Universidade de São Paulo
Instituto de Matemática e Estatística

Sistema de Acompanhamento Musical Automatizado

Roberto Piassi Passos Bodo
Marcelo Queiroz (Orientador)

---

08 de Fevereiro de 2010

Sumário

1 Introdução

Computação Musical é a área da Ciência da Computação que está em foco nesta monografia. Curtis Roads[1] apresenta, em seu livro chamado *The Computer Music Tutorial* [1], detalhes sobre diversas sub-áreas da Computação Musical, como Síntese de Áudio, Processamento de Sinais de Áudio, Análise de Áudio, Psicoacústica, etc. Entre tantas destaca-se a sub-área que lida com Acompanhamento Computadorizado e Performance Interativa. Abaixo veremos detalhes sobre o assunto.

Robert Rowe[2] descreve, em um de seus livros [2], dois tipos de sistemas de acompanhamento computadorizado: os sistemas orientados a performance e os orientados a partitura.

Sistemas orientados a performance utilizam, essencialmente, o material sonoro gerado durante a performance, aplicando processamentos e transformações nesse material e produzindo a saída como resultado dessas transformações. Esses sistemas possuem regras que dizem o que fazer (ou como reagir) de acordo com o contexto musical e com a entrada do músico. Já os sistemas orientados a partitura sabem de antemão o que o músico irá tocar e possuem um *script* de como a peça deverá soar.

O sistema aqui apresentado é orientado a partitura, mais especificamente, o sistema pertence à classe de aplicações conhecida como *score followers* (tradução livre: seguidores de partitura). Esses programas, basicamente, são capazes de seguir a performance de um músico e, comparando-a instante a instante com a partitura original, inferir o andamento do mesmo.

Roger B. Dannenberg[3] apresenta, em dois artigos [3, 4], algumas técnicas para a criação de sistemas desse tipo, que serão detalhadas mais adiante.

[1] http://clang.mat.ucsb.edu/clang/home.html
[2] http://homepages.nyu.edu/∼rr6/
[3] http://www.cs.cmu.edu/∼rbd/

1.1 O Problema do Acompanhamento Musical Automatizado

Seguindo as idéias apresentadas em um dos artigos de Roger Dannenberg [3] podemos formalizar o problema do acompanhamento musical automatizado como segue: dada uma música com diversas faixas, cada uma contendo um instrumento, e dada a partitura[4] para todas elas, definimos uma das faixas como a faixa solo e as restantes como o acompanhamento. Queremos, então, tocar o acompanhamento de forma correta, dada a entrada do músico em tempo real.

1.2 A Solução proposta por Dannenberg

A abordagem escolhida divide o problema definido na seção anterior em três sub-problemas:

- processar a entrada do músico solista em tempo real:
  Precisamos saber constantemente o que o músico está fazendo. Não podemos perder nenhum tipo de informação e precisamos apresentar uma resposta para cada ato do solista com pouco ou nada de atraso, de preferência.

- comparar essa entrada com a partitura original:
  Precisamos saber em que ponto da música o solista está e, para isso, comparamos tudo o que foi tocado com a partitura da faixa solo que contém os eventos esperados.

- gerar o acompanhamento de acordo com o andamento do músico:
  Sabendo em que parte da música estamos, podemos calcular o andamento da música e, assim, tocar o acompanhamento de forma correta.

Em seus artigos, Roger Dannenberg dá mais ênfase para o segundo item descrito acima, ou seja, para o algoritmo de *matching*. Por outro lado, o gerenciamento do relógio virtual que acompanha o andamento flexível do músico é um problema sutil e não-trivial. Por isso apresentaremos, nesta monografia, a solução para o problema do acompanhamento musical automatizado dividida em quatro partes, a saber, o processamento da entrada, a comparação com a partitura, a atualização do relógio virtual e a geração da saída, que serão apresentadas nas Seções 2–5.

1.3 Motivações/Objetivos

A criação deste sistema pretende atender as necessidades de um músico solitário. Podemos imaginar um estudante de Música que está treinando,

[4] deste ponto até o final desta monografia iremos utilizar o termo "partitura" para qualquer forma de representação musical, não necessariamente no formato tradicional, com pentagrama, clave, etc.

pela primeira vez, um concerto para um instrumento solo qualquer. No início, algumas passagens são tocadas mais devagar e ficaria difícil acompanhar a gravação original. Pensando nisso, o sistema gera o acompanhamento conforme o ritmo do músico, na velocidade em que ele se sentir confortável.

Ainda no caso do estudante, o sistema apresentado não apenas se adapta ao andamento do músico, mas também pode mostrar quais erros foram cometidos pelo mesmo. Sempre sabemos da partitura original qual é a próxima nota esperada, então sempre que o solista cometer um erro podemos avisá-lo disso.

Outro caso que queremos cobrir é o de um intérprete que não tem um grupo de músicos para acompanhá-lo. Nesse caso, o sistema pode ser utilizado em uma apresentação ao vivo para suprir essa falta.

2 Entrada

O sistema apresentado aqui possui dois tipos de entrada: a partitura e a entrada do músico. A primeira será carregada no início da execução do sistema. Já a segunda será lida em tempo real durante a execução do sistema. Nesta seção veremos detalhes sobre ambas.

