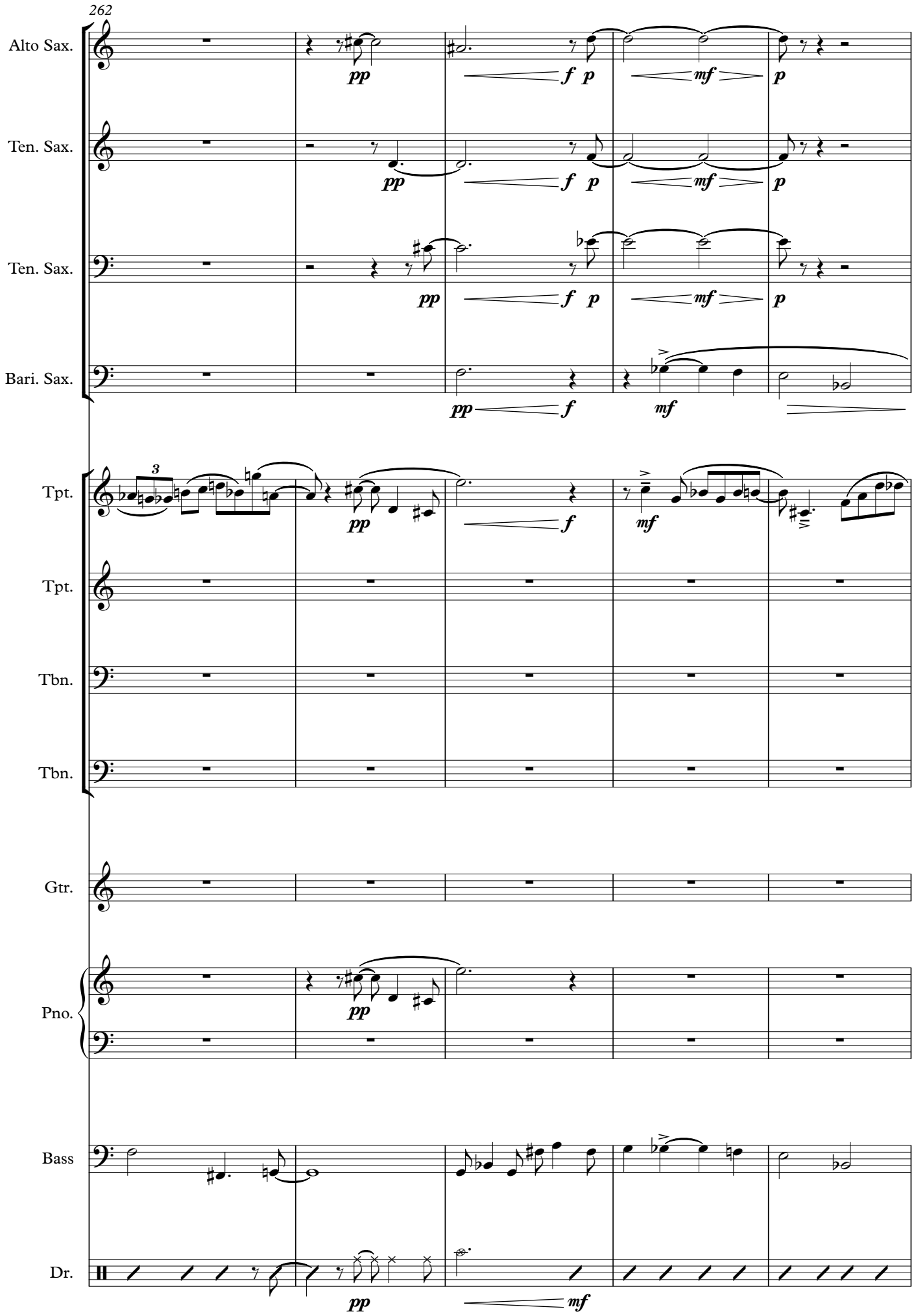

237
Alto Sax.
Ten. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
pp
p
f
mp
3

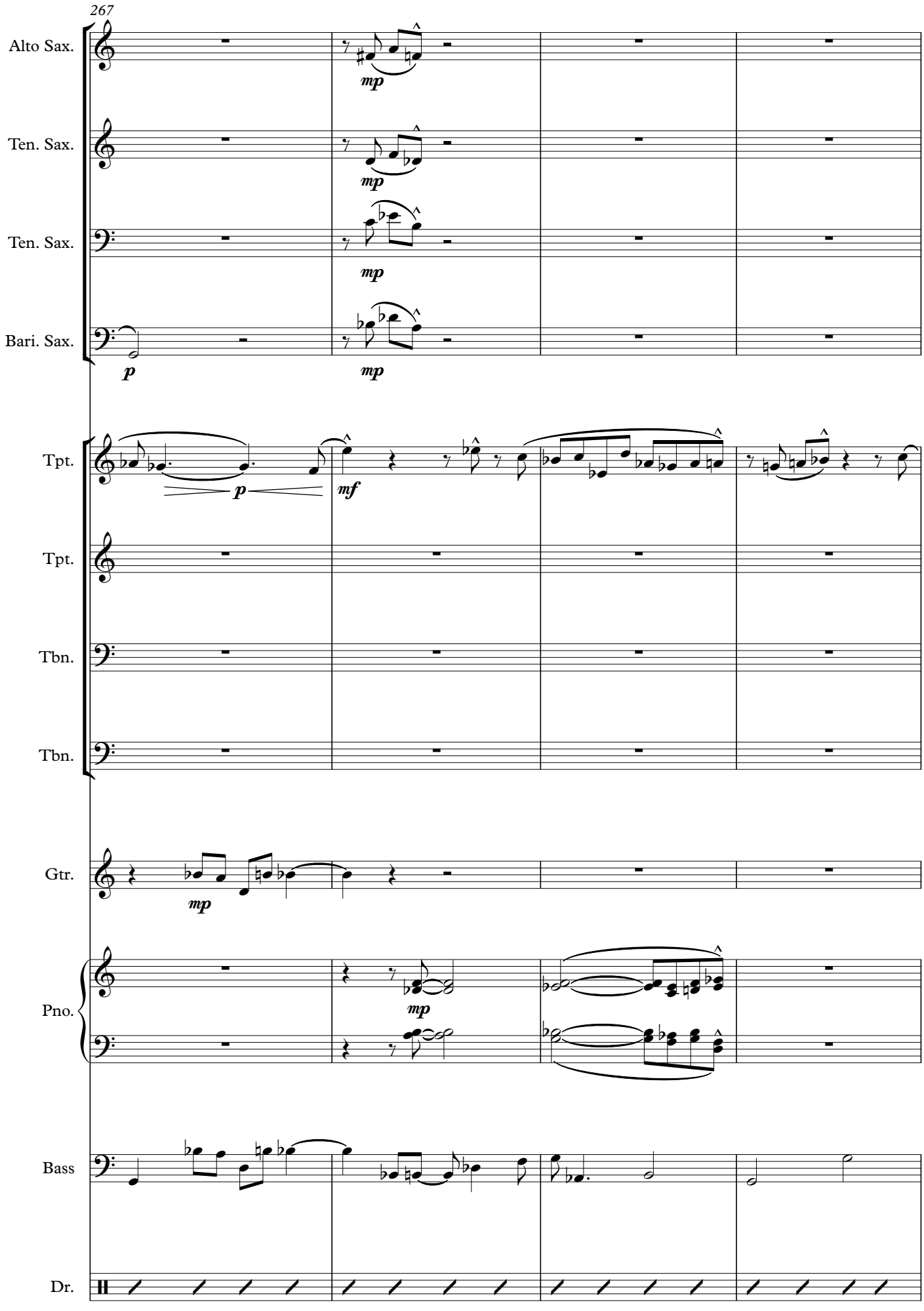

240
Alto Sax.
Ten. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.

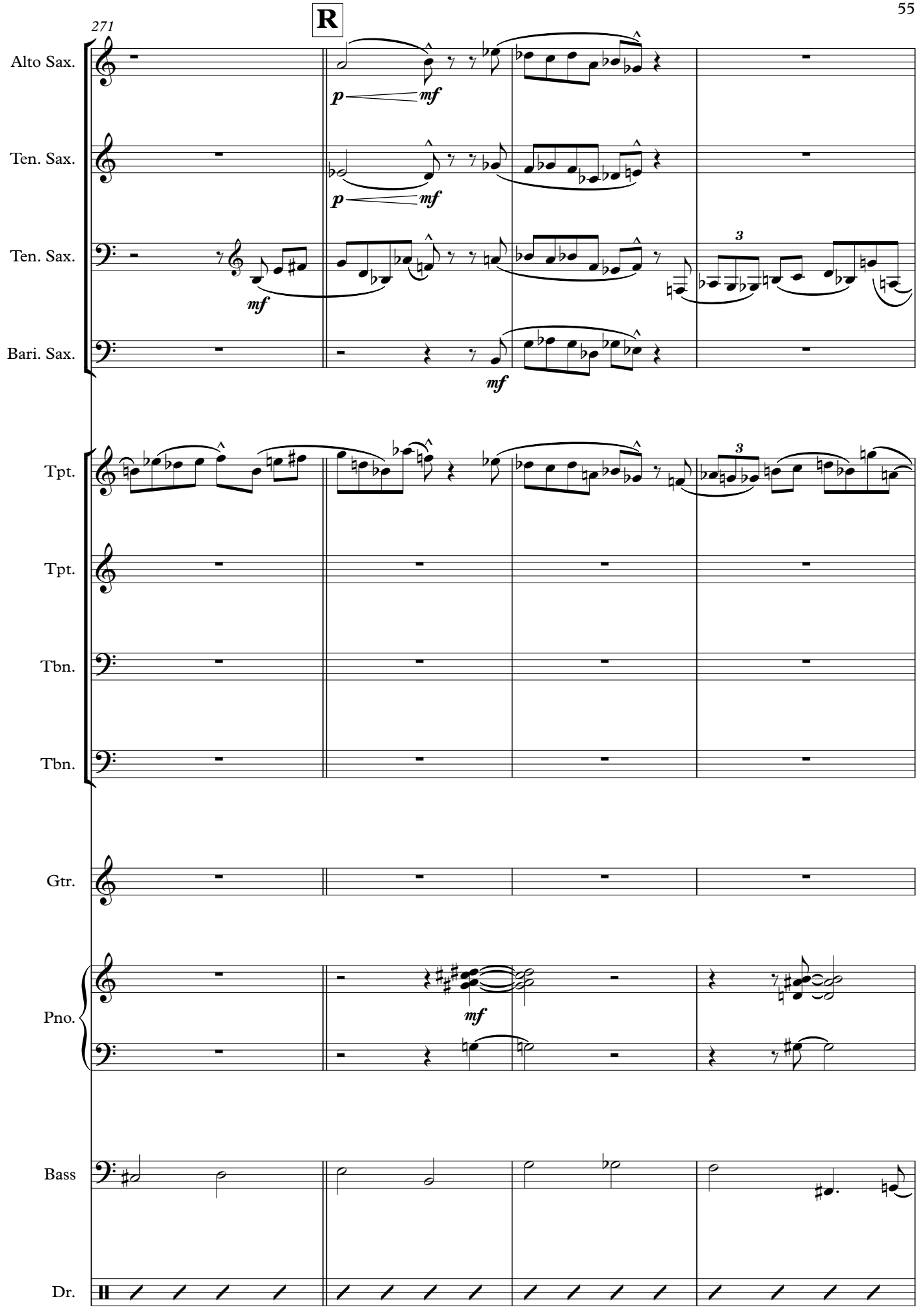

244
Alto Sax.
Ten. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.

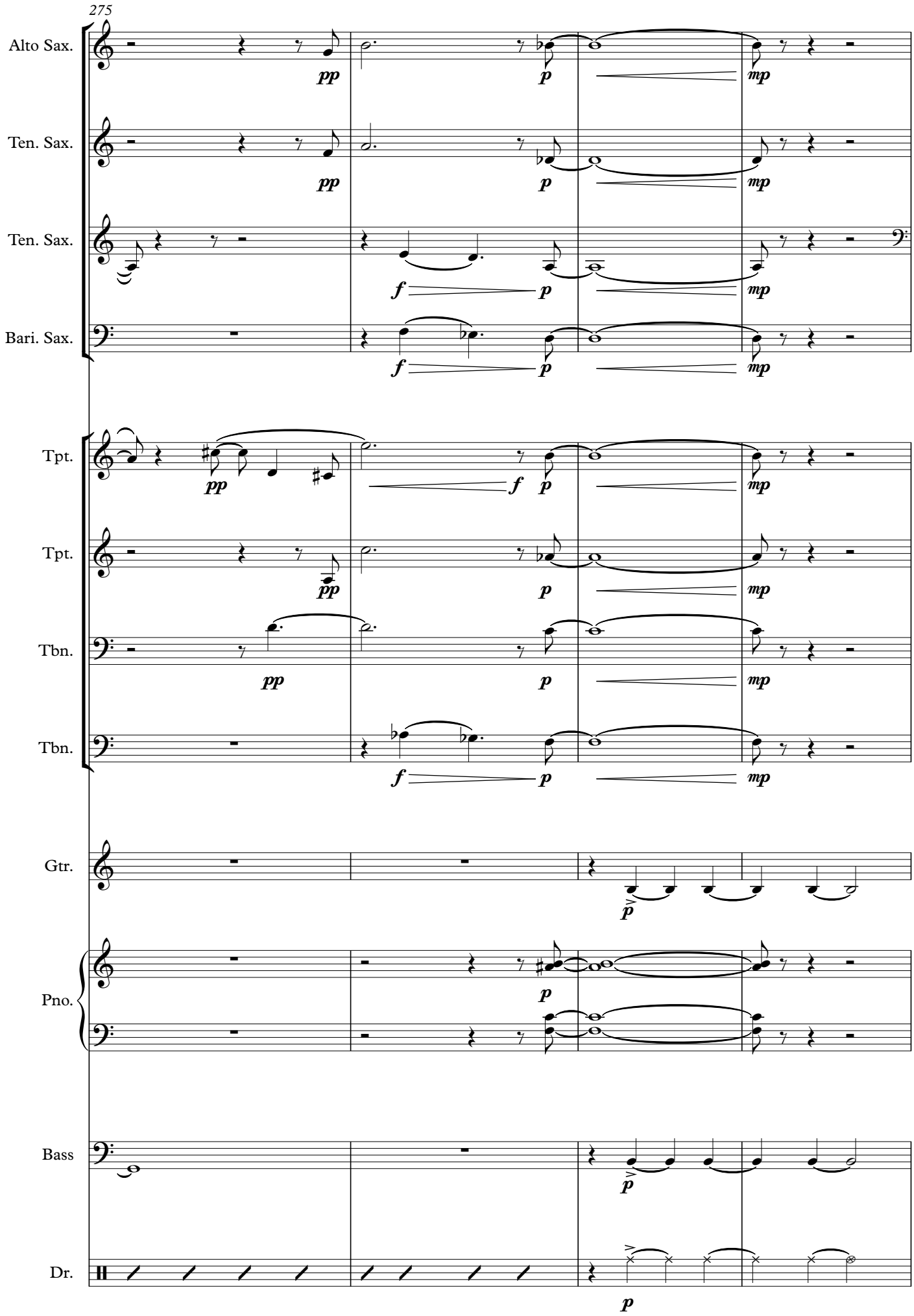

249
accel.
Alto Sax.
Ten. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
f
ppp
mf
p
8vb
sfffz

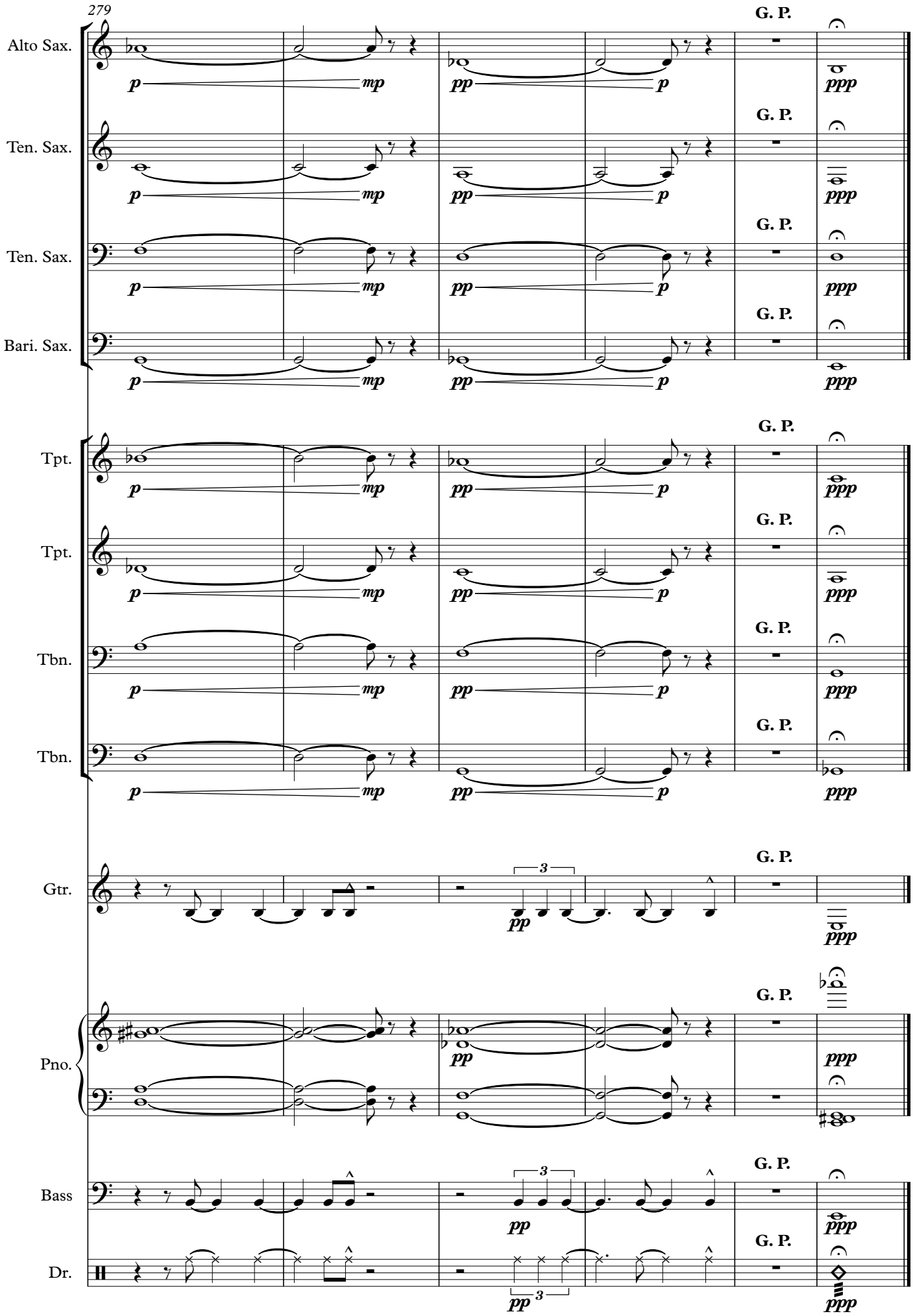

256
♩ = 160
Q
Alto Sax.
Ten. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
ff
f
mf
8vb
sffffz
BREAK
swing in two