2.1 Partitura (Acompanhamento/Solista)

Conforme foi explicado na seção anterior poderíamos ter qualquer formato de entrada para a partitura. Veremos abaixo detalhes sobre as extensões escolhidas.

Extensão SMF - *Standard MIDI File*

O protocolo MIDI (*Musical Instrument Digital Interface*), publicado em 1982, apareceu pela primeira vez em um instrumento em 1983. Ele foi criado para permitir a comunicação entre sintetizadores construídos por diferentes fabricantes, como Roland, Yamaha e Oberheim. Dave Phillips afirma, em seu livro chamado *Linux Music & Sound* [5], que, por volta de 1985, o MIDI já tinha se tornado um dos mais importantes avanços do século 20 na área de tecnologia musical. Uma das extensões escolhidas para a partitura é a extensão *.mid*. Diversos fatores influenciaram a escolha dessa extensão.

Primeiramente, existe uma grande disponibilidade e variedade de arquivos MIDI. Uma rápida busca na Internet nos dá diversos resultados de *sites* que contêm uma quantidade enorme desses arquivos. O *site mididb.com*, por exemplo, disponibiliza toda a discografia dos Beatles em MIDI. Este fator nos dá uma ótima variedade de entradas para testes.

Em segundo lugar, destacamos a facilidade para edição e manipulação dos dados presentes nesse formato. Na Figura 1 podemos ver, de uma forma bem intuitiva, quais notas foram tocadas e quando foram tocadas. Podemos imaginar um arquivo MIDI como uma régua no tempo, mostrando quando as notas começam e quando terminam.

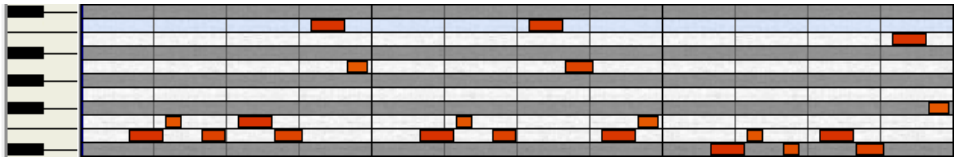

Figura 1: Representação de um arquivo MIDI no *software* Rosegarden.

Um terceiro fator é o tamanho dos arquivos MIDI, que permite a representação de peças longas em arquivos muito pequenos em comparação com arquivos de áudio, por exemplo. Só para termos uma noção desse fato, uma das músicas utilizadas nos testes possuia mais de 6 minutos de duração, 19

faixas e apresentava apenas 92KB de tamanho, enquanto um arquivo WAV estéreo a 44.1kHz com esta duração teria mais de 60MB.

Um último motivo é manter a compatibilidade com a entrada do músico que será feita por um instrumento MIDI (conforme veremos na Seção 5). Os eventos provenientes de um instrumento MIDI possuem a mesma sintaxe que os eventos presentes em um arquivo MIDI, isto é, os *bytes* lidos apresentam o mesmo formato, o que é considerado um fator positivo.

Abaixo temos um trecho da música representada na Figura 1 para ilustrarmos a sintaxe de um arquivo MIDI. Observe que os valores já foram convertidos para números inteiros.

```
 79, 144, 64, 127
135, 128, 64, 77
139, 144, 65, 111
167, 128, 65, 84
199, 144, 64, 120
235, 128, 64, 87
259, 144, 65, 127
315, 128, 65, 91
319, 144, 64, 119
365, 128, 64, 75
379, 144, 72, 126
435, 128, 72, 73
439, 144, 69, 110
472, 128, 69, 80
```

O primeiro valor é chamado de *delta time* e diz quando o evento será executado.

O segundo valor ocupa 1 *byte* e diz qual é o tipo do evento (nos 4 primeiros *bits*) e para qual canal MIDI ele será enviado (nos 4 últimos *bits*). Acima temos **144** (10010000) para o evento do tipo *Note On* (enviado para o canal MIDI 0) e **128** (10000000) para o evento do tipo *Note Off* (enviado também para o canal MIDI 0).

Os dois valores restantes são os parâmetros de cada evento. Para eventos do tipo *Note On* temos o valor da nota MIDI seguido da intensidade com que a tecla foi tocada. Para eventos do tipo *Note Off* temos o valor da nota MIDI seguido da velocidade com a qual a tecla foi solta.

Extensão CSV - *Comma Separated Values*

Utilizando o *software midicsv*[5] podemos converter arquivos MIDI para a extensão *.csv*, que possue o mesmo conteúdo que um arquivo MIDI em formato texto. Isso faz com que a leitura e edição desses arquivos seja ainda

[5] http://www.fourmilab.ch/webtools/midicsv/

mais fácil. Inicialmente utilizado para testes (conforme veremos na Seção 6), esse formato foi incorporado ao sistema como mais uma opção de entrada. Abaixo temos o mesmo trecho da mesma música apresentado anteriormente, porém na sintaxe de um arquivo CSV.

```
2,  79, Note_on_c,  0, 64, 127
2, 135, Note_off_c, 0, 64, 77
2, 139, Note_on_c,  0, 65, 111
2, 167, Note_off_c, 0, 65, 84
2, 199, Note_on_c,  0, 64, 120
2, 235, Note_off_c, 0, 64, 87
2, 259, Note_on_c,  0, 65, 127
2, 315, Note_off_c, 0, 65, 91
2, 319, Note_on_c,  0, 64, 119
2, 365, Note_off_c, 0, 64, 75
2, 379, Note_on_c,  0, 72, 126
2, 435, Note_off_c, 0, 72, 73
2, 439, Note_on_c,  0, 69, 110
2, 472, Note_off_c, 0, 69, 80
```

Podemos observar que um arquivo CSV apresenta dois campos a mais por evento: o primeiro campo é a faixa (ou *track*) e o terceiro e quarto campos são o tipo do evento e o canal MIDI já explicitamente separados.