262
Alto Sax.
Ten. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
pp
f
p
mf
3


267
Alto Sax.
Ten. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
mp
p
mf


271
R
Alto Sax.
Ten. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
p
mf
3

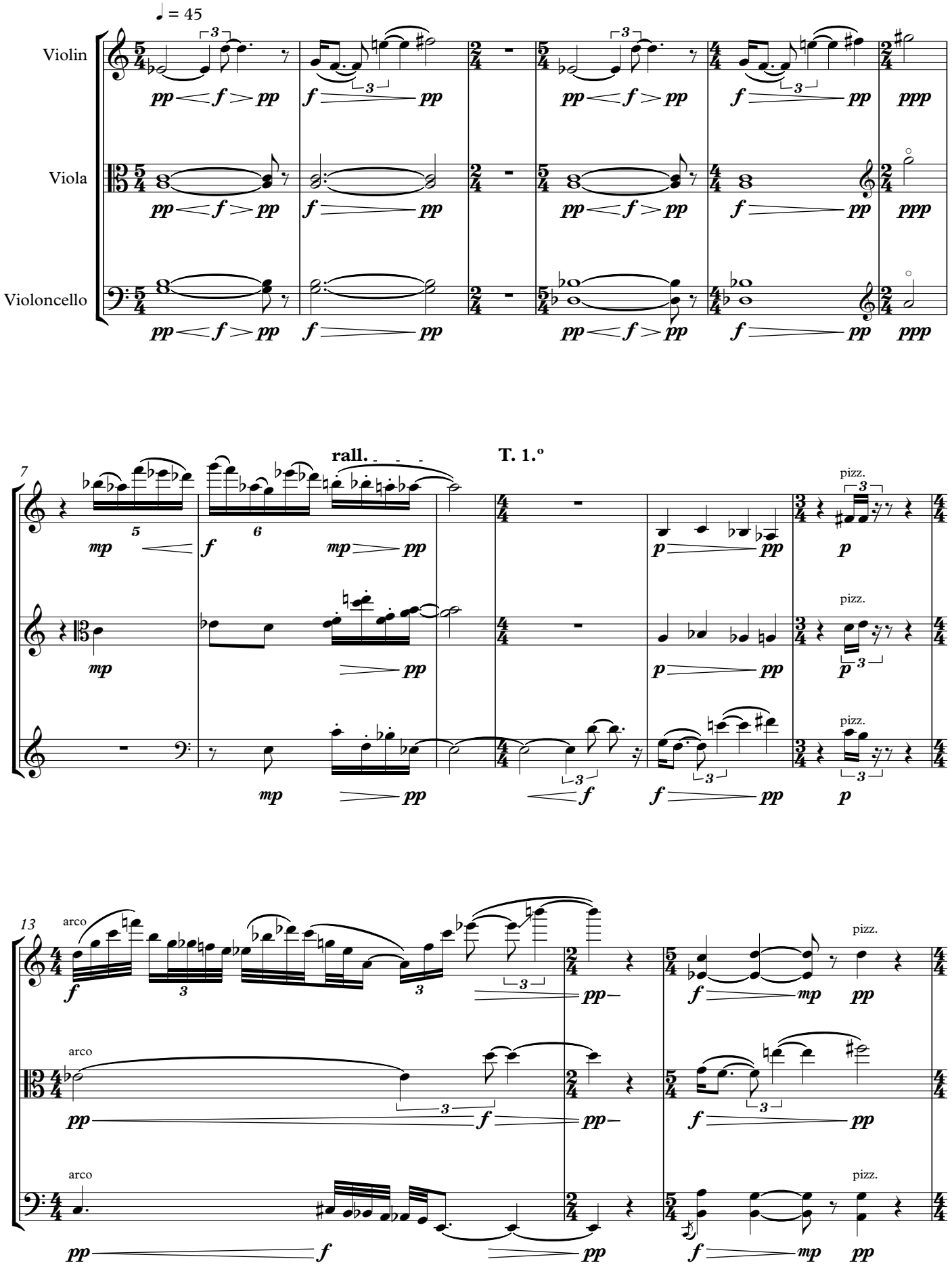

275
Alto Sax.
Ten. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
pp
p
mp
f
p
mp
pp
p
mp
f
p
mp
pp
p
mp
f
p
mp
p
p
p

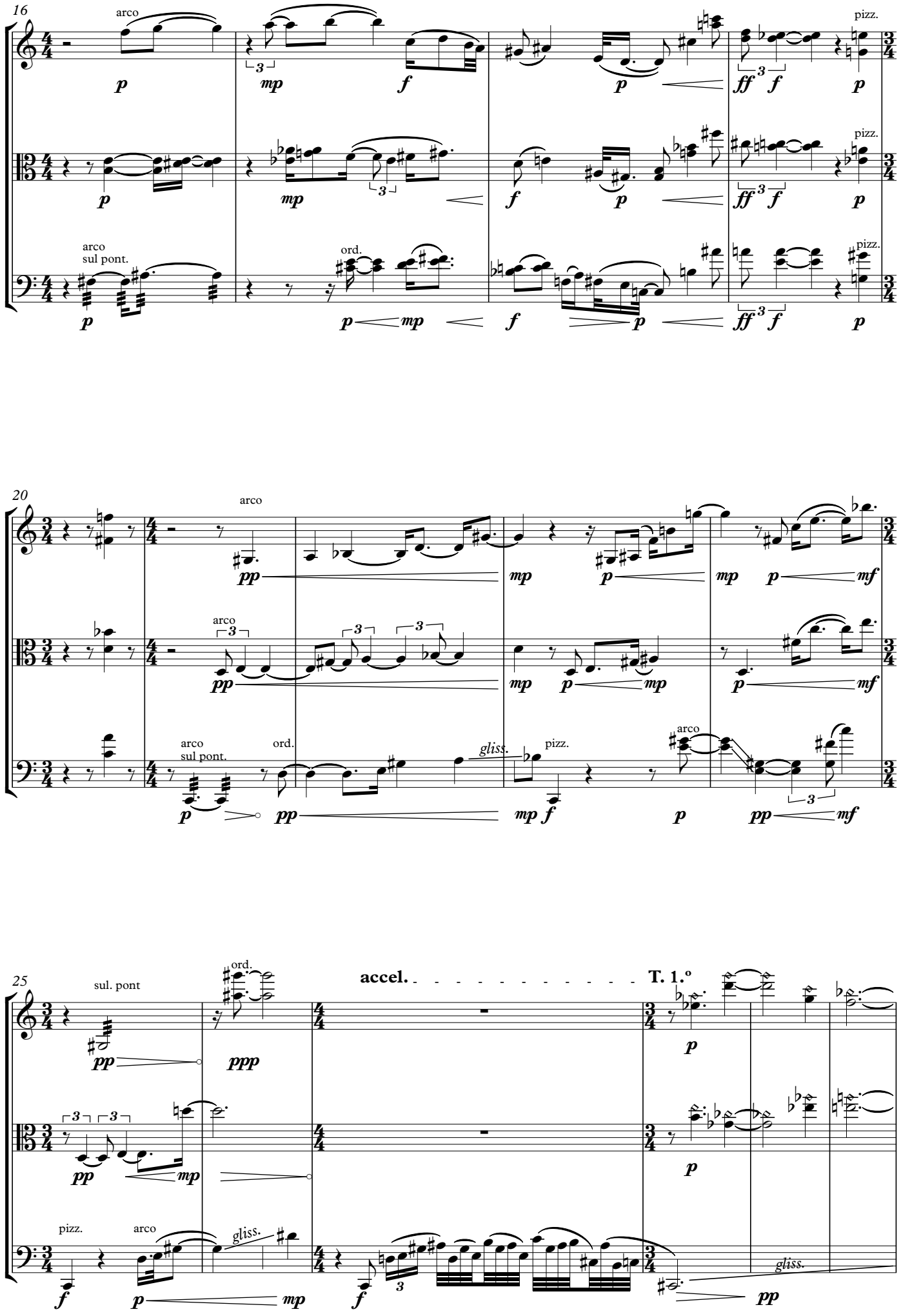

279
Alto Sax.
Ten. Sax.
Ten. Sax.
Bari. Sax.
Tpt.
Tpt.
Tbn.
Tbn.
Gtr.
Pno.
Bass
Dr.
G. P.
p
mp
pp
ppp
3

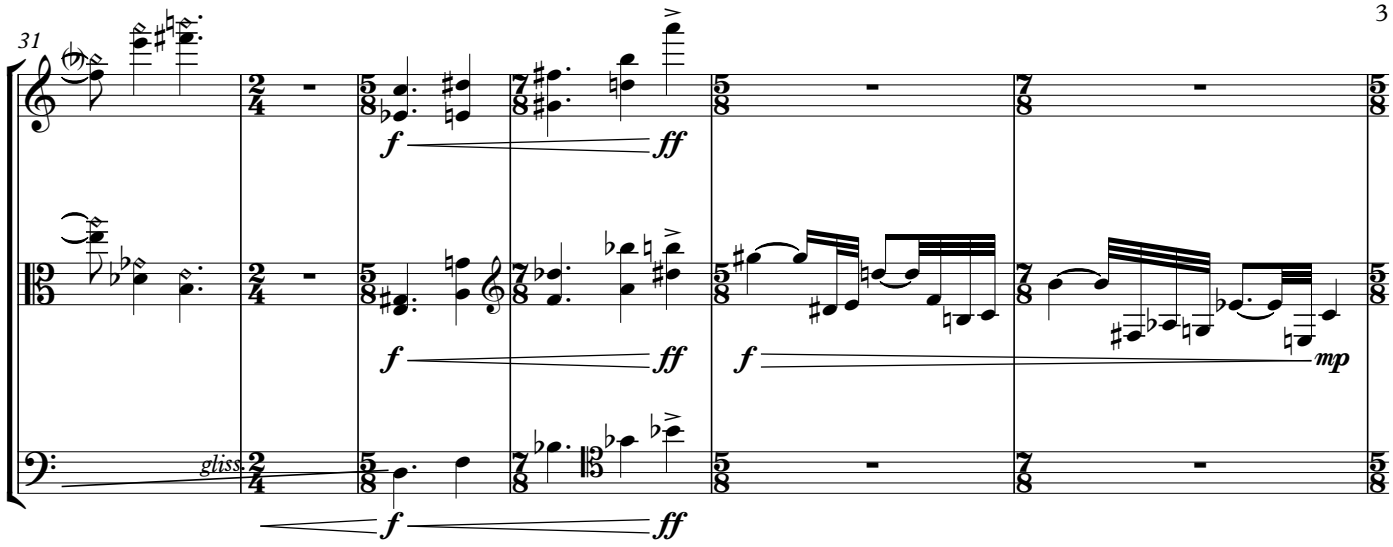
4.4 Efémero, para trio de cordas

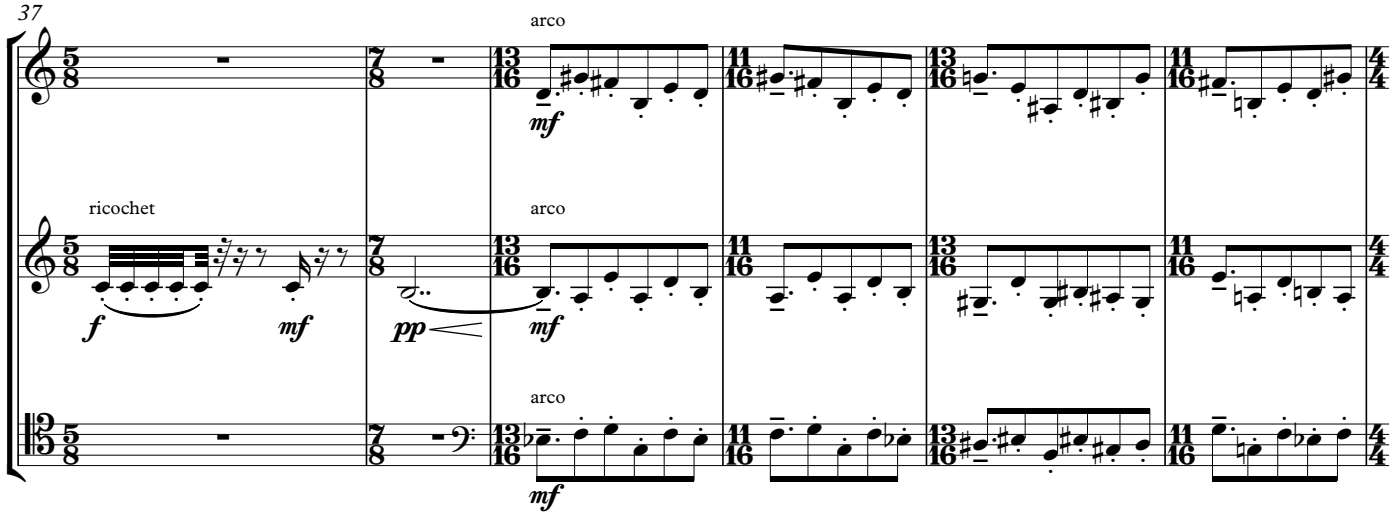
Efémero

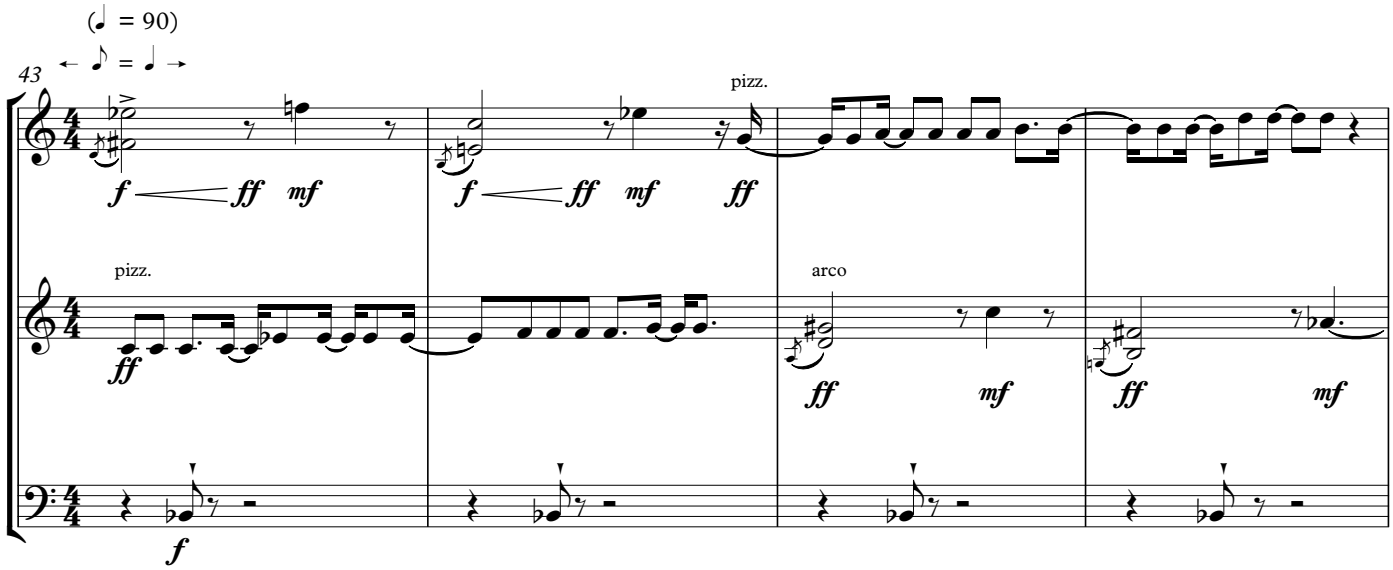
Nuno Trocado

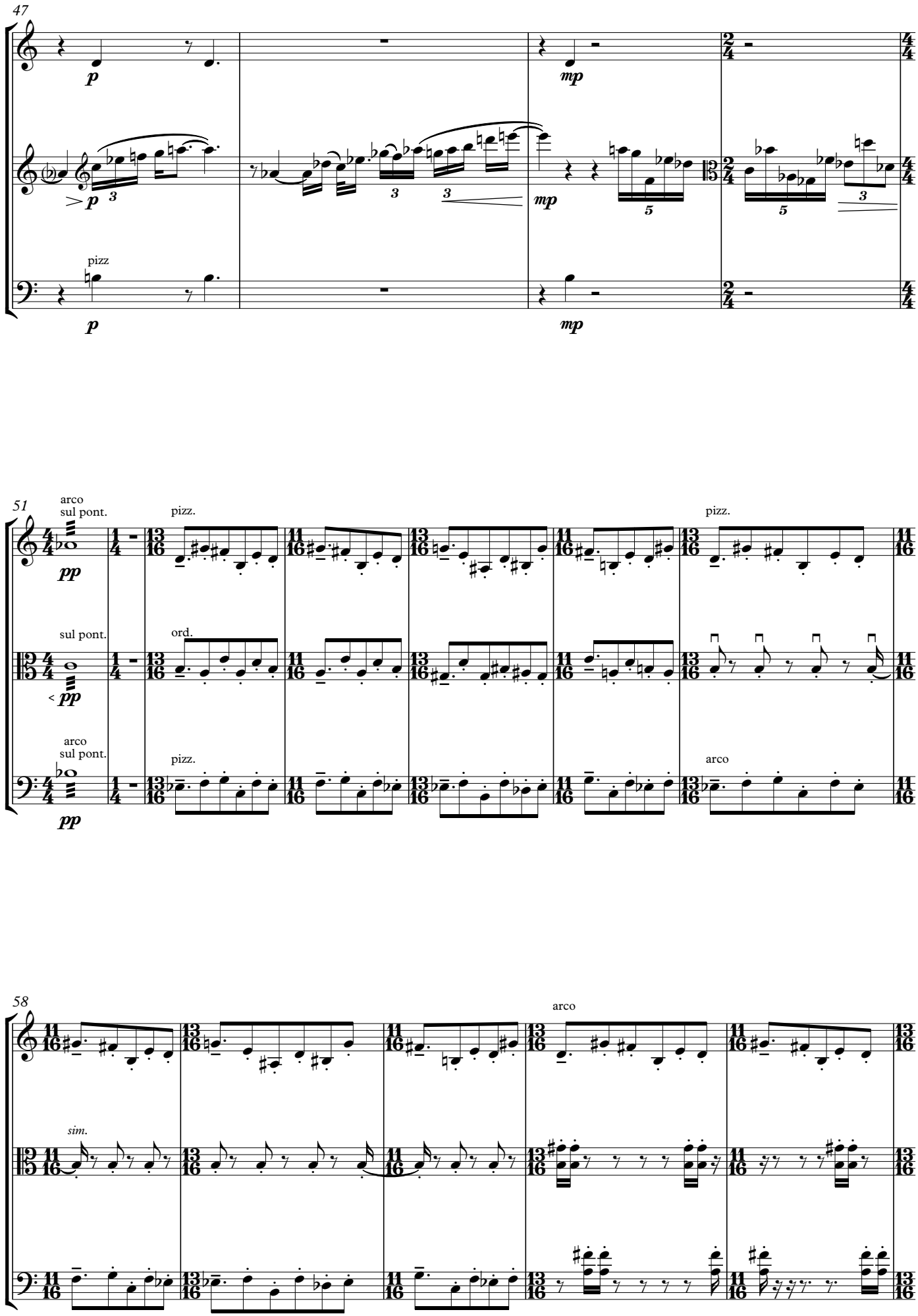


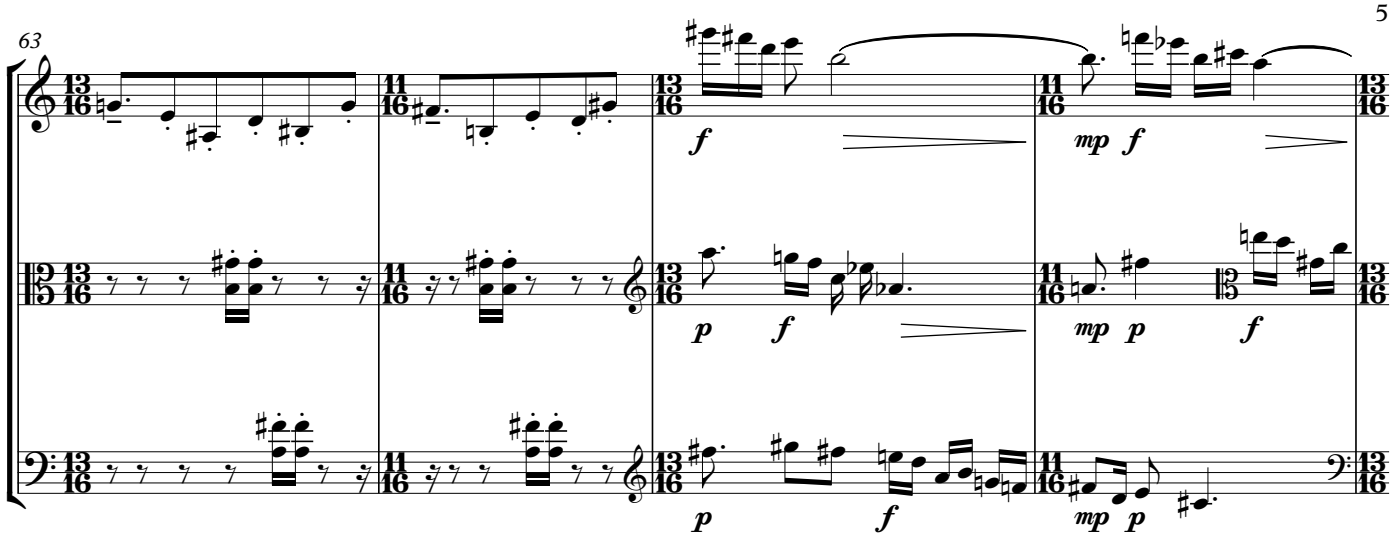

16
arco
pizz.
arco
sul pont.
ord.
20
arco
sul pont.
ord.
gliss.
pizz.
25
sul. pont
ord.
accel.
T. 1.º
pizz.
arco
gliss.

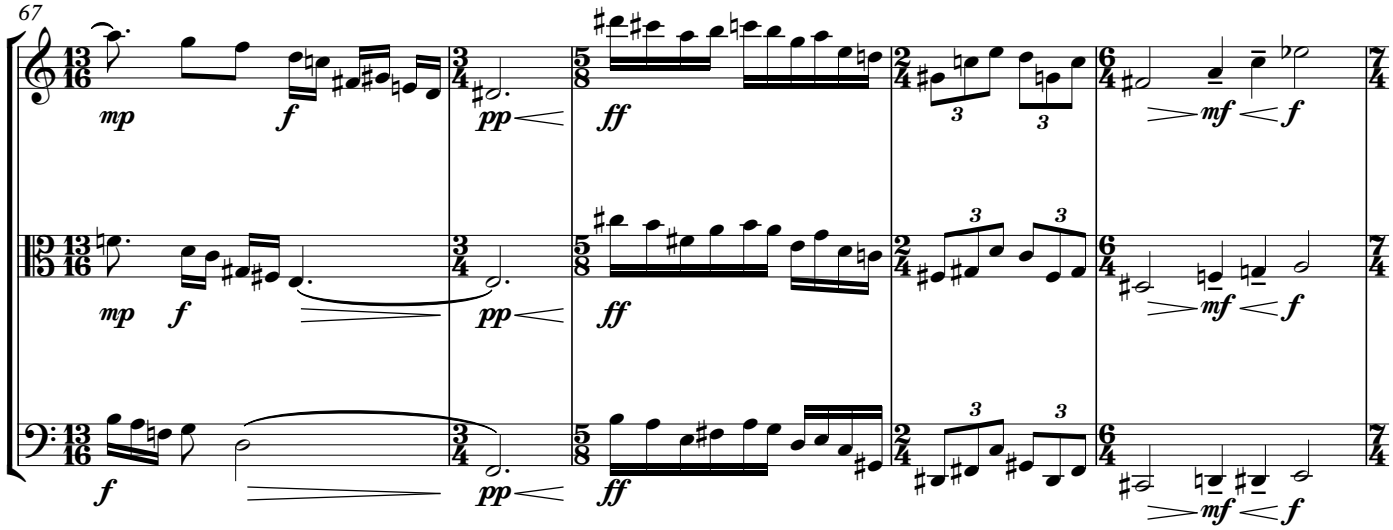

31
gliss.
f
ff
mp

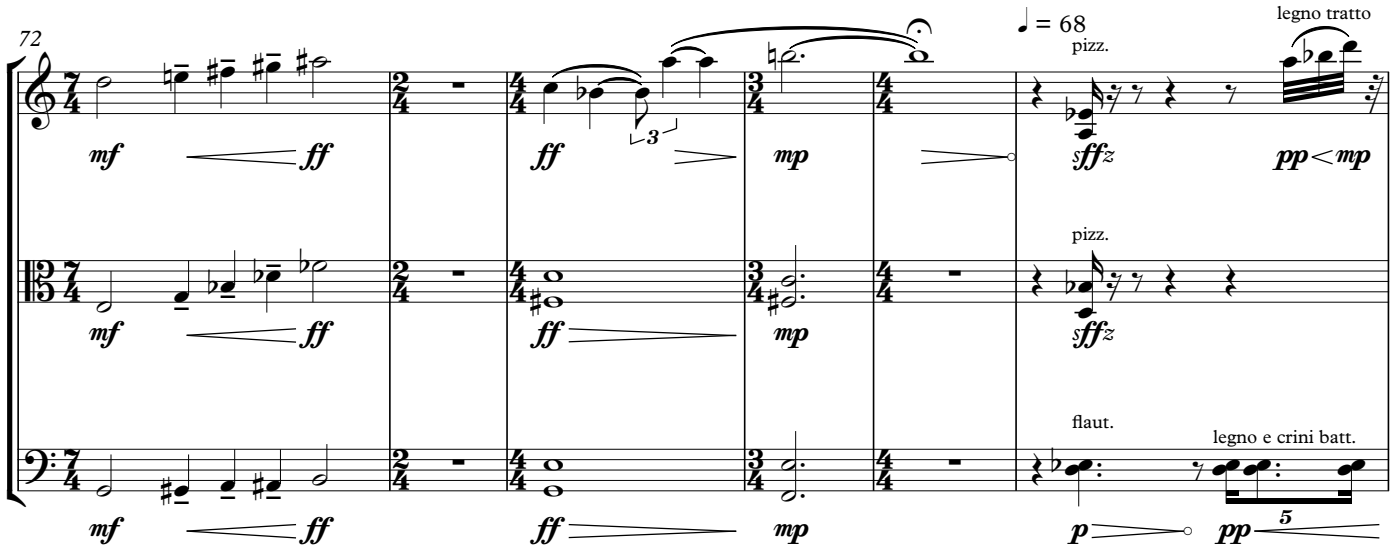

37
arco
ricochet
f
mf
pp

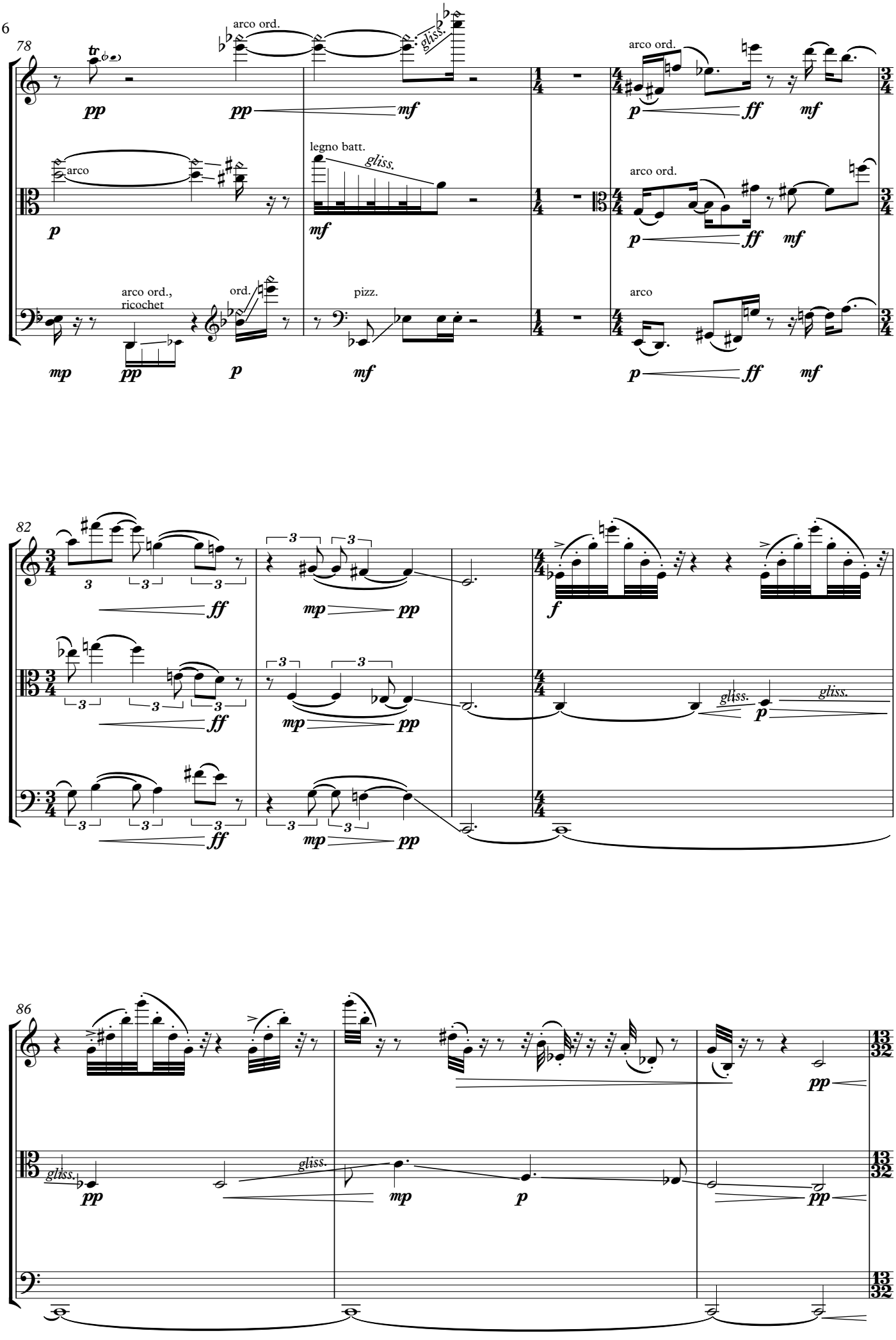

(♩ = 90)
← ♪ = ♩ →
43
pizz.
arco
ff
mf
f

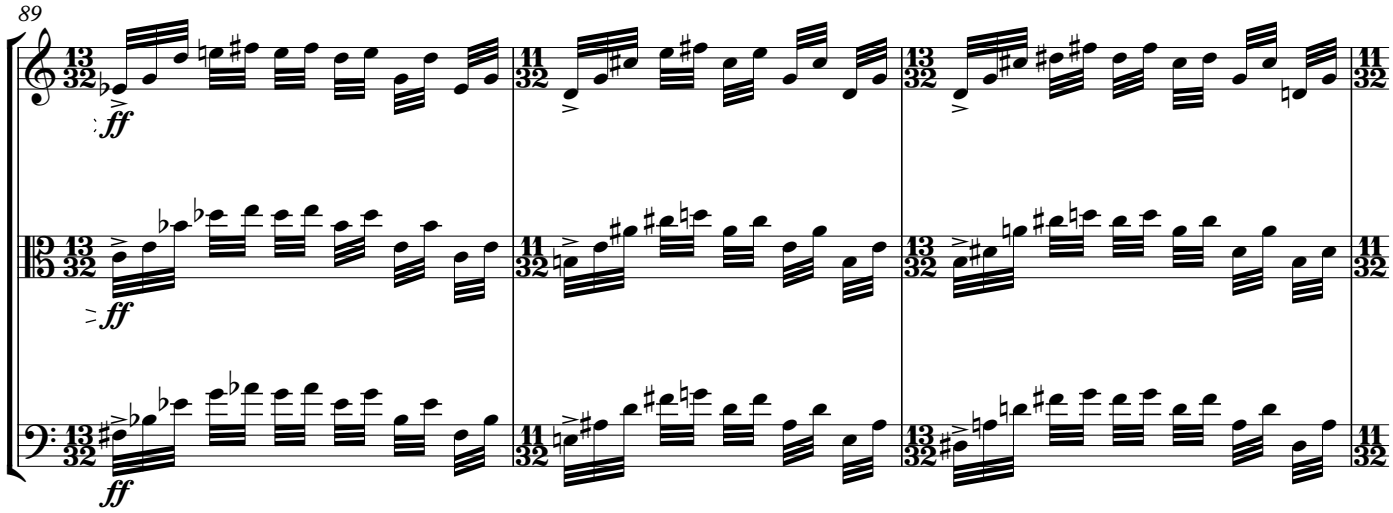

47
p
mp
pizz
51
arco
sul pont.
pizz.
sul pont.
ord.
arco
sul pont.
pizz.
arco
58
arco
sim.