Independentemente do formato escolhido pelo usuário precisamos ainda fazer alguns pré-processamentos dos dados originais. Para simplificar a atualização do relógio virtual é feita a conversão dos *delta times* para valores absolutos de tempo. Além disso, a faixa do solista é filtrada a fim de manter apenas os eventos relevantes para o rastreamento do andamento do músico.

A unidade original de um *delta time* é uma unidade de tempo relativa (*ticks*) e uma das etapas de pré-processamento consiste em convertê-las para uma unidade de tempo absoluta (milissegundos). Um arquivo MIDI fornece duas informações relativas ao andamento da música:

- *time division*:
  Este campo encontrado no cabeçalho inicial do arquivo define a taxa de *ticks* por pulso (*beat*).

- *set tempo*:
  Este meta-evento define a taxa de microssegundos por pulso.

Para convertermos os *delta times*, temos que aplicar as seguintes contas:

$$\underbrace{delta\ time'}_{milissegundos} = \underbrace{delta\ time}_{ticks} * \frac{1}{\underbrace{time\ division}_{\frac{ticks}{beat}}} * \underbrace{tempo}_{\frac{microssegundos}{beat}} * \frac{1}{1000}$$

O evento *set tempo* pode aparecer diversas vezes no decorrer da música e portanto temos que considerar as diversas mudanças de andamento ocorridas entre dois outros eventos. Isso fica ilustrado na figura abaixo:

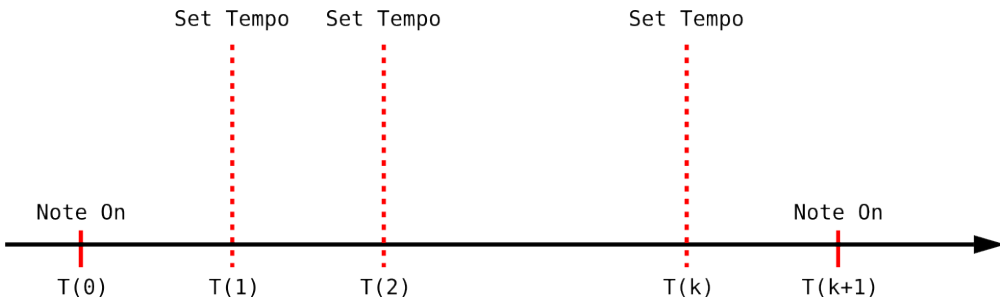

Figura 2: Diversas mudanças de andamento entre dois eventos MIDI.

O valor do *delta time* nesse caso será calculado assim:

$$T'_{k+1} = \sum_{i=0}^{k} (T_{i+1} - T_i) * C,$$

$$\text{onde } C = \frac{1}{\underbrace{time\ division}_{\frac{1}{\frac{ticks}{beat}}}} * \underbrace{tempo}_{\frac{microssegundos}{beat}} * \frac{1}{1000}$$

A segunda alteração na partitura é feita na faixa solo. Não vamos considerar eventos como o *Pitch Bend*, por exemplo, na hora de comparar a entrada do músico com a partitura. Concretamente, vamos remover todos os eventos de tipos diferentes de *Note On*, inclusive os eventos do tipo *Note Off*. Podemos fazer isso, pois estamos considerando que a entrada será feita por um instrumento melódico, assim todo *Note On* será seguido de um *Note Off* correspondente.

2.2 Performance do Solista

Um sistema de acompanhamento musical automatizado pode permitir a entrada de instrumentos convencionais, através de captadores ou microfones, ou de instrumentos MIDI, que geram o código simbólico no formato descrito anteriormente. No primeiro caso seria necessário um pré-processamento da entrada de áudio (normalmente usando Transformadas de Fourier) para reconhecer os inícios de notas e as alturas musicais a fim de produzir símbolos comparáveis aos que aparecem na partitura. No segundo caso a informação gerada pelo instrumento já está no mesmo formato usado para representar a partitura; os instrumentos MIDI são facilmente lidos e eles geram uma

seqüência muito bem definida de *bytes*, iguais às presentes nos arquivos MIDI.

Nosso sistema trata presentemente o segundo caso, podendo ser estendido por ferramentas externas (como Intelliscore WAV to MIDI converter ou inst2midi) para tratar o primeiro caso.

Vale a pena explicitarmos aqui que, pelo mesmo motivo que limpamos a faixa solo da partitura, só vamos considerar os eventos da entrada do tipo *Note On* para efeito de comparação com a partitura. Outros eventos são enviados diretamente para a saída, ou seja, se o músico tocar uma nota seguida de um *Pitch Bend*, ele ouvirá o resultado dessa combinação saindo nas caixas de som.