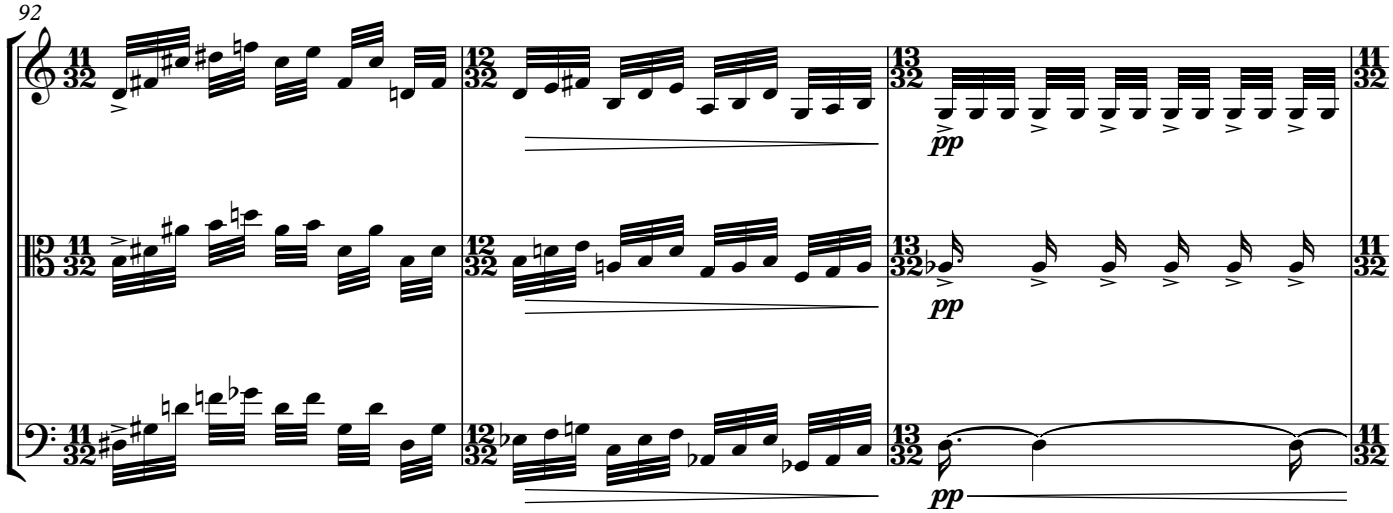

63
f
mp f
p f
mp p f
p f
mp p

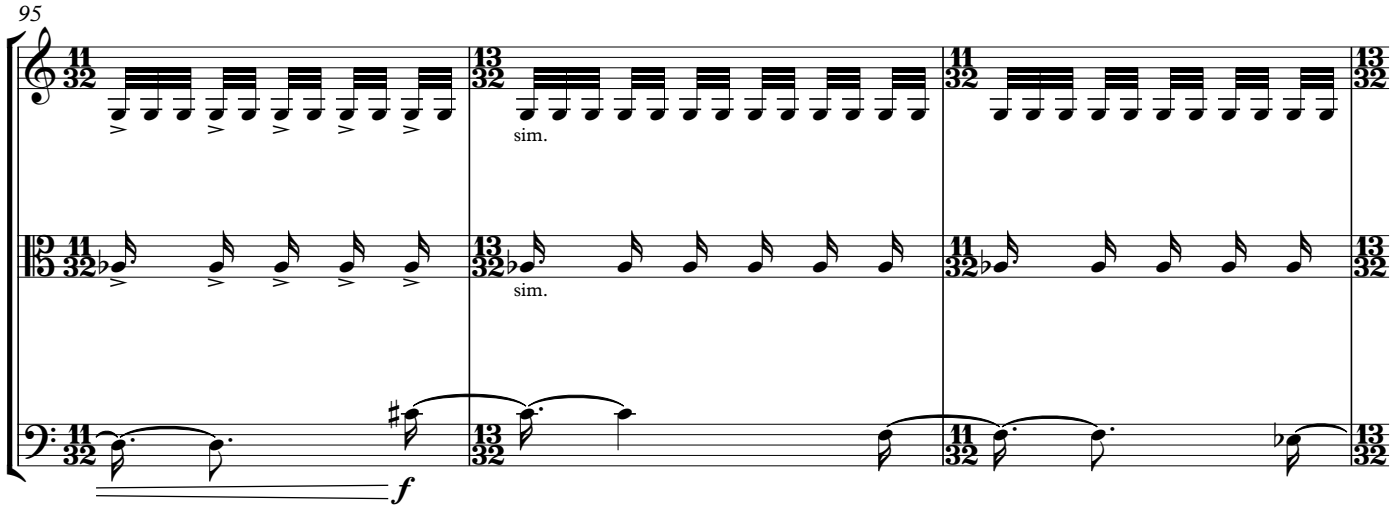

67
mp f
pp
ff
mf f
mp f
pp
ff
mf f
f
pp
ff
mf f

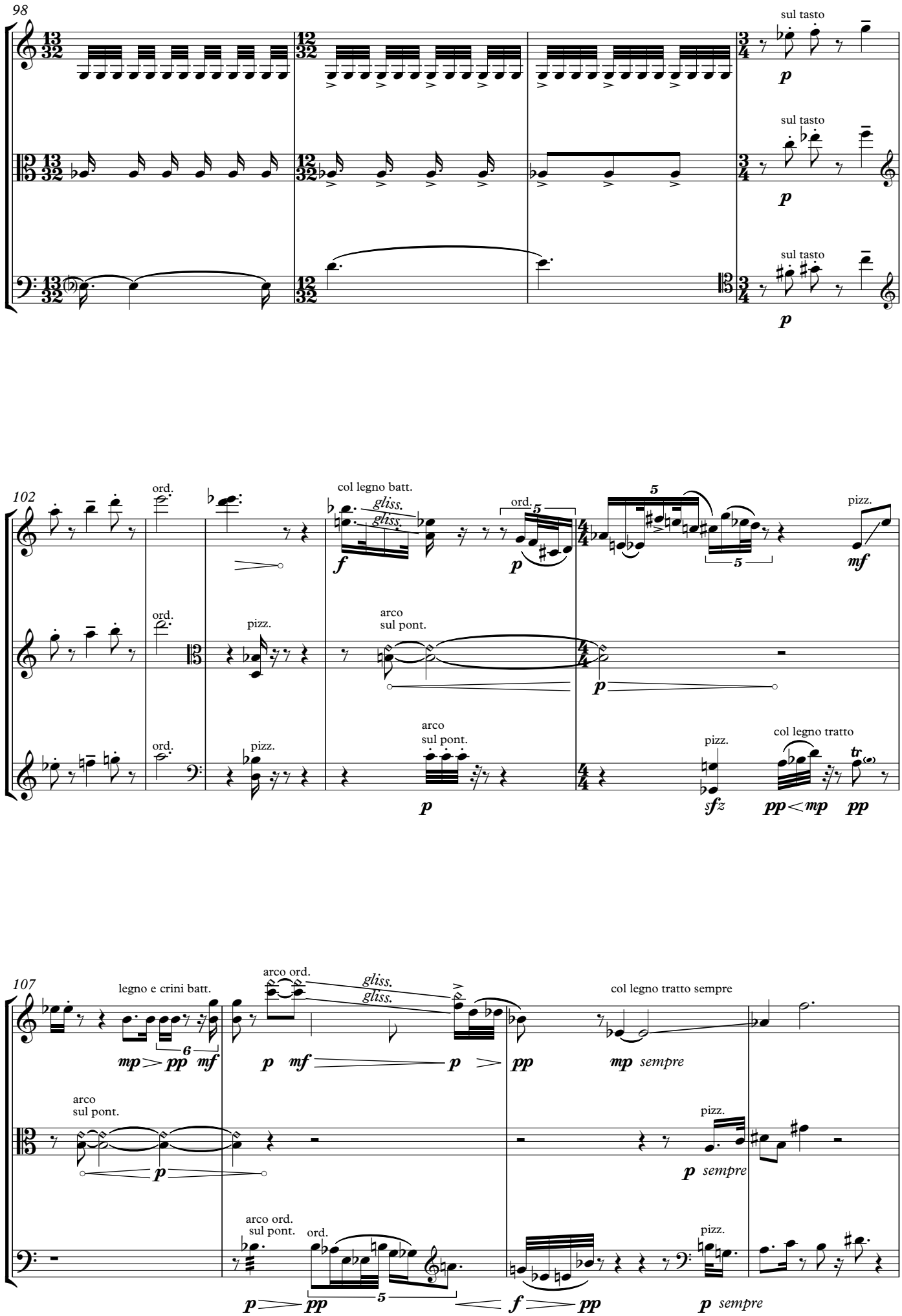

72
mf ff
ff
mp
♩ = 68
pizz.
sffz
pp mp
legno tratto
pizz.
mf ff
ff
mp
sffz
flaut.
legno e crini batt.
mf ff
ff
mp
p
pp

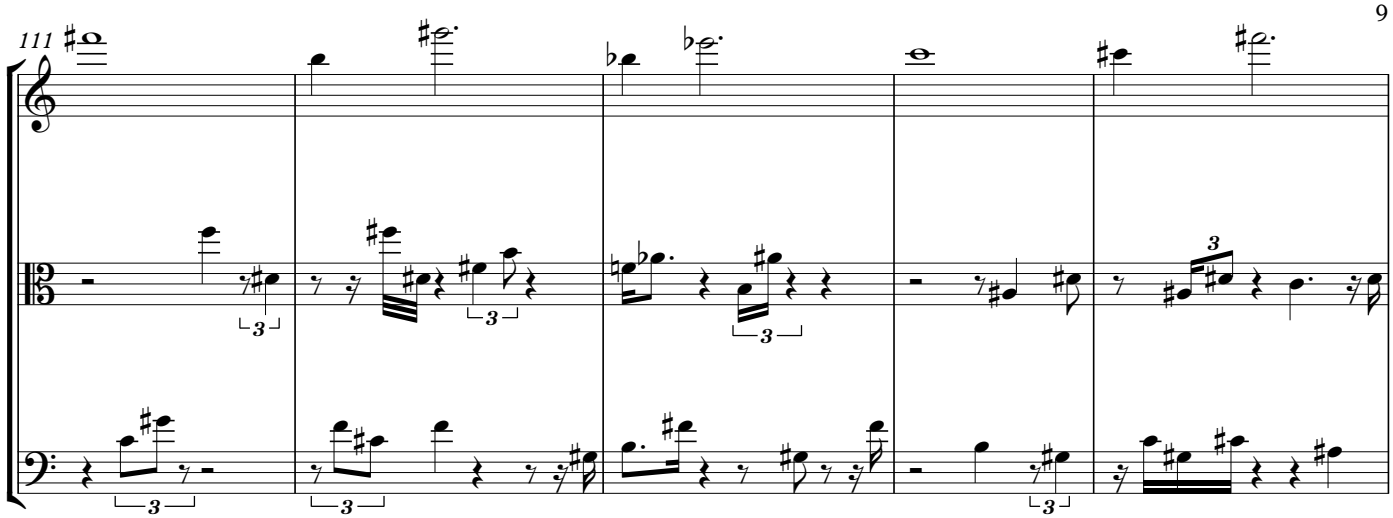

78
arco ord.
gliss.
arco ord.
arco
legno batt.
gliss.
arco ord.
arco ord., ricochet
ord.
pizz.
arco
82
gliss.
gliss.
86
gliss.
gliss.

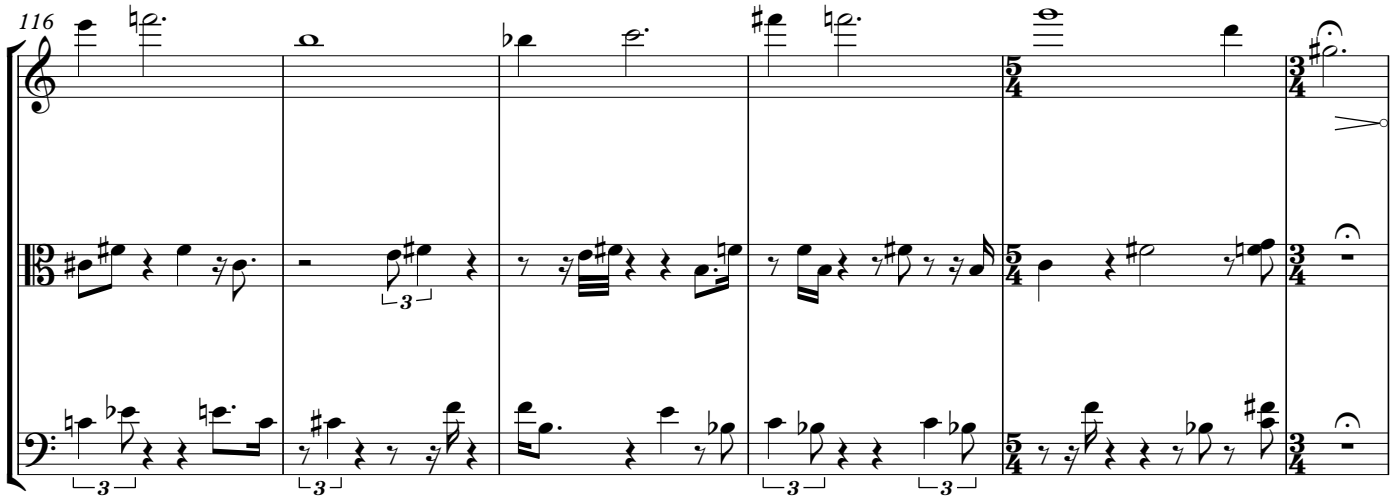

89
ff
ff
ff

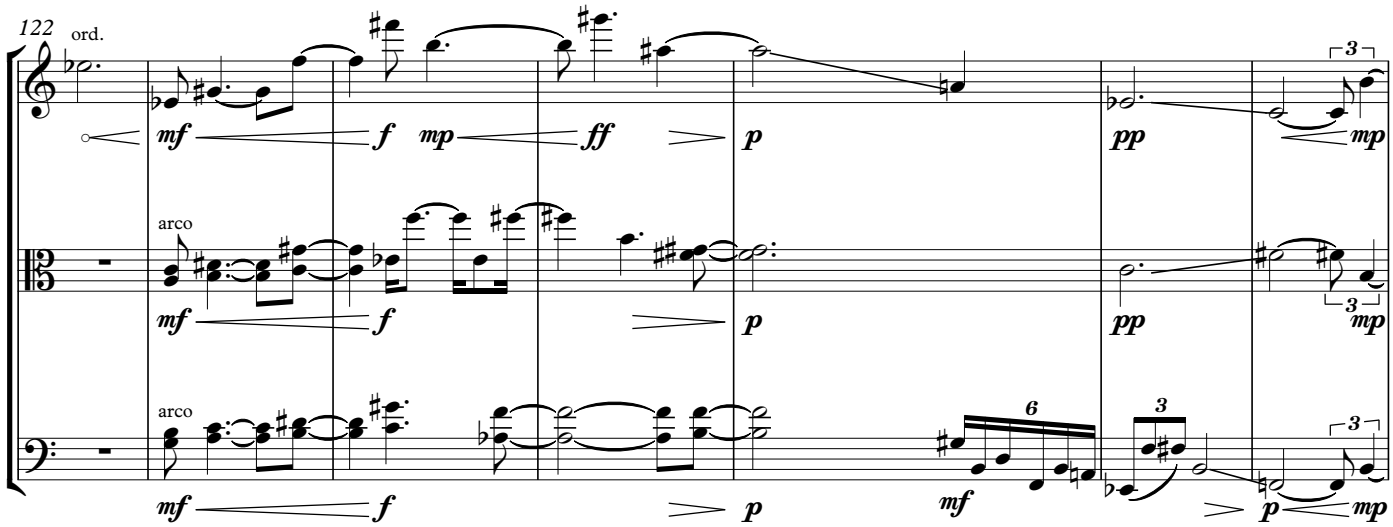

92
pp
pp
pp

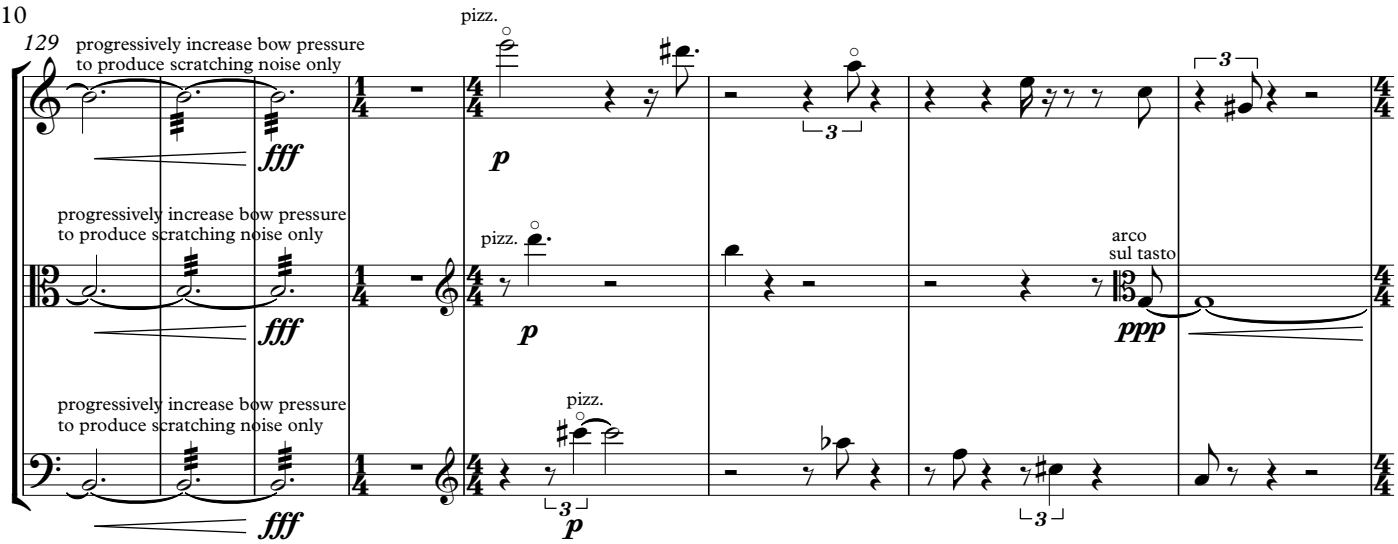

95
sim.
sim.
f

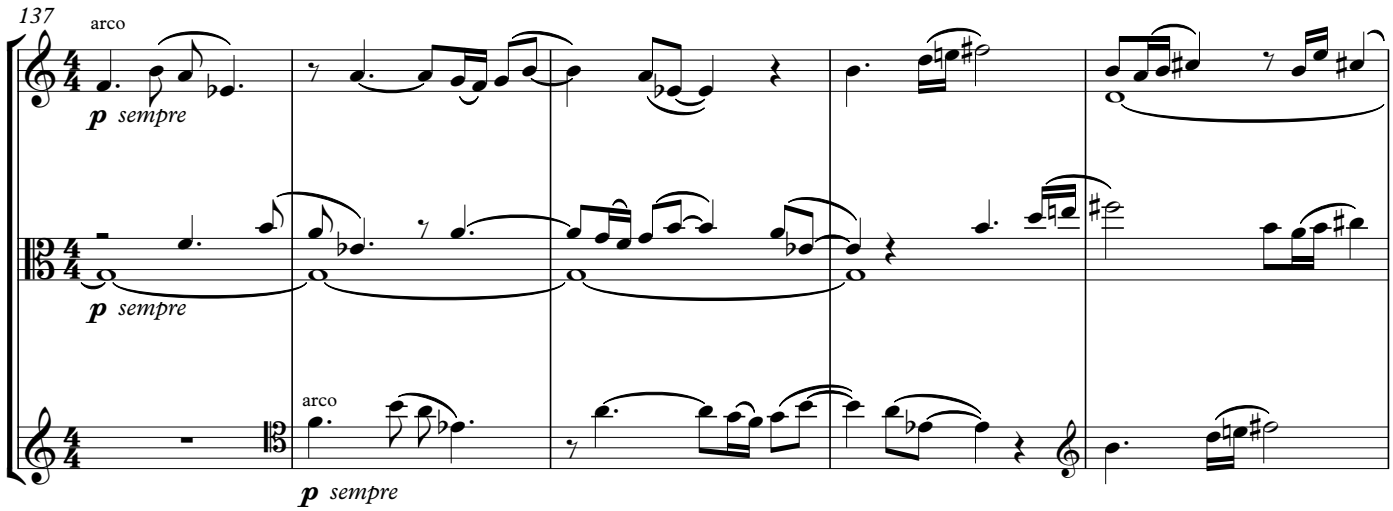

98
sul tasto
sul tasto
sul tasto
102
ord.
col legno batt.
gliss.
gliss.
ord.
pizz.
arco
sul pont.
col legno tratto
107
legno e crini batt.
arco ord.
gliss.
gliss.
col legno tratto sempre
arco
sul pont.
pizz.
sempre
arco ord.
sul pont.
ord.
pizz.
sempre

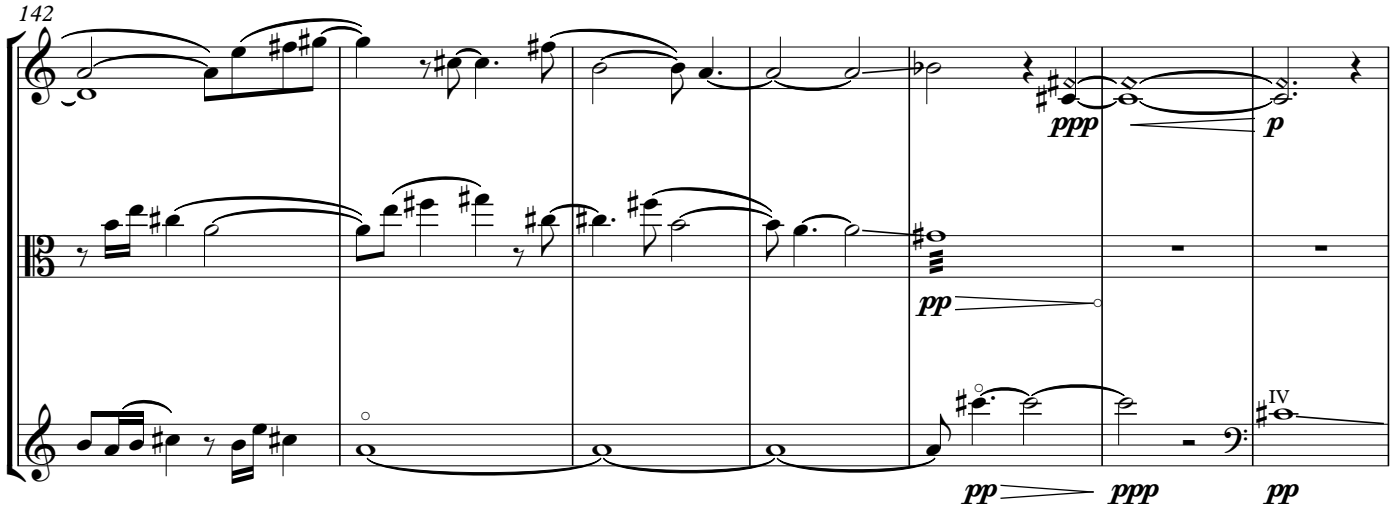

111

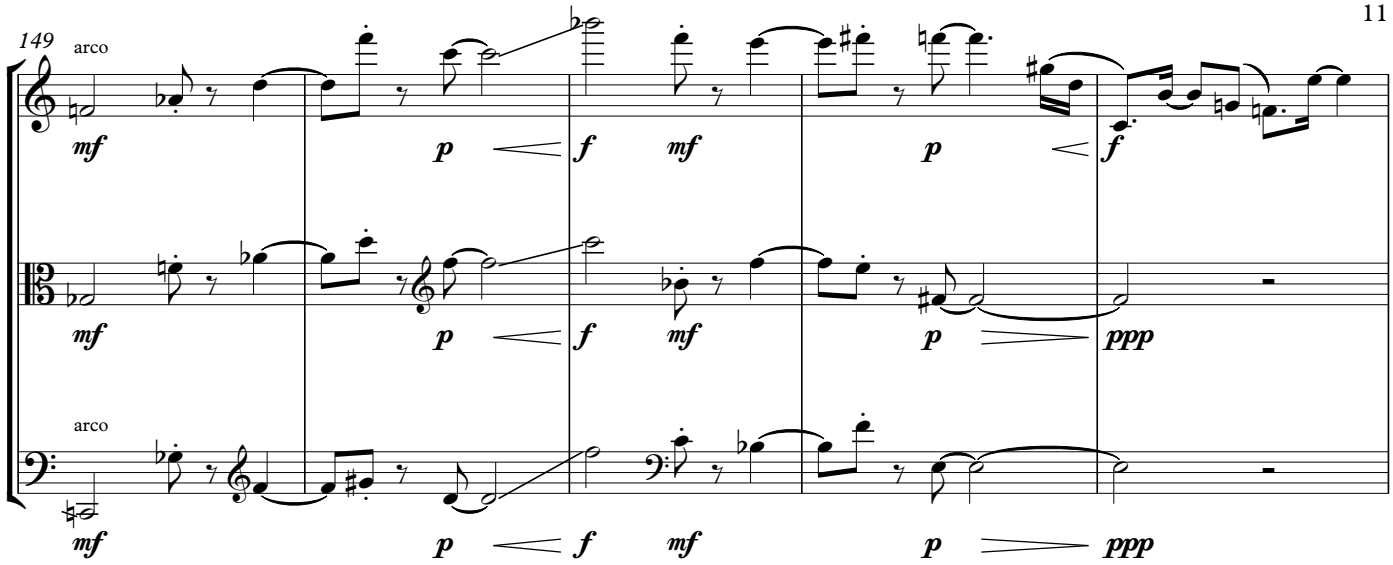

116

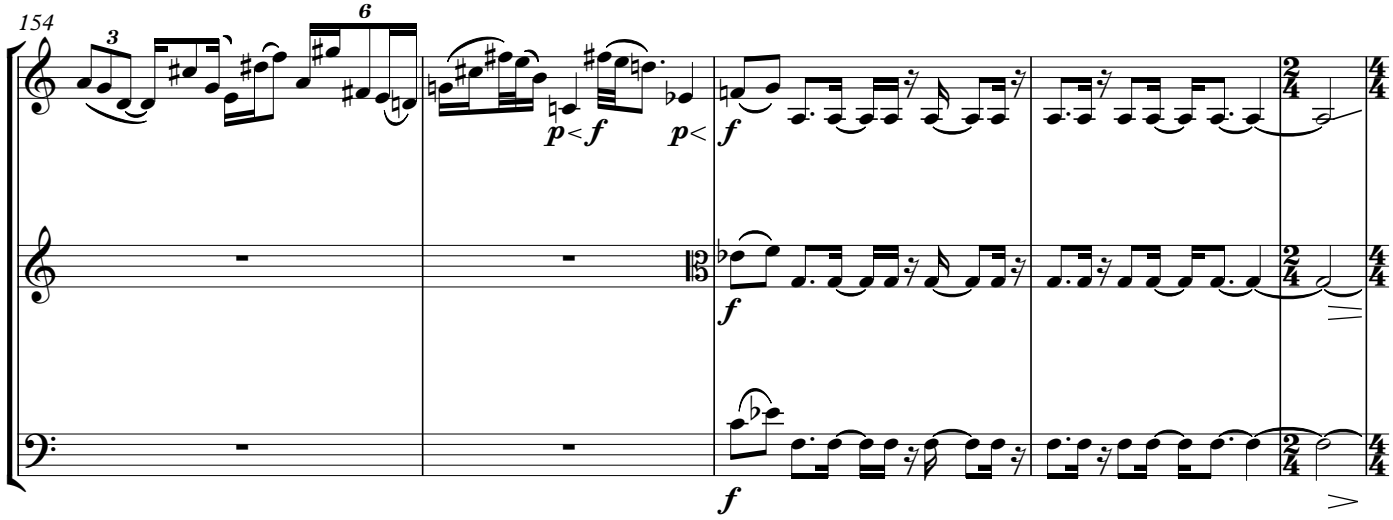

122
ord.
arco
arco

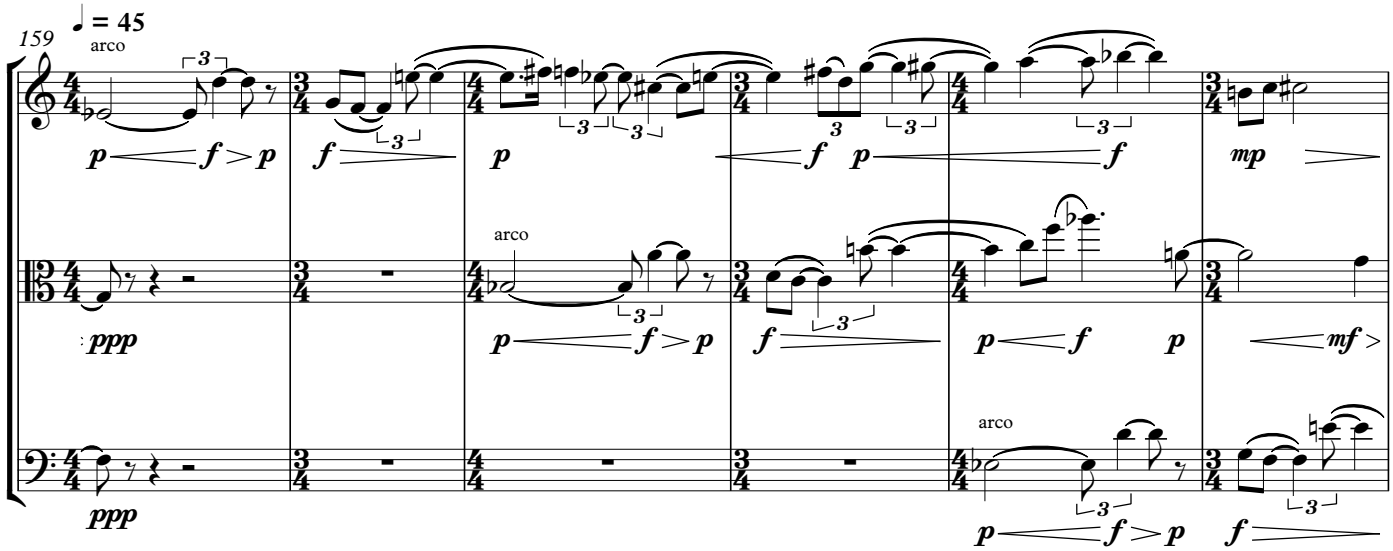

129
progressively increase bow pressure
to produce scratching noise only
fff
pizz.
p
3
progressively increase bow pressure
to produce scratching noise only
fff
pizz.
p
arco
sul tasto
ppp
progressively increase bow pressure
to produce scratching noise only
fff
pizz.
3
p
3

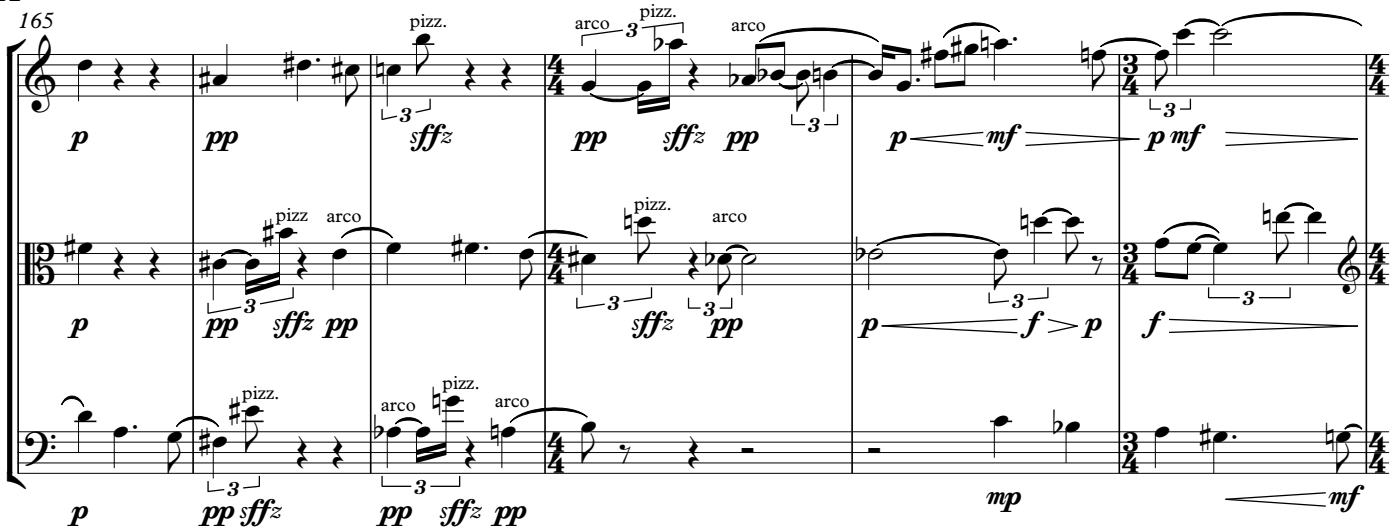

137
arco
p sempre
p sempre
arco
p sempre

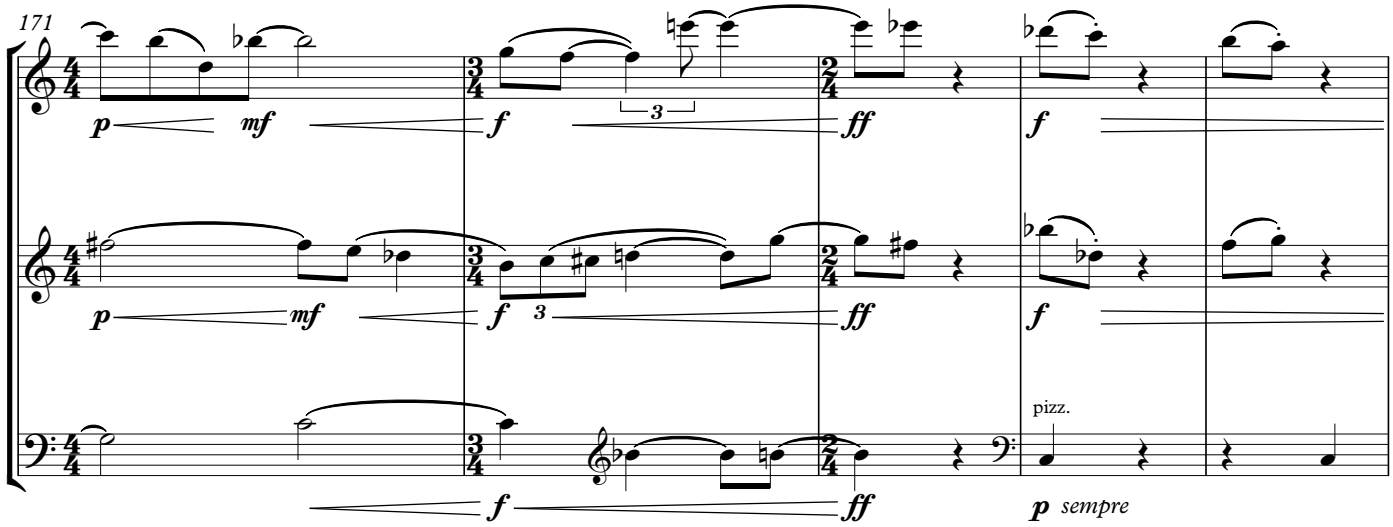

142
ppp
p
pp
pp
ppp
IV
pp

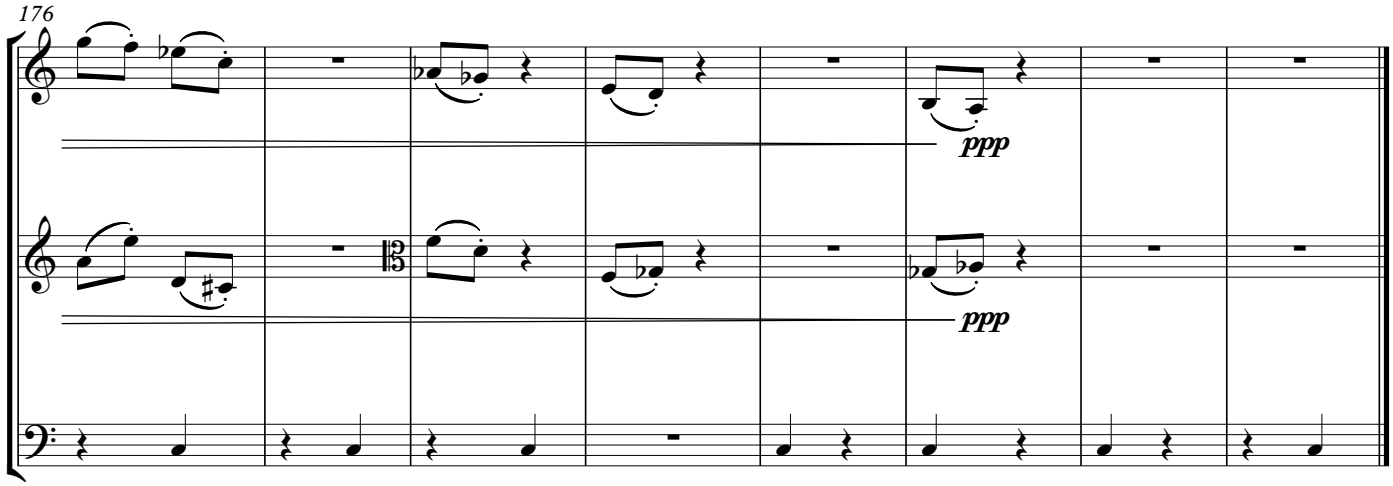

149
arco
mf
p
f
mf
p
f
mf
p
f
mf
p
ppp
arco
mf
p
f
mf
p
ppp


154
p < f
p <
f
f
f


159
♩ = 45
arco
p
f
p
f
p
f
p
f
p
f
p
mp
arco
ppp
p
f
p
f
p
f
p
mf
arco
ppp
p
f
p
f


165
pizz.
arco
pizz.
arco
pizz.
arco
pizz.
arco
pizz.
arco
pizz.
arco
p
pp
sffz
pp
sffz
pp
p
mf
p
mf
p
pp
sffz
pp
pp
sffz
pp
p
f
p
f
p
pp
sffz
pp
sffz
pp
mp
mf


171
p
mf
f
ff
f
p
mf
f
ff
f
pizz.
f
ff
p sempre


176
ppp
ppp

— 5 —

Conclusões

Através da composição algorítmica assistida por computador logrei expandir a minha imaginação musical, alcançando ideias que não me surgiriam nem seriam possíveis de realizar de outro modo.