O tratamento de instrumentos MIDI conectados ao computador varia de acordo com o sistema operacional utilizado. No Apêndice A desta monografia temos um código de exemplo que mostra como ler um instrumento MIDI em um sistema GNU/Linux, usando o dispositivo MIDI padrão definido por esse sistema, utilizando a linguagem ANSI C.

3 Algoritmo de *Matching*

Com a partitura carregada e o instrumento pronto para ser lido, podemos comparar as duas entradas. Relembramos que tanto as notas da partitura quanto as notas provenientes do instrumento MIDI podem ser representadas como símbolos (como A, B, C), o que permite reduzir o problema de comparação das seqüências de notas ao problema clássico de comparação de textos. Roger Dannenberg propõe a utilização de um algoritmo de programação dinâmica para esse trabalho em um de seus artigos [3]. Nesta seção vamos mostrar maiores detalhes de como isso foi feito.

3.1 Programação Dinâmica Aplicada ao Problema LCS

Seguindo a mesma idéia dos algoritmos de divisão-e-conquista, os algoritmos de programação dinâmica resolvem um problema quebrando-o em subproblemas menores, resolvendo esses últimos e combinando suas soluções. O livro *Introduction to Algorithms* [6] divide o método de programação dinâmica em quatro passos:

- caracterizar a estrutura da solução ótima;
- definir recursivamente o valor de uma solução ótima;
- calcular o valor de uma solução ótima de maneira *bottom-up*;
- construir a solução ótima a partir dos resultados obtidos.

Esse método pode ser utilizado para encontrar a maior subseqüência comum a duas seqüências dadas. Seguindo os passos apresentados acima, precisamos primeiramente caracterizar a estrutura da solução ótima e, para isso, vamos citar o teorema apresentado em [6].

> Teorema[6]: Seja $X = x_1, x_2, ..., x_m$ e $Y = y_1, y_2, ..., y_n$ duas seqüências e seja $Z = z_1, z_2, ..., z_k$ uma maior subseqüência comum a X e Y.
>
> - Se $x_m = y_n$, então $z_k = x_m = y_n$ e Z_{k-1} é uma maior subseqüência comum a X_{m-1} e Y_{n-1}.
> - Se $x_m \neq y_n$ e $z_k \neq x_m$, então Z é uma maior subseqüência comum a X_{m-1} e Y.
> - Se $x_m \neq y_n$ e $z_k \neq y_n$, então Z é uma maior subseqüência comum a X e Y_{n-1}.

[6] A prova deste teorema pode ser encontrada no capítulo 15 da referência [6] e será omitida neste texto.

O teorema acima leva à seguinte solução recursiva para achar a maior subseqüência comum a X e Y (passo 2 do método de programação dinâmica):

- se $x_m = y_n$, devemos achar a maior subseqüência comum a X_{m-1} e Y_{n-1}; concatenando $x_m = y_n$ a essa subseqüência temos a maior subseqüência comum a X e Y.
- se $x_m \neq y_n$, devemos achar a maior subseqüência comum a X_{m-1} e Y, e a X e Y_{n-1}. A maior delas será a maior subseqüência comum a X e Y.

Dada a solução recursiva podemos calcular o tamanho da maior subseqüência comum a X e Y (passo 3 do método de programação dinâmica) utilizando o algoritmo descrito abaixo. A versão original recebe como entrada as duas seqüências, o tamanho de cada uma delas e uma matriz para guardar os valores das subseqüências máximas que foram calculadas até o momento. O código abaixo implementado em C é baseado no algoritmo em pseudo-código apresentado em [6].

```
void lcs_length(char* x, int m, char* y, int n, int** c) {
    int i, j;

    for (i = 0; i <= m; i++) {
        c[i][0] = 0;
    }

    for (j = 0; j <= n; j++) {
        c[0][j] = 0;
    }

    for (i = 1; i <= m; i++) {
        for (j = 1; j <= n; j++) {
            if (x[i-1] == y[j-1]) {
                c[i][j] = c[i - 1][j - 1] + 1;
            }
            else {
                if (c[i - 1][j] >= c[i][j - 1]) {
                    c[i][j] = c[i - 1][j];
                }
                else {
                    c[i][j] = c[i][j - 1];
                }
            }
        }
    }
}
```

O quarto e último passo do método de programação dinâmica (construção da solução ótima) não foi considerado na implementação do sistema, pois o que importa para o *matching* entre a partitura e a performance é a

posição na partitura da última nota tocada pelo solista, supondo que ela pertença à subseqüência máxima (e não seja um erro, por exemplo).

Na seção a seguir veremos como as informações da matriz c permite tratar os erros de performance e localizar a posição do músico na partitura.

3.2 Adaptação ao Sistema de Acompanhamento Automatizado

Para ser utilizado no sistema de acompanhamento automatizado o algoritmo apresentado na seção acima passou por quatro modificações.

A primeira modificação é bem conhecida e é feita para economizar memória. Ao invés de guardar a matriz inteira, podemos ficar com apenas duas linhas, já que o algoritmo utiliza apenas os valores da linha anterior para calcular os valores da linha atual.