Em simultâneo, desenvolvi um conjunto de ferramentas que passam a integrar o arsenal de recursos disponíveis para trabalhos futuros, e que modestamente coloco à disposição da comunidade.

A exploração algorítmica de conceitos teóricos conduziu à optimização da técnica composicional e do vocabulário individual.

Entendo que não me compete pronunciar-me sobre a valia estética das obras aqui apresentadas. Porém, não tenho dúvidas de que sai confirmada a viabilidade da escrita musical fundada numa conjugação de esforços entre o ser humano e a máquina.

Sobeja ainda espaço para desenvolvimentos ulteriores das técnicas aqui exploradas, maximizando a respectiva utilidade na criação musical.

Referências Bibliográficas

Agostini, A., & Ghisi, D. (2013). Real-time computer-aided composition with bach. *Contemporary Music Review*, *32*(1), 41–48. doi:10.1080/074944627.2013.774221

Agostini, A., & Ghisi, D. (2015). A Max library for musical notation and computer-aided composition. *Computer Music Journal*, *39*(2), 11–27. doi:10.1162/COMJ_a_00296

Anders, T., & Miranda, E. R. (2009). Interfacing manual and machine composition. *Contemporary Music Review*, *28*(2), 133–147. doi:10.1080/07494460903322422

Anders, T., & Miranda, E. R. (2011). Constraint programming systems for modeling music theories and composition. *ACM Computing Surveys*, *43*(4), 30:1–30:38. doi:10.1145/1978802.1978809

Andrikopoulos, D. (2013). *A portfolio of compositions* (tese de doutoramento, University of Birmingham). Obtido de http://etheses.bham.ac.uk/4704/1/Andrikopoulos13PhD.pdf

Ariza, C. (2005). *An open design for computer-aided algorithmic music composition*. Irvine, California, EUA: Universal Publishers. Obtido de https://books.google.pt/books?id=XukW-mq76mcC

Arom, S. (2004). *African polyphony and polyrhythm: musical structure and methodology*. Cambridge, Reino Unido: Cambridge University Press. Obtido de https://books.google.pt/books?id=cuFu7ipCl5kC

Asmussen, S. (2003). *Applied probability and queues* (2ª ed.). Nova Iorque, EUA: Springer.

Assayag, G., Rueda, C., Laurson, M., Agon, C., & Delerue, O. (1999). Computer-sssisted composition at IRCAM: From PatchWork to OpenMusic. *Computer Music Journal*, *23*(3), 59–72. doi:10.2307/3681240

Block, S., & Douthett, J. (1994). Vector products and intervallic weighting. *Journal of Music Theory*, *38*(1), 21–41.

Boden, M. (2009). Computer models of creativity. *AI Magazine*, *30*, 23–34.

Borges, J. L. (1944). La biblioteca de Babel. Em *Ficciones*. Buenos Aires, Argentina: Editorial Sur.

Brown, A. R. (2000). Modes of compositional engagement. Em *Proceedings of the Interfaces: The australasian computer music conference* (pp. 8–17). Brisbane, Austrália: Australian Computer Music Association.

Brualdi, R. (2010). *Introductory combinatorics* (5ª ed.). Londres, Reino Unido: Pearson Education.

Chemillier, M. (2002). Ethnomusicology, ethnomathematics. The logic underlying orally transmitted artistic practices. Em *Mathematics and music* (pp. 161–183). Berlim, Alemanha: Springer.

Chemillier, M. (2004). Periodic musical sequences and Lyndon words. *Soft Computing*, *8*(9), 611–616. doi:10.1007/s00500-004-0387-2

Chemillier, M., & Truchet, C. [Charlotte]. (2003). Computation of words satisfying the ”rhythmic oddity property”(after Simha Arom’s works). *Information Processing Letters*, *86*(5), 255–261. doi:10.1016/S0020-0190(02)00521-5

Chen, C., Koh, K., & Khee-Meng, K. (1992). *Principles and Techniques in Combinatorics*. Singapura: World Scientific.

Computer aided. (s.d.). Wikipedia. Obtido 12 setembro 2017, de https://en.wikipedia.org/wiki/Computer-aided

Cramer, F. (2002). Concepts, notations, software, art. Obtido de http://cramer.pleintekst.nl/all/concept_notations_software_art/concepts_notations_software_art.html

Durkin, A. (2014). *Decomposition: A music manifesto*. Nova Iorque, EUA: Pantheon Books.

Duval, J.-P. (1988). Génération d’une section des classes de conjugaison et arbre des mots de Lyndon de longueur bornée. *Theoretical Computer Science*, *60*, 255–283.

Fernández, J. D., & Vico, F. (2013). AI methods in algorithmic composition: A comprehensive survey. *Journal of Artificial Intelligence Research*, *48*(1), 513–582. Obtido de http://dl.acm.org/citation.cfm?id=2591248.2591260

Galeazzi, F. (2012). *The theoretical-practical elements of music, parts III and IV* (D. Burton, & G. Harwood, Trad.). Champaign, Illinois, EUA: University of Illinois Press. Obtido de https://books.google.pt/books?id=JrIQihkXTMkC. (Trabalho original publicado em 1796)

Gato, G. (2016). *Algorithm and decision in musical composition* (tese de doutoramento, Guildhall School of Music e Drama, Londres, Reino Unido). Obtido de http://www.goncalogato.com/wordpress/wp-content/uploads/2017/02/Algorithm-and-Decision-in-Musical-Composition-Gon%C3%A7alo-Gato.pdf

Hall, R. W., & Klingsberg, P. (2004). Asymmetric rhythms, tiling canons, and Burnside’s lemma. Em *Bridges: Mathematical connections in art, music and science*. Obtido de http://archive.bridgesmathart.org/2004/bridges2004-189.pdf

Hedges, S. A. (1978). Dice music in the eighteenth century. *Music and Letters*, *59*(2), 180–187. doi:10.1093/ml/59.2.180

Hill, R. K. (2016). What an algorithm is. *Philosophy & Technology*, *29*(1), 35–59. doi:10.1007/s13347-014-0184-5

Hiller, L. A., Jr., & Isaacson, L. M. (1958). Musical composition with a high-speed digital computer. *Journal of the Audio Engineering Society*, *6*(3), 154–160.

Houaiss, A. (2001). *Dicionário Houaiss da Língua Portuguesa*. Rio de Janeiro, Brasil: Editora Objetiva.

Jedrzejewski, F. (2016). Counting words satisfying the rhythmic oddity property. *ArXiv e-prints*. arXiv: 1607.07175

Jones, D., Brown, A. R., & d'Inverno, M. (2012). The extended composer. Em *Computers and creativity*. Berlim, Alemanha: Springer. doi:10.1007/978-3-642-31727-9_7

Lambert, J. P. (1990). Interval cycles as compositional resources in the music of Charles Ives. *Music Theory Spectrum*, *12*(1), 43–82.

Laske, O. (1981). Composition theory in Koenig's Project One and Project Two. *Computer Music Journal*, *5*(4), 54–65.

Laurson, M. (1999). Recent developments in patchwork: PWConstraints-a rule based approach to complex musical problems. Em *Symposium on systems research in the arts, Baden-Baden*. Obtido de http://www2.siba.fi/soundingscore/PDF/PWConstraints.pdf

Laurson, M., Kuuskankare, M., & Kuitunen, K. (2005). Introduction to computer-assisted music analysis in PWGL. Em *Proceedings of the sound and music conference 2005, Salerno, Italy*. Obtido de http://smc.afim-asso.org/smc05/papers/MikaelLaurson/intro-analysis-final-phr.pdf

Laurson, M., Kuuskankare, M., & Norilo, V. (2009). An overview of PWGL, a visual programming environment for music. *Computer Music Journal*, *33*(1), 19–31. doi:10.1162/comj.2009.33.1.19

London, J. (2002). Non-isomorphisms between pitch and time. *Journal of Music Theory*, *1/2*.

Messiaen, O. (1944). *The technique of my musical language*. Paris, França: Alphonse Leduc.

Miranda, E. R., Manzolli, J., & Maia Jr., A. (2005). Granular synthesis of sounds through fuzzyfied Markov chains. Em *Proceedings of IX National Convention of the Audio Engineering Society*. São Paulo, Brasil. Obtido de https://www.academia.edu/2910494/Granular_Synthesis_of_Sounds_through_Fuzzyfied_Markov_Chains

Monahan, C. B., & Carterette, E. C. (1985). Pitch and duration as determinants of musical space. *Music Perception*, *3*, 1–32.

Moschovakis, Y. N. (2001). What is an algorithm? Em B. Engquist, & W. Schmid (Eds.), *Mathematics unlimited – 2001 and beyond* (pp. 919–936). Berlim, Alemanha: Springer.

Nierhaus, G. (2009). *Algorithmic composition: Paradigms of automated music generation*. Viena, Áustria: Springer.

Noguchi, H. (1990). Mozart: Musical game in C K.516f. *Mitteilungen der Internationalem Stiftung Mozarteum*, *38*. Obtido de http://www.asahi-net.or.jp/~rb5h-ngc/e/k516f.htm

Palisca, C. V. (1980). *Guido of Arezzo*. Em S. Sadie (Ed.), *The new Grove dictionary of music and musicians* (Vol. 7, pp. 803–807). Londres, Reino Unido: Macmillan Publishers.

Peretz, I., & Kolinsky, R. (1993). Boundaries of separability between melody and rhythm in music discrimination: A neuropsychological perspective. *The Quarterly Journal of Experimental Psychology*, *46A*, 301–325.

Perle, G. (1977). Berg's master array of the interval cycles. *The Musical Quarterly*, *63*(1), 1–30.

Pinkerton, R. C. (1956). Information theory and melody. *Scientific American*, *194*(2), 77–87. doi:10.1038/scientificamerican0256-77

Pressing, J. (1983). Cognitive isomorphisms between pitch and rhythm in world musics: West Africa, the Balkans and western tonality. *Studies in Music*, *17*, 38–61.

Prince, J. B., & Schmuckler, M. A. (2014). The tonal-metric hierarchy: A corpus analysis. *Music Perception*, 24–270.

Rahn, J. (1983). *A theory for all music*. Toronto, Canadá: University of Toronto Press.

Rameau, J.-P. (1722). *Traité de l'harmonie réduite à ses principes naturels*. Paris, França: Jean-Baptiste-Christophe Ballard.

Roads, C. (2015). *Composing electronic music: A new aesthetic*. Oxford, Reino Unido: Oxford University Press.

Rossi, F., van Beek, P., & Walsh, T. (2008). Constraint programming. Em F. van Harmelen, V. Lifschitz, & B. Porter (Eds.), *Handbook of knowledge representation* (pp. 181–211). Amsterdão, Países Baixos: Elsevier. doi:10.1016/S1574-6526(07)03004-0

Sandred, Ö. (2017). *The musical fundamentals of computer assisted composition*. Winnipeg, Canadá: Audiospective Media.

Sandred, Ö., Laurson, M., & Kuuskankare, M. (2009). Revisiting the Illiac Suite - A rule-based approach to stochastic processes. *Sonic Ideas/Ideas Sonicas*, *2*, 42–46.

Sethares, W. A. (2005). *Tuning, timbre, spectrum, scale*. Londres, Reino Unido: Springer. doi:10.1007/b138848

Simoni, M. (2003). *Algorithmic composition: A gentle introduction to music composition using common lisp and common music*. Ann Arbor, EUA: The Scholarly Publishing Office, University of Michigan. doi:10.3998/spobooks.bbv9810.0001.001

Simpson, E. H. (1949). Measurement of diversity. *Nature*, *163*, 688. doi:10.1038/163688a0

Stoecker, P. (2014). Aligned cycles in Thomas Adès's Piano Quintet. *Musical Analysis*, *1*(33), 32–64. doi:10.1111/musa.12019

Strauss, J. N. (2005). *Introduction to post-tonal theory* (3ª ed.). Upper Saddle River, EUA: Pearson Prentice Hall.

Taylor, S. A. (2012). Hemiola, maximal evenness, and metric ambiguity in late Ligeti. *Contemporary Music Review*, *31*(2-3), 203–220. doi:10.1080/07494467.2012.717362

Tenney, J. (1988). *A history of 'consonance' and 'dissonance'* (E. M. P. Company, Ed.). Nova Iorque, EUA.

Toussaint, G. (2013). *The geometry of musical rhythm: What makes a "good" rhythm good?* Boca Raton, EUA: CRC Press.

Trapani, C. (2017). Computer-assisted composition with OpenMusic and bach. Obtido 1 outubro 2017, de http://christophertrapani.com/wordpresssite/computer-assisted-composition/

Travers, A. (2004). *Interval cycles, their permutations and generative properties in Thomas Adès's Asyla* (tese de doutoramento, University of Rochester).

Truchet, C. [C.], Assayag, G., & Codognet, P. (2003). OMClouds, a heuristic solver for musical constraints. Em *Proceedings of the international conference on metaheuristics*. Quioto, Japão.

Tymoczko, D. (2011). *A geometry of music: Harmony and counterpoint in the extended common practice*. Nova Iorque, EUA: Oxford University Press.

Vardi, M. (2012). What is an algorithm? *Communications of the ACM*, *55*(3). doi:10.1145/2093548.2093549

von Helmholtz, H. (1877). *Die Lehre von den Tonempfindungen als physiologische Grundlage für dir Theorie der Musik* (4ª ed.). Braunschweig, Alemanha: Friedrich Vieweg.

Wen, O. X., & Krumhansl, C. L. (2016). Isomorphism of pitch and time. Em B. Burger, J. Bamford, & E. Carlson (Eds.), *Proceedings of the 9th International Conference of Students of Systematic Musicology, Jyväskylä, Finland, 8th-10th June 2016*.

Xenakis, I. (1971). *Formalized music*. Bloomington, EUA: Indiana University Press.