A segunda modificação transforma o algoritmo anterior em um algoritmo que funciona em tempo real. No nosso caso, conhecemos de antemão apenas a partitura (primeira seqüência) e atualizamos as linhas da matriz conforme o músico solista toca, isto é, conforme ele produz a segunda seqüência.

```
void lcs_length_online(char* score, int n, char note) {
    int i;

    for (i = 1; i <= n; i++) {
        if (score[i - 1] == note) {
            c[1][i] = c[0][i - 1] + 1;
        }
        else {
            if (c[0][i] >= c[1][i - 1]) {
                c[1][i] = c[0][i];
            }
            else {
                c[1][i] = c[1][i - 1];
            }
        }
    }

    for (i = 1; i <= n; i++) {
        c[0][i] = c[1][i];
    }
}
```

Em terceiro lugar, nunca vamos olhar para a partitura como um todo. Uma música pode conter milhares de notas e isto, além de tornar a matriz desnecessariamente grande, ainda permite a ocorrência de erros grosseiros (como por exemplo uma nota errada ser considerada um acerto em um ponto diferente da música). Para evitar isso utilizaremos apenas um trecho da partitura a cada momento, o qual será denominado de janela. O tamanho dessa

janela é variável e será definido em função do número de erros cometidos pelo músico. Especificamente, usaremos um critério de tolerância (a saltos na partitura) proporcional ao número de erros cometidos pelo solista, ou seja, quanto mais erros o solista cometer, mais podemos olhar à frente na partitura. Concretamente, temos uma variável que guarda o número de erros atuais do músico e ela é usada para calcular o tamanho da janela, conforme a função abaixo.

$$\text{Window\_Size}(\text{errors}) = 2 * \text{errors} + 1$$

Finalmente é importante determinar como os resultados obtidos serão usados para a tomada de decisão em relação à geração do acompanhamento. Para isso, um critério é utilizado para decidir se o tamanho da subseqüência encontrada é suficientemente grande, a fim de decidirmos em qual ponto da partitura o músico está.

Um dos fatores considerados é o número de erros atuais; quanto mais erros acumulados, maior a exigência sobre a entrada do músico, forçando que a subseqüência comum seja cada vez maior. Para definir a exigência sobre o tamanho da subseqüência comum, dado um certo número de erros, utilizamos a função linear abaixo, que nos dá o tamanho da subseqüência comum suficiente para considerar a última nota tocada como um acerto, dado o número de erros atuais.

$$\text{Least\_Subsequence\_Size}(\text{errors}) = \lceil \frac{errors}{5} \rceil$$

Definimos no código MAX_ERRORS como 25 e limitamos o tamanho da subseqüência a 5, para que as decisões sobre localização não sejam tomadas com muito atraso. Por fim, quando decidimos que temos um *match* zeramos tanto a variável que guarda o número de erros, quanto a matriz do algoritmo.

Vamos mostrar agora alguns exemplos de como o algoritmo descrito acima se comporta em algumas situações comuns:

- No exemplo abaixo, o músico toca todas as notas da partitura de forma correta.

	A	B	B	C	A
A	1	1	1	1	1
B	1	2	2	2	2
B	1	2	3	3	3
C	1	2	3	4	4
A	1	2	3	4	5

Observe como a cada nota produzida pelo músico (ou seja, a cada nova linha da matriz) o maior valor aparece pela primeira vez exatamente sobre a nota (coluna) correspondente da partitura, indicando a posição do músico solista naquele instante.

- No segundo exemplo, o músico toca uma nota a mais do que o trecho da partitura possui.

	A	B	B	C	A
A	1	1	1	1	1
B	1	2	2	2	2
B	1	2	3	3	3
B	1	2	3	3	3
C	1	2	3	4	4
A	1	2	3	4	5

Quando o terceiro B é tocado, ele será ignorado de certa forma, pois a linha 4 da matriz ficará exatamente igual à anterior, indicando que a posição do músico solista na partitura não avançou. A partir da quinta nota tocada (quinta linha da matriz) voltamos a observar os casamentos com as notas correspondentes na partitura.

- No terceiro exemplo, o músico toca uma nota a menos do que a partitura possui.

	A	B	B	C	A
A	1	1	1	1	1
B	1	2	2	2	2
C	1	2	2	3	3
A	1	2	2	3	4

Podemos ver que agora é o segundo B da partitura que será ignorado e a terceira nota tocada indica um avanço do músico solista para a quarta nota da partitura.

- No último exemplo, o músico toca o mesmo número de notas da partitura, mas substitui uma delas por uma nota errada.

	A	B	B	C	A
A	1	1	1	1	1
B	1	2	2	2	2
F	1	2	2	2	2
C	1	2	2	3	3
A	1	2	2	3	4

O F tocado será ignorado e, de uma forma muito parecida com o segundo exemplo, a linha 3 da matriz ficará exatamente igual à anterior. O erro do músico será compensado no próximo passo do algoritmo, pelo casamento que acontece entre a quarta nota tocada e a quarta nota da partitura.

Na próxima seção discutiremos a implementação do relógio virtual, que utiliza as informações de *matching* que acabamos de discutir para prever de forma adaptativa a contagem de tempo do músico, e assim permitir a geração do acompanhamento de forma ininterrupta.