— A —

Código

```
;;; ------------------
;;; GENERAL UTILITIES
;;; ------------------

(defun flatten (obj)
  (do* ((result (list obj))
        (node result))
       ((null node) (delete nil result))
    (cond ((consp (car node))
           (when (cdar node) (push (cdar node) (cdr node)))
           (setf (car node) (caar node)))
          (t (setf node (cdr node))))))

(defun sublistp (input)
  "Returns T if <input> is a list of lists."
  (when (and (listp input) (loop for i in input thereis (listp i))) t))

(defun remove-duplicate-sublists (list)
  (remove-duplicates list :test #'equal))

(defun list< (a b)
  "Returns true when the first element of list <a> is lower than the
  first element of list <b>"
  (cond ((null a) (not (null b)))
        ((null b) nil)
        ((= (first a) (first b)) nil)
        (t (< (first a) (first b))) ))

(defun list> (a b)
  "Returns true when the first element of list <a> is higher than the
  first element of list <b>"
  (cond ((null a) (not (null b)))
        ((null b) nil)
        ((= (first a) (first b)) nil)
        (t (> (first a) (first b)))))

(defun scale-value (value orig-min orig-max dest-min dest-max)
  "Scales <value> from an original to a destination range. If <value>, <orig-min> and <orig-max> are all the
      same, returns the lowest value of the destination bracket."
  (when (and (>= value orig-min)
             (<= value orig-max))
    (if (= value orig-min orig-max)
    dest-min
    (+ (/ (* (- value orig-min)
          (- dest-max dest-min))
           (- orig-max orig-min))
        dest-min))))

(defun binary-list (n &optional acc)
  "Accepts a non-negative integer, returns its binary representation in list form."
  ;; http://stackoverflow.com/questions/22668217/decimal-to-binary-in-lisp-make-a-non-nested-list
  (cond ((zerop n) (or acc (list 0)))
```

```
52          ((plusp n)
53           (binary-list (ash n -1) (cons (logand 1 n) acc)))
54          (t (error "~S: non-negative argument required, got ~s" 'binary-list n))))
55
56 (defun rotate (lst n)
57   ;; https://github.com/bbatsov/cl-99-problems/blob/master/p119.lisp
58   (let ((n-int (if (plusp n) n (- (length lst) (abs n)))))
59     (labels ((rotate* (lst index result)
60                (cond
61                  ((null lst) result)
62                  ((< index n-int) (rotate* (rest lst) (1+ index) (cons (first lst) result)))
63                  (t (append lst result)))))
64       (rotate* lst 0 nil))))
65
66 (defun random-no-repeats (bottom top size no-repeat-size)
67   "Returns a list of random numbers between <bottom> and <top>, with the length <size>.
68 Numbers will be unique in each subsequence of length <no-repeat-size>."
69   (when (> size no-repeat-size)
70     (loop :repeat size
71         :with new-last-x
72         :for last-x := nil :then new-last-x
73         :collect (loop :for nn := (+ (random (- top (- bottom 1))) bottom)
74             :do (setf new-last-x (if (< (length last-x) no-repeat-size)
75                                      (cons nn last-x)
76                                      (butlast (cons nn last-x))))
77             :until (unique-p new-last-x)
78             :finally (return nn)))))
79
80 ;;; ------------------
81 ;;; PWGL or OpenMusic
82 ;;; ------------------
83
84 (defun midi-cents (n)
85   "Converts a midi note number to the midicents format used by OpenMusic if this is being
86 run in that environment."
87   (if (find :om *features*)
88       (* n 100)
89       n))
90
91 ;;; ------------------
92 ;;; MUSICAL UTILITIES
93 ;;; ------------------
94
95 (defun p->i (pitch-list)
96   "Converts a list of pitches into a list of sequential intervals."
97   (loop :for (p q) :on pitch-list :while q :collect (- q p)))
98
99 (defun i->p (interval-list start)
100   "Converts a list of intervals into a list of pitches."
101   (loop :for i :in (push start interval-list) :sum i :into z :collect z))
102
103 (defun freq-to-midi (freq)
104   "Converts a pitch with frequency <freq> in Hz to midi cents."
```

```
105    (* (midi-cents 1) (+ 69 (* 12 (log (/ freq 440) 2)))))
106
107  (defun midi-to-freq (note)
108    "Converts a note in midi cents to frequency in Hz."
109    (* 440 (expt 2 (/ (- (/ note (midi-cents 1)) 69) 12))))
110
111  (defun durations-to-offsets (duration-list)
112    "Given a list of durations, returns a list of the corresponding offsets in ms."
113    (loop :for d :in duration-list
114        :sum d into total
115        :collect total into results
116        :finally (return (butlast (push 0 results)))))
117
118  (defun offsets-to-durations (offset-list)
119    "Given a list of offsets, returns a list of the corresponding durations in ms."
120    (loop :for (o1 o2) :on offset-list :while o2
121        :collect (- o2 o1)))
122
123  (defun harmonic-series (fundamental n-partials)
124    "Returns <n-partials> of the harmonic series starting on <fundamental> note in midi cents."
125    (let ((fundamental-freq (midi-to-freq fundamental)))
126      (loop :for p :from 1 :upto n-partials
127          :collect (freq-to-midi (* fundamental-freq p)))))
128
129  ;;; ----------------
130  ;;; TRANSFORMATIONS
131  ;;; ----------------
132
133  (defun one-rotation (chord &optional (interval (midi-cents 12)))
134    "Repeatedly transposes the lowest note of <chord>
135  up by an <interval> until it's the highest."
136    (let ((lowest (apply #'min chord))
137      (highest (apply #'max chord)))
138      (loop :for a := lowest :then (+ a interval)
139          :maximizing a :until (> a highest)
140          :finally (return (subst a lowest chord)))))
141
142  (defun all-rotations (chord &optional (interval (midi-cents 12)))
143    "Returns a list of all <chord> rotations."
144    (let ((sorted-chord (sort (copy-seq chord) #'<)))
145      (labels ((upa (a b)
146                 (cond ((> a b) a)
147                       (t (upa (+ a interval) b)))))
148        (append (list sorted-chord)
149                (loop :for note :in (butlast sorted-chord)
150                    :for achord := (subst (upa note (apply #'max sorted-chord))
151                                          note sorted-chord)
152                    :then (subst (upa note (apply #'max achord))
153                                 note achord)
154                    :collect achord)))))
155
156  (defun many-rotations (chord interval iterations multiplier)
157    "Rotates the <chord> using <interval> (see above);
```

```
158 in each one of the <iterations> the note that goes to the top of the chord
159 is transposed by a increasing amount, multiplied by a <multiplier>."
160   (loop :for i :from 0 :below iterations
161     :for a := chord :then
162               (let ((rotated-chord (sort (copy-seq (one-rotation a interval)) #'<)))
163                 (append (list (+ (car (last rotated-chord))
164                                  (* i (midi-cents multiplier))))
165                         (butlast rotated-chord)))
166     :collect a))
167
168 (defun many-many-rotations (chord interval iterations
169                             multiplier-min multiplier-max multiplier-step
170                             &key (rounded t))
171   "Returns a list of lists with all the 'many-rotations' for a range of multipliers,
172 between <multiplier-min> and <multiplier-max>, progressing by <multiplier-step>.
173 The :rounded keyword returns results in 12TET when set to T, and microtonal when
174 set to NIL."
175   (loop :for m :from multiplier-min :upto multiplier-max :by multiplier-step
176     :collect (if rounded
177                  (mapcar (lambda (x)
178                            (mapcar (lambda (y)
179                                      (* (midi-cents 1) (round (/ y (midi-cents 1)))))
180                                    x))
181                          (many-rotations chord interval iterations m))
182                  (many-rotations chord interval iterations m))))
183
184 (defun rotation-matrix (chord &key (interval (midi-cents 12)) (rem-dups t))
185   "Returns a matrix of rotations, where for each note in <chord> there's a set of
186 rotations, transposed to keep that pivot note constant. The :rem-dups keyword
187 removes ocurring duplicate chords."
188   (let* ((rotations (all-rotations chord interval))
189          (sorted-list (mapcar (lambda (x) (sort (copy-seq x) #'<)) rotations))
190          (r (loop :for h :from 0 :upto (length rotations)
191                   :for pivot :in (first sorted-list)
192                   :append
193                   (loop :for i :in sorted-list
194                         :for tquot := (- (nth h i) pivot)
195                         :collect (mapcar (lambda (x) (- x tquot)) i)))))
196     (if rem-dups
197         (remove-duplicates r :test #'equal :from-end t)
198         r)))
199
200 (defun expand-chord-up (chord &optional (multiplier 1))
201   "Increases the interval between notes, consecutively from bottom to top,
202 a semitone between the first and second notes, two semitones between the
203  second and third, etc. For intervals other than semitones use <multipler>."
204   (let ((sorted-chord (sort (copy-seq chord) #'<)))
205     (mapcar
206      (lambda (x) (+ x (* (position x sorted-chord) (midi-cents 1) multiplier)))
207      sorted-chord)))
208
209 (defun expand-chord-down (chord &optional (multiplier 1))
210   "Same as above, but from top to bottom."
```

```
211   (let ((sorted-chord (sort (copy-seq chord) #'>)))
212     (mapcar
213      (lambda (x) (- x (* (position x sorted-chord) (midi-cents 1) multiplier)))
214      sorted-chord)))
215
216 (defun expand-chord-pivot (chord &optional (multiplier 1))
217   "Same as above, expand both up and down around a pivot middle note."
218   (let* ((sorted-chord (sort (copy-seq chord) #'<))
219          (average-elt (elt sorted-chord (- (round (/ (length sorted-chord) 2)) 1))))
220     (loop :for n :in sorted-chord
221           :if (> n average-elt) :collect n :into over
222           :if (< n average-elt) :collect n :into under
223           :if (= n average-elt) :collect n :into over :and :collect n :into under
224           :finally (return (rest (append (expand-chord-up over multiplier)
225                                          (expand-chord-down under multiplier)))))))
226
227 (defun many-expansions (chord multiplier direction iterations)
228   "Performs one of the previous operarions several times, returning a sequence of chords."
229   (loop :for new-chord := chord
230         :then (cond ((eq direction 'down) (expand-chord-down new-chord multiplier))
231                     ((eq direction 'up) (expand-chord-up new-chord multiplier))
232                     ((eq direction 'pivot) (expand-chord-pivot new-chord multiplier)))
233         :repeat iterations :collect new-chord))
234
235 ;;; -------
236 ;;; VOICING
237 ;;; -------
238
239 (defun transpoct (chord bottom top)
240   "Transposes each note of <chord> to the nearest octave between <bottom> and <top>."
241   (labels ((up (note bottom)
242              (if (> note bottom)
243                  note
244                  (up (+ note (midi-cents 12)) bottom)))
245            (down (note top)
246              (if (< note top)
247                  note
248                  (down (- note (midi-cents 12)) top))))
249     (loop :for note :in chord
250           :if (< note bottom) :collect (up note bottom)
251           :else :if (> note top) :collect (down note top)
252           :else :collect note)))
253
254 (defun transpoct-seq (seq bottom top)
255   "Applies the function transpoct to a series of chords."
256   (mapcar #'(lambda (x) (transpoct x bottom top)) seq))
257
258 (defun top-limit (chord-or-sequence top)
259   "Transposes chords one or more octaves down until all the notes are below <top>."
260   (labels ((range (chord top)
261                   (if (< (apply #'max chord) top)
262                       chord
263                       (range (mapcar (lambda (x) (- x (midi-cents 12))) chord) top))))
```

```
264     (if (sublistp chord-or-sequence)
265     (loop :for chord :in chord-or-sequence
266         :collect (range chord top))
267     (range chord-or-sequence top))))
268
269 (defun bottom-limit (chord-or-sequence bottom)
270   "Transposes chords one or more octaves up until all the notes are above <bottom>."
271   (labels ((range (chord bottom)
272                   (if (> (apply #'min chord) bottom)
273             chord
274             (range (mapcar (lambda (x) (+ x (midi-cents 12))) chord) bottom))))
275     (if (sublistp chord-or-sequence)
276     (loop :for chord :in chord-or-sequence
277         :collect (range chord bottom))
278     (range chord-or-sequence bottom))))
279
280 (defun closed-position (chord)
281   "Puts <chord> in closed position."
282   (let* ((rotations (all-rotations chord))
283      (range (loop :for ch :in rotations
284                   :collect (cons (- (apply #'max ch) (apply #'min ch))
285                                  ch))))
286     (cdr (first (sort (copy-seq range) #'list<)))))
287
288 ;;; --------
289 ;;; ANALYSIS
290 ;;; --------
291
292 (defun unique-p (l)
293   "Checks if a list only has unique elements."
294   (or (null l)
295       (and (not (member (car l) (cdr l)))
296        (unique-p (cdr l)))))
297
298 (defun mod12 (chord)
299   (mapcar (lambda (x) (mod (/ x (midi-cents 1)) 12)) chord))
300
301 (defun mod12-unique-p (chord)
302   "Checks if a list only has unique mod12 elements."
303   (unique-p (mod12 chord)))
304
305 (defun count-chords-with-repeated-pcs (chord-list)
306   "Counts the number of chords in <chord-list> that have repeated pitch classes."
307   (loop :for chord :in chord-list
308     :counting (not (mod12-unique-p chord))))
309
310 (defun count-unique-chords (chord-list)
311   "Counts the number of different mod12 chords in <chord-list>."
312   (let ((mod12-sorted (loop :for chord :in (mapcar #'mod12 chord-list)
313                 :collect (copy-seq (sort chord #'<)))))
314     (length (remove-duplicate-sublists mod12-sorted))))
315
316 (defun find-best-expansion (chord direction iterations &optional (decimals 0))
```

```
317   "Finds the multipler for function <many-expansions> that generates the largest number of different mod12 chords.
          Returns a dotted pair (best multiplier . number of different chords)."
318   (let* ((results (loop :for m :from 1 :upto 11 :by (/ 1
319                                                        (expt 10 decimals))
320                         :collect (count-unique-chords (mapcar (lambda (x) (mapcar #'round x))
321                                                               (many-expansions chord
322                                                                                m
323                                                                                direction
324                                                                                iterations)))))
325         (best (+ 1 (position (apply #'max results) results))))
326     (values best (elt results (- best 1)) results)))
327
328 (defun all-intervals (chord)
329   "Returns a list of all intervals present in <chord>."
330   (loop :for achord := chord :then (rest achord)
331       :while (> (length achord) 1)
332       :append (loop :for n :in achord
333           :for a := (first achord)
334           :when (not (eql n a)) :collect (- n a))))
335
336 (defun count-intervals (chord)
337   "Returns a list of dotted pairs (interval . its ocurrences in <chord>)."
338   (let* ((intervals (all-intervals chord))
339       (k (remove-duplicates intervals)))
340     (loop :for i :in k
341       :collect (cons i (count i intervals)))))
342
343 (defun duplicate-pcs (chord)
344   "Returns the number of duplicate pitch classes."
345   (loop :with mod12-chord := (mod12 chord)
346     :for n :in mod12-chord
347     :count (> (count n mod12-chord) 1)))
348
349 (defun duplicate-pcs-relative (chord)
350   "Returns the number of duplicate pitch classes, in proportion to the number of notes in <chord>."
351   (/ (duplicate-pcs chord)
352       (length chord)))
353
354 (defun interval-score (chord &optional (score '(0 20 16 8 8 4 20 4 12 12 16 20)))
355   "Attributes a value to each of the mod 12 intervals present in <chord> according to the
356 optional list <score>, in which the first element is the value of interval class 0, the second
357 element the value of interval class 1, etc. Sums all the values and returns a total score for the <chord>."
358   (let ((sorted-chord (copy-seq (sort chord #'<))))
359     (loop :for n :in (count-intervals sorted-chord)
360       :sum (loop :for s :from 0 :upto 11
361             :when (eql (mod (/ (car n) (midi-cents 1)) 12) s)
362               :sum (* (elt score s) (cdr n))))))
363
364 (defun simpsons-index (chord)
365   "Attributes a score to the ordered set <chord> according to the diversity of its intervals, calculated using the
          formula for Simpson's index of diversity. There must be at least two different intervals."
366   (let ((intervals (count-intervals chord)))
367     (when (> (length intervals) 1)
```

```
        (loop :for n :in (count-intervals chord)
      :summing (* (cdr n) (- (cdr n) 1)) into r
      :summing (cdr n) :into q
      :finally (return (- 1 (/ r (* q (- q 1)))))))))

(defun harmonic-coincidence (chord fundamental &key (inverse nil) (compare-spectra nil))
  "Compares <chord> with the harmonic series starting on <fundamental>.
Returns the degree of coincidence. Higher values mean more coincidence."
  (let* ((spectrum (harmonic-series fundamental 100))
     (new-chord (if compare-spectra
            (flatten (mapcar (lambda (x) (harmonic-series x 14)) chord))
            chord))
     (degree (/ 1 (1+ (/ (loop :for note :in new-chord
                   :sum (loop :for p :in spectrum
                          :minimize (abs (- note p))))
                 (length new-chord))))))
    (if inverse
    (- 1 degree)
    degree)))

;;; ---------------
;;; SORT AND SEARCH
;;; ---------------

(defun sort-chords (list-of-chords functions
            &key (weights (make-list (length functions) :initial-element 1)))
  "Scores <list-of-chords> according to <functions> and <weights>, returning a sorted tree of
relative values in the form ((<score1> (<chord1>)) (<score2> (<chord2>)) ... (<scoren> (<chordn>))), where <
    score1> is the highest."
  (labels ((scaled-score (s) (mapcar (lambda (x) (scale-value x
                                     (apply #'min s)
                                     (apply #'max s)
                                     0
                                     1))
                     s)))
    (let* ((weights-sum (reduce #'+ weights))
       (relative-weights (mapcar (lambda (x) (/ x weights-sum)) weights))
       (scores (loop :for f :in functions
             :for w :in relative-weights
             :collect (mapcar (lambda (x) (* x w))
                      (scaled-score (mapcar f list-of-chords)))
             :into r
             :do (print r)
             :finally (return (loop :for i :from 0 :below (length (first r))
                        :collect (reduce #'+ (mapcar (lambda (x) (nth i x))
                                     r)))))))
      (sort (copy-seq (mapcar #'list scores list-of-chords)) #'list>))))

(defun sort-sequences (list-of-chord-sequences functions
               &key (weights (make-list (length functions) :initial-element 1)))
  "Scores <list-of-chord-sequences> according to <functions> and <weights>, returning a sorted tree
in the form ((<score1> ((chord 1a) (chord 1b) ... (chord 1n)))
            (<score2> ((chord 2a) (chord 2b) ... (chord 2n)))
```

```
                ...
                (<scoren> ((chord na) (chord nb) ... (chord nn)))), where <score1> is the highest."
  (sort (copy-seq (mapcar (lambda (chord-sequence)
                    (cons
                     (reduce #'+ (mapcar #'car
                                 (sort-chords chord-sequence
                                              functions
                                              :weights weights)))
                     chord-sequence))
                  list-of-chord-sequences))
    #'list>))

;;; ---------------
;;; RHYTHM GRAVITY
;;; ---------------

(defun gravity-force (m1 m2 r)
  (let ((g (* 6.67398 0.00000000001)))
    (/ (* g m1 m2) (expt r 2))))

(defun center-of-mass (body-list)
  (loop
     :with total-mass = (loop :for a :in body-list :sum (getf a :mass))
     :for b :in body-list
     :sum (* (getf b :mass) (getf b :pos)) :into s
     :finally (return (* (/ 1 total-mass) s))))

(defun create-body-list (positions masses)
  (loop :for p :in positions
     :for m :in masses
     :collect (list ':pos p ':mass m ':vel 0 ':f 0)))

(defun gravity (body-list)
  (loop
     :for body :in body-list :collect
     (loop :for other :in body-list
    :when (not (eq body other))
    :sum (* (gravity-force
          (getf body :mass)
          (getf other :mass)
          (- (getf other :pos) (getf body :pos)))
         (if (< (getf body :pos) (getf other :pos)) 1 -1)))))

(defun update (body-list dt scale)
  (let ((f-list (gravity body-list)))
    (loop :for f :in f-list ;Update gravity-force
       :for body :in body-list
       :do (setf (getf body :f) f))
    (loop :for body :in body-list ;Calculate new positions and velocites
       :for x := (getf body :pos)
       :for v := (getf body :vel)
       :for new-pos := (+ x (* v dt))
       :for f := (getf body :f)
```

```
        :for m := (getf body :mass)
        :for new-vel := (+ v (* (/ f m) dt))
        :do (progn)
        (setf (getf body :pos) new-pos)
        (setf (getf body :vel) new-vel)
        :finally (return (collision-control body-list scale)))))

(defun collision-control (body-list scale)
  (loop :with scale-factor = (/ (* scale 3) 100)
      :for (a b) :on body-list :while b
      :do
      (when (or (eql (round (/ (getf a :pos) scale-factor))
                (round (/ (getf b :pos) scale-factor)))
              (>= (getf a :pos) (getf b :pos)))
        (progn
      (setf (getf a :pos) -1)
      (setf (getf b :vel) (/
                           (+
                            (* (getf a :mass) (getf a :vel))
                            (* (getf b :mass) (getf b :vel)))
                           (+ (getf a :mass) (getf b :mass))))))
      :finally (return (remove-if (lambda (x) (eql (getf x :pos) -1)) body-list))))

(defun get-offsets (body-list)
  (loop :for body :in body-list
      :collect (getf body :pos)))

(defun rhythm-gravity (positions masses time &optional (step 20))
  "Calculates a list of offsets after <time>"
  (let ((scale (- (apply #'max positions) (apply #'min positions))))
    (loop :for i :upto time
        :for bl = (create-body-list positions masses) :then (update bl step scale)
        :finally (return (get-offsets bl)))))

(defun rg (positions masses time &optional (step 20))
  "Outputs a visualisation of the gravity function."
  (loop :with scale = (- (apply #'max positions) (apply #'min positions))
      :for i :upto time
      :for bl = (create-body-list positions masses) :then (update bl step scale)
      :while (> (list-length bl) 1)
      :do (let ((line (make-string (+ 1 (apply #'max positions))
                                   :initial-element #\_)))
        (loop :for b :in bl
            :for position = (round (getf b :pos))
            :for c = (cond
                ((< (getf b :mass) 300000) ".")
                ((and (>= (getf b :mass) 300000)
                      (< (getf b :mass) 600000)) "o")
                ((>= (getf b :mass)) 600000 "O"))
            :do (replace line c :start1 position :end1 (+ 1 position))
            :finally (format t "~a ~a~%" line i)))))

(defun make-body-list (magnitudes)
```

```
526   (mapcar (lambda (x) (* x 100000)) magnitudes))
527
528 (defun time-series (positions masses iterations step)
529   "Returns a list of offsets for a number of <iterations>."
530   (let ((scale (- (apply #'max positions) (apply #'min positions))))
531     (loop :for i :upto iterations
532         :for bl = (update (create-body-list positions masses) step scale)
533         :then (update bl step scale)
534         :collect (get-offsets bl))))
535
536 ;;; ---------
537 ;;; NECKLACES
538 ;;; ---------
539
540 (defun all-necklaces (limit &rest filters)
541   "Because a necklace can be expressed as a series of zeros (rests)
542 and ones (onsets), converts all numbers from 1 up to (2^<limit>)-1 into base-2,
543 and returns the corresponding binary lists. The results can be filtered
544 to only include the necklaces for which all the functions <filters> return T.
545 For example: (all-necklaces 6 #'rhythmic-oddity-p (lambda (x) (< (count '1 (binary->interonset x)) 2))) -> all
       necklaces with total length up to 6, with the rhythmic oddity property and no more than two consecutive attacks.
       "
546   (loop :for i :from 1 :upto (1- (expt 2 limit))
547     :for b := (binary-list i)
548     :when (or (not filters)
549             (loop :for fun :in filters
550               :always (funcall fun b)))
551       :collect b))
552
553 (defun count-necklaces (limit &rest filters)
554   "Same as all-necklaces, but more efficiently just counts how many solutions there are, without returning them all."
555   (loop :for i :from 1 :upto (1- (expt 2 limit))
556     :for b := (binary-list i)
557     :when (or (not filters)
558             (loop :for fun :in filters
559               :always (funcall fun b)))
560       :count b))
561
562 (defun binary->interonset (l)
563     "Accepts a list <l> of binary digits and returns a list
564 of inter-onset intervals. For example (1 1 0 0 0 1) -> (1 4 1)."
565     (when (member 1 l)
566       (let ((normal-l
567             (loop :for ll := l :then (rotate ll 1)
568               :while (zerop (first ll))
569               :finally (return ll))))
570     (loop :for o :in normal-l
571             :for c := 0 :then (incf c)
572             :when (and (plusp o) (plusp c))
573           :collect c :into r :and :do (setf c 0)
574             :finally (return (append r (list (+ c 1))))))))
575
576 (defun interonset->binary (l)
```

```
577   "Accepts a list <l> of inter-onset intervals and returns a list
578 of binary digits. For example (1 4 1) -> (1 1 0 0 0 1)."
579   (unless (member-if-not #'plusp l)
580     (flatten (mapcar (lambda (x) (if (eq x '1)
581                                      '1
582                                      (list '1
583                                        (make-list (1- x)
584                                                   :initial-element '0))))
585                      l))))
586
587 (defun lyndon-words (n a M)
588   "Generates all Lyndon words of length <= <n> over an alphabet <a>..<M>. The algorithm is an adaptation of
          the one by Jean-Paul Duval, Gnration d'une section des classes de conjugaison et arbre des mots de Lyndon de
          longueur borne, in Theoretical Computer Science, 60, 1988, pp. 255-283."
589   (let ((w (make-list (1+ n)))
590     (i 1))
591     (setf (elt w 1) a)
592     (loop
593       :do
594      (loop :for j :from 1 :to (- n i)
595            :do (setf (elt w (+ i j)) (elt w j)))
596       :collect (subseq w 1 (1+ i))
597       :do
598      (setf i n)
599      (loop :while (and (> i 0)
600                        (eq (elt w i) M))
601            :do (decf i))
602      (when (> i 0)
603        (setf (elt w i) (1+ (elt w i))))
604       :until (or (= i 0))))))
605
606 (defun lyndon-words-with-duration (n a M duration)
607   "Generates all Lyndon words of length <= <n> over an alphabet <a>..<M>, where <a> and <M> are numbers
          and <a> < <M>, and where the sum of all numbers is <= <duration>."
608   (remove-if-not (lambda (x) (<= (reduce #'+ x) duration))
609           (lyndon-words n a M)))
610
611 (defun necklace-chord (root inter-onsets)
612   "Builds a pitch collection starting on <root> and following the <inter-onsets> intervals."
613   (loop :for n :in inter-onsets
614     :for r := (+ root n) :then (+ r n)
615     :collect r :into results
616     :finally (return (push root results))))
617
618 ;;; ------------------
619 ;;; GEOMETRY OF RHYTHM
620 ;;; ------------------
621
622 (defun rhythmic-oddity-p (input &key (interonset-intervals nil))
623   "Checks if <input> has the rhythmic oddity property.
624 Accepts a list of binary digits by default. If <input> is a list of inter-onset intervals
625 then the function must be called with :interonset-intervals t."
626   (let* ((necklace (if interonset-intervals
```

```
                    (interonset->binary input)
                    input))
      (w (length necklace)))
     (when (evenp w)
       (loop :for pulse :in necklace
         :for i :from 0
         :never (and (plusp pulse)
                     (plusp (elt necklace
                                 (mod (+ i (/ w 2))
                                  w))))))))

(defun count-necklaces-with-rhythmic-oddity (length)
  (count-necklaces (expt 2 length) :filter #'rhythmic-oddity-p))


;;; --------
;;; EVENNESS
;;; --------

(defun evenness (ioi)
  "Returns an index of how well distributed are the onsets."
  (let* ((n (reduce #'+ ioi))
      (k (length ioi))
      (h (/ n k))
      (m (apply #'max ioi)))
    (/ 1 (1+ (- m h)))))

(defun evenness-weight (ioi)
  "With the onsets distributed around an unit circle, returns the sum of the chord lengths between
every pair of onsets. See Steven Block and Jack Douthett, 'Vector Products and Intervalic Weighting', in Journal of
      Music Theory, vol. 38, no. 1, pp. 21-41, 1994."
  (let* ((n (reduce #'+ ioi))
      (weighting-vector (loop :for i :from 1 :upto (/ n 2)
                  :collect (* 2 (sin (/ (* i pi) n)))))
      (interval-vector (mapcar (lambda (x) (let* ((m (mod x n)))
                          (if (<= m (floor (/ n 2)))
                            m
                            (- n m))))
                  (all-intervals (loop :for i :in (append '(0) (butlast ioi))
                             :sum i :into z
                             :collect z)))))
    (reduce #'+ (mapcar (lambda (x) (elt weighting-vector (1- x))) interval-vector))))

(defun evenness-weight-index (ioi)
  "Returns the proportion between the weight of cycle <ioi>, expressed as a list of inter-onset
intervals, and the weight of a maximally even cycle with the same cardinality."
  (if (= (length ioi) 1)
      1
      (/ (evenness-weight ioi)
      (evenness-weight (make-list (length ioi) :initial-element 1)))))


;;; ----------------
```

```
679 ;;; SPECIFIC SEARCH
680 ;;; ----------------
681
682 (defun necklace-specific-search (min-attacks max-length singles min-evenness mod12)
683   (all-necklaces max-length
684     #'rhythmic-oddity-p
685     (lambda (x)
686       (let ((ioi (binary->interonset x)))
687         (and (>= (length ioi) min-attacks)
688          (<= (count 1 ioi :test #'equalp) singles)
689          (> (evenness-weight-index ioi) min-evenness)
690          (or (not mod12) (mod12-unique-p (i->p ioi 0))))))))
691
692 (defun necklace-specific-search-with-lyndon (max-length min-attacks max-attacks
693                                              min-ioi max-ioi singles
694                                              min-evenness mod12)
695   (remove-if-not (lambda (ioi) (and (>= (length ioi) min-attacks)
696                                     (<= (count 1 ioi) singles)
697                                     (> (evenness ioi) min-evenness)
698                                     (rhythmic-oddity-p ioi :interonset-intervals t)
699                                     (or (not mod12) (mod12-unique-p (i->p ioi 0)))))
700          (lyndon-words-with-duration max-attacks min-ioi max-ioi max-length)))
```

markov-n.asd

```
(defsystem "markov-n"
  :description "Markov chains on sample level with pcm audio files."
  :version "1"
  :author "Nuno Trocado"
  :depends-on ("alexandria")
  :components ((:file "packages")
               (:file "alias-method" :depends-on ("packages"))
               (:file "markov-n" :depends-on ("alias-method"))))
```

packages.lisp

```
(in-package "COMMON-LISP-USER")

(defpackage :alias-method
  (:use :common-lisp)
  (:export :make-discrete-random-var))

(defpackage :markov-n
  (:use :common-lisp :alexandria :alias-method)
  (:export :main))
```

alias-method.lisp

```
(in-package :alias-method)

;;An implementation of the Alias Method.
;;
;; NOTE: Additional comments, and an updated, more accurate version
;; can be found here:
;;
;;
;; The function MAKE-DISCRETE-RANDOM-VAR takes an array of
;; probabilities and an (optional) array of values. Produces a
;; function which returns each of the values with the specified
;; probability (or the corresponding integer no values have been
;; given).
;;
;; It needs only one call to RANDOM for every value produced.
;;
;; For the licence, see at the end of the file.

(defun create-alias-method-vectors (probabilities)
  (let* ((N (length probabilities))
      (threshold (/ 1.0d0 N))
      (alternatives (make-array N :element-type 'fixnum))
      (p_keep (make-array N :initial-contents (coerce probabilities 'list))) bigs lows)
    (loop :for i :from 0 :below N :do
       (if (>= threshold (aref probabilities i)) (push i lows) (push i bigs)))
    (loop :while lows :do
       (let* ((0ne (pop lows))
          (Tw0 (pop bigs))
          (delta (- threshold (aref p_keep 0ne))))
     (if tw0
```

```
            (progn (setf (aref p_keep 0ne) (/ (aref p_keep 0ne) threshold))
                (setf (aref alternatives 0ne) Tw0)
                (decf (aref p_keep Tw0) delta)
                (if (>= threshold (aref p_keep Tw0))
                (push Tw0 lows)
                (push Tw0 bigs)))
            (setf (aref p_keep 0ne) 1.0d0))))

      ;; Numerical noise might leave some bigs dangling, with
      ;; | p_keep − 1/N | very small.

      (dolist (k bigs) (setf (aref p_keep k) 1.0d0))
      (values p_keep alternatives)))

(defun make−discrete−random−var (probabilities &optional values)
  (when (and values (/= (length values) (length probabilities)))
    (error "different number of values and probabilities."))
  (let* ((N (float (length probabilities) 0.0d0)))
    (multiple−value−bind
      (p_keep alternatives)
      (create−alias−method−vectors probabilities)
      #'(lambda () (labels ((result (k) (if values (aref values k) k)))
              (multiple−value−bind (k r) (floor (random N))
                (if (> r (aref p_keep k)) (result (aref alternatives k)) (result k))))))))

;; Tests the alias method. p holds the prescribed probabilities, and
;; cnt the measured ones.
(defun test−alias−method (n runs)
  (let ((p (make−array n :element−type 'double−float))
    (cnt (make−array n :initial−element 0.0d0)))
    (dotimes (i n) (setf (aref p i) (random 1.0d0)))
    (let ((nc (loop for i from 0 below n summing (aref p i))))
      (dotimes (i n) (setf (aref p i) (/ (aref p i) nc)))
      (let ((rp (make−discrete−random−var p)))
    (dotimes (i runs) (incf (aref cnt (funcall rp))))
    (dotimes (i n) (setf (aref cnt i) (/ (aref cnt i) runs)))))
    (values p cnt)))

;;; Copyright (c) 2006, Mario S. Mommer ;;; Permission is hereby granted, free of charge, to any person ;;; obtaining a
    copy of this software and associated documentation ;;; files (the "Software"), to deal in the Software without ;;;
    restriction, including without limitation the rights to use, copy, ;;; modify, merge, publish, distribute, sublicense,
    and/or sell copies ;;; of the Software, and to permit persons to whom the Software is ;;; furnished to do so,
    subject to the following conditions: ;;; The above copyright notice and this permission notice shall be ;;; included
    in all copies or substantial portions of the Software. ;;; THE SOFTWARE IS PROVIDED "AS IS", WITHOUT
    WARRANTY OF ANY KIND, ;;; EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
    WARRANTIES OF ;;; MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND ;;;
    NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT ;;; HOLDERS BE LIABLE
     FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, ;;; WHETHER IN AN ACTION OF CONTRACT,
    TORT OR OTHERWISE, ARISING FROM, ;;; OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
    THE USE OR OTHER ;;; DEALINGS IN THE SOFTWARE.
```

markov-n.lisp

```
(in−package :markov−n)
```

```
2
3  (defparameter *table* (make-hash-table))
4  (defparameter *rand* 0)
5
6  (defun combine-bytes (bytes)
7    (loop :for n :downfrom (1- (length bytes))
8        :for b :in bytes
9        :summing (* b (expt 65536 n))))
10
11 (defun read-file (file n)
12   (with-open-file (data
13                    file
14                    :direction :input
15                    :element-type '(signed-byte 16))
16     (let* ((data-seq (make-sequence 'list (file-length data)))
17            (data-length (length data-seq)))
18       (read-sequence data-seq data)
19       (format t "File read. Size is ~a samples.~%" data-length)
20       (format t "~a~%" (make-string 100 :initial-element #\_))
21       (nconc data-seq (subseq data-seq 0 n))
22       (loop :with counter-step := (ceiling (/ data-length 100))
23           :for next :in (subseq data-seq n)
24           :for i :from n
25           :for subseq-start := (- i n)
26           :do (let* ((data-subseq (subseq data-seq subseq-start i))
27                      (key (combine-bytes data-subseq)))
28                 (if (not (hash-table-p (gethash key *table*)))
29                 (progn
30                   (setf (gethash key *table*) (make-hash-table))
31                   (setf (gethash next (gethash key *table*)) 1))
32                 (if (not (gethash next (gethash key *table*)))
33                 (setf (gethash next (gethash key *table*)) 1)
34                 (incf (gethash next (gethash key *table*)))))
35                 (when (zerop (mod i counter-step))
36           (progn (format t ".") (finish-output)))))))))
37
38 (defun get-next (current)
39   (let* ((hash-size (hash-table-count (gethash current *table*)))
40       (all-keys (alexandria:hash-table-keys (gethash current *table*)))
41       (all-values (alexandria:hash-table-values (gethash current *table*)))
42       (value-total (reduce #'+ all-values))
43       (prob-array (make-array hash-size
44                   :initial-contents (mapcar
45                                       #'(lambda (x) (/ x value-total))
46                                       all-values)))
47       (values-array (make-array hash-size
48                     :initial-contents all-keys))
49       (rp (make-discrete-random-var prob-array values-array)))
50     (funcall rp)))
51
52 (defun write-file (file n size &optional (initial-list (make-list (1+ n) :initial-element 0)))
53   (with-open-file (out
54                    file
```

```
55                :direction :output
56                :element-type '(signed-byte 16)
57                :if-exists :supersede)
58        (let ((buffer (subseq initial-list 0 n)))
59          (format t "~a~%" (make-string 100 :initial-element #\_))
60          (dotimes (i size t)
61        (let ((new-byte (if (> (the fixnum i) (the fixnum n))
62                            (get-next (combine-bytes buffer))
63                            (elt initial-list i))))
64          (write-byte new-byte out)
65          (setf buffer (rest (nconc buffer (list new-byte))))
66          (when (zerop (ceiling (mod i (/ size 100))))
67            (progn (format t ".") (finish-output)))))))))
68
69  (defun main (order input output size)
70    "Analyses <input> file and creates a new <output> file with <size> bytes."
71    (read-file input order)
72    (format t "~%Creating new file...~%")
73    (let ((first-bytes (make-sequence 'list (1+ order))))
74      (with-open-file (file input :direction :input :element-type '(signed-byte 16))
75        (read-sequence first-bytes file))
76      (write-file output order size first-bytes)))
```

ESCOLA
SUPERIOR
DE MÚSICA
E ARTES
DO ESPETÁCULO
POLITÉCNICO
DO PORTO

P.PORTO

M

MESTRADO
COMPOSIÇÃO E TEORIA MUSICAL

Interacção homem-máquina no processo composicional

Nuno André Pinheiro Trocado da Costa