4 Relógio Virtual

Na seção anterior vimos como determinar um casamento entre a entrada do solista e a partitura da faixa correspondente. Agora precisamos calcular o andamento do músico e, para isso, utilizaremos um relógio virtual. Nesta seção vamos mostrar o comportamento deste relógio virtual e como utilizar esses casamentos para atualizar o mesmo.

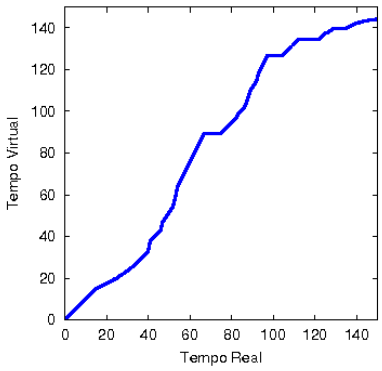

Figura 3: Tempo Virtual X Tempo Real

O gráfico acima é um exemplo de como o relógio virtual pode evoluir em relação ao relógio real. Inicialmente o relógio virtual tem um andamento igual ao relógio real, ou seja, tem o andamento original da música. Podemos ver também que o gráfico possui derivadas sempre maiores ou iguais a zero, isto é, o tempo virtual sempre avança, embora não tenha sempre a mesma taxa. Para entendermos melhor o comportamento do relógio virtual vamos ilustrar abaixo três casos extremos.

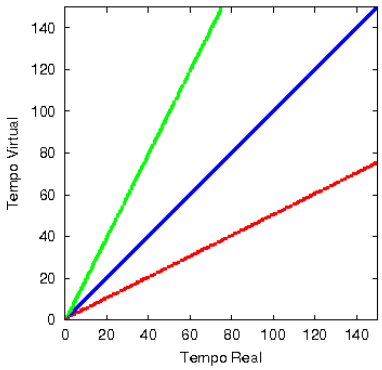

Figura 4: $f(x) = 2x$; $g(x) = x$; $h(x) = \frac{1}{2}x$.

No primeiro caso ($f(x) = 2x$) o músico sempre se mantêm em um andamento maior do que o original (ilustrativo do que ocorre nos trechos do exemplo

anterior onde a derivada é maior do que 1), no segundo caso ($g(x) = x$) o músico se mantêm sempre no mesmo andamento da música original (ilustrativo do que ocorre nos trechos onde a derivada é exatamente igual a 1) e no terceiro caso ($h(x) = \frac{1}{2}x$) o músico sempre se mantêm em um andamento menor do que o original (ilustrativo dos trechos onde a derivada é menor do que 1).

Voltando a olhar para a Figura 3 podemos ver trechos de derivada iguais a zero. Isso acontece pois não podemos deixar o relógio virtual ultrapassar o instante da próxima nota esperada da partitura. Se essa restrição não é feita o relógio poderia evoluir até o final da música sem que houvesse nenhuma entrada do solista. Esses trechos de derivada igual a zero representam momentos em que o computador está esperando por alguma entrada. Existem também trechos que correspondem a mudanças contínuas de andamento, como *accelerando* e *ritardando*.

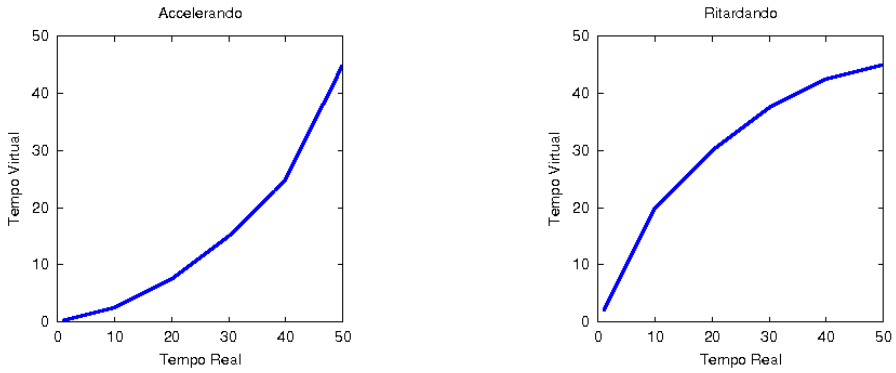

Figura 5: Casos de mudanças contínuas de andamento.

Atualizamos o valor do relógio virtual constantemente durante a execução do sistema, de acordo com os casos abaixo:

1. Se há neste instante uma entrada do músico solista:
   Neste caso podemos tomar decisões a partir da entrada para atualizar o relógio virtual.

   (a) Se há um casamento entre a entrada e a partitura:
   Quando temos um casamento entre a nota que o músico acabou de tocar e uma nota no instante T da partitura, sabemos onde o relógio virtual deve estar.

      i. Se há algum acompanhamento pendente:
      Se existe um acompanhamento para ser tocado entre o momento em que estávamos e T, precisamos correr para alcançar o músico. Para isso, estabelecemos uma meta, que é chegar ao tempo T, e um prazo para isso acontecer. Neste caso,

temos uma *flag* que muda o modo de execução do sistema até todo o acompanhamento pendente ser tocado.

ii. Se não há acompanhamentos pendentes:
Podemos atribuir o valor T diretamente ao relógio virtual.

(b) Se não houve casamento entre a entrada e a partitura:
Tratamos este caso do mesmo modo como quando não temos entrada, ou seja, consideramos a entrada do músico inútil, para efeito de atualização do relógio virtual.

2. Se não há neste instante uma entrada do músico solista:
Quando não temos nenhuma informação vinda do solista, temos que inferir quais são os próximos valores do relógio virtual. Para isso utilizamos os dois últimos momentos em que ocorreram casamentos entre notas e utilizamos a derivada desse trecho para inferir qual o próximo valor do relógio virtual. Vale a pena observarmos que, conforme já foi mencionado antes, não podemos ultrapassar o instante da próxima nota esperada da partitura, ou seja, só vamos incrementando o valor do relógio virtual até esse instante e o relógio fica parado até que haja alguma nova entrada.

A linguagem C possui algumas funções que verificam se há entrada ou não nos dispositivos, permitindo que a execução do programa não seja bloqueada pelas funções de leitura. Maiores detalhes de como isso foi feito serão mostrados no Apêndice A desta monografia.

5 Saída/Acompanhamento

Dada a evolução do relógio virtual podemos gerar a saída de nosso sistema, ou seja, podemos tocar o acompanhamento. Até agora só lidamos com música simbólica, mas queremos ouvir o que está sendo tocado e, para isso, precisamos sintetizar áudio a partir do MIDI.

Existem diversas técnicas de síntese de áudio a partir de modelos simbólicos, tais como a síntese aditiva ou a síntese subtrativa, porém a complexidade em emular instrumentos tradicionais com estas técnicas é tão grande que tiraria o foco do trabalho do problema do acompanhamento automatizado.

Existem, também, placas de som que recebem eventos MIDI e fazem a síntese de áudio em *hardware*, mas isso excluiria parte dos usuários que não possuem esse tipo de equipamento.

Por último, existem alguns *softwares* que fazem a síntese de áudio a partir de eventos MIDI e essa foi considerada a melhor alternativa. Escolhemos o *software* TiMidity++[7] por algumas razões expostas a seguir.

TiMidity++ é um software livre que roda em sistemas GNU/Linux e funciona, basicamente, como um sintetizador MIDI. Ele pode tocar arquivos MIDI em tempo real ou convertê-los para arquivos WAV. No nosso caso, vamos utilizá-lo como um seqüenciador MIDI, ou seja, o inicializaremos de forma que possamos enviar eventos MIDI para síntese em tempo real.

Veremos mais a frente que não basta inicializá-lo nesse modo, precisamos também utilizar um dispositivo MIDI virtual que fará o meio de campo entre o nosso sistema e o TiMidity++. No Apêndice A desta monografia temos um pequeno tutorial que demonstra como fazer isso.

Por último, vale a pena mostrarmos a estrutura de dados que armazena a partitura e o modo em que vamos escrever os eventos MIDI na saída.

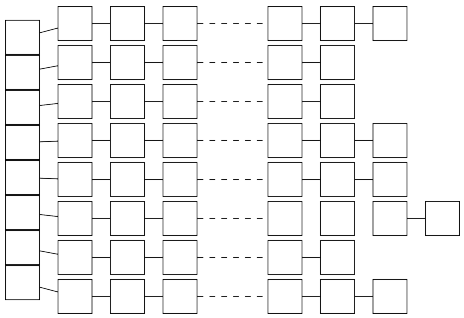

Figura 6: Representação da tabela que armazena a partitura.

Temos uma estrutura parecida com uma tabela, sendo que cada linha é uma lista ligada que contém o *delta time* e os *bytes* referentes ao evento MIDI no formato esperado pelo TiMidity++. A todo instante comparamos

[7] http://timidity.sourceforge.net/

o valor do relógio virtual com os *delta times* dos eventos armazenados e aqueles que tiverem valor menor ou igual ao relógio virtual são escritos no dispositivo MIDI de saída.

```
for (i = 0; i < n_tracks; i++) {
    if (i != (soloist_track - 1)) {
        while (links[i] != NULL &&
                links[i]->delta_time <= virtual_clock) {
            write(output_device_fd, links[i]->bytes, 3);
            links[i] = links[i]->next;
        }
    }
}
```

Acima temos um pequeno trecho de código que mostra como o acompanhamento é tocado de fato. Temos um vetor de ponteiros (um ponteiro para cada faixa, exceto a faixa do solista) e tocamos todos os eventos que possuem um *delta time* menor ou igual ao relógio virtual, conforme já foi dito antes.

É interessante notar que eventos simultâneos são normalmente representados e transmitidos seqüencialmente no protocolo MIDI, e não há perda de qualidade em processarmos os dados da mesma maneira, já que o tempo de varredura desta tabela é mínimo comparado ao intervalo normal entre comandos enviados por um controlador MIDI, que é da ordem de 1ms.

6 Considerações Finais

O sistema apresentado nesta monografia foi todo implementado utilizando a linguagem C e voltado para sistemas GNU/Linux. Seguem abaixo alguns detalhes sobre a metodologia utilizada.

- *IDE*:
  Todo projeto foi feito utilizando o Eclipse. Geralmente utilizada para se programar em Java, esta *IDE* apresenta um pacote voltado para C que deixa o trabalho muito mais dinâmico e produtivo.

- controle de versão:
  Todo o projeto foi feito utilizando Subversion. Normalmente utilizado para se trabalhar em grupos grandes de programadores, esta ferramenta se torna muito útil para um único programador quando temos um projeto com muitas linhas de código. Utilizando um *plugin* para o Eclipse chamado *Subclipse* tarefas como reverter um trecho de código se tornam banais.

- testes automatizados:
  Algumas partes do projeto produzem saída textual e sua corretude pode ser facilmente verificada utilizando expressões regulares. Por exemplo, a biblioteca que lida com arquivos MIDI lê os *bytes* do arquivo e imprime na tela os eventos correspondentes (e seus eventuais parâmetros). Conforme foi dito na Seção 2 podemos converter um arquivo MIDI em um arquivo CSV. Portanto, podemos comparar a saída do código com o arquivo CSV e, para isso, foi utilizada a linguagem Perl que é ideal para se trabalhar com expressões regulares.

6.1 Resultados Obtidos

Alguns testes foram realizados e eles apresentam ótimos resultados em dois quesitos principais:

1. O sistema se adapta a qualquer andamento do músico.

   Casos em que o andamento é muito lento, incluindo o caso em que o músico pára de tocar completamente, foram testados e sempre temos o acompanhamento tocado de forma correta. Por exemplo, quando o músico pára de tocar, se tivermos algumas notas soando, elas continuam soando até o músico voltar e dar continuidade à música.

   Os casos de andamentos muito rápidos também foram testados, mas vale a pena explicitarmos aqui que os testes não foram feitos com músicos profissionais, ou seja, não foram testados casos em que a velocidade é extremamente rápida. Os testes mostram que o sistema

também gera o acompanhamento de forma correta para as velocidades alcançadas.

2. O sistema se adapta aos erros cometidos pelo usuário.

   Existem três erros básicos: o de inserção, o de deleção e o de substituição. Todos já foram apresentados na Seção 3, onde mostramos alguns exemplos de matrizes do algoritmo de programação dinâmica:

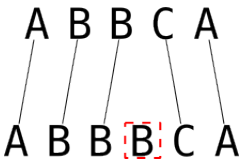

Figura 7: Inserção: o músico toca uma nota a mais.

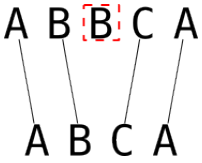

Figura 8: Deleção: o músico toca uma nota a menos.

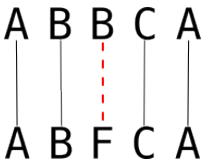

Figura 9: Substituição: o músico troca uma das notas.

   Os testes realizados mostram que o sistema se adapta muito bem a todos os erros apresentados acima.

Agora vamos mencionar alguns casos de entrada do solista que não são toleradas nesta versão do sistema, mas que estão nas perspectivas de trabalhos futuros.

- trinados:
  Trinados são um tipo de ornamento musical que consiste na oscilação rápida entre duas notas adjacentes, sendo que na partitura apenas uma das notas aparece escrita. Aqui, todas as notas exceto a primeira serão tratadas como erros (inserções).

- glissandos:
  Um glissando consiste na transição de uma nota a outra tocando-se todas as notas no meio do caminho. Como a partitura só indica as notas

inicial e final, um glissando será tratado como um erro. Por exemplo, se tivermos uma guitarra com um captador MIDI e o músico utilizar um *slide* para tocar, então, mesmo que ele passe por todas as notas da partitura de forma correta, todas as transições serão consideradas como erros.

- improvisação:
  Nesta implementação do sistema não deixamos muito espaço para improvisação. Apesar disso temos uma tolerância pequena para erros cometidos na seqüência, representada por uma variável que contém o número máximo de erros permitidos. Se o músico tocar um número de notas erradas maior do que o número máximo de erros permitidos, o programa se encerra, com um aviso de erro.

6.2 Conclusões

A monografia aqui apresentada mostrou detalhes sobre o processo de criação de um sistema de acompanhamento musical automatizado orientado a partitura.

A Seção 2 apresentou detalhes sobre o formato de entrada da partitura e, junto com o Apêndice A, detalhes de como ler um instrumento MIDI em tempo real. A Seção 3 apresentou o problema da maior subseqüência comum e a maneira como ela foi utilizada para fazer o casamento entre as duas entradas. A Seção 4 mostrou como determinar o andamento da música utilizando um relógio virtual e a Seção 5 apresentou um modo de tocar o acompanhamento, dada a evolução do relógio virtual e, junto com o Apêndice A, deu detalhes de como sintetizar os eventos MIDI para áudio.

Conforme foi visto nesta seção, o sistema implementado possui ainda espaço para melhorias, mas demonstra potencial em alguns quesitos. Procurando por outros sistemas do mesmo tipo na Internet encontramos diversos resultados, porém, nenhum deles roda em sistemas GNU/Linux, nem possuem código aberto. Vale a pena citarmos aqui, também, que quase todos eles possuem um custo demasiado alto para um estudante de música, por exemplo.

Portanto, o código do sistema aqui descrito, junto com esta monografia, poderão servir de base para a implementação de um sistema bem mais robusto e, futuramente, até servir de alternativa para músicos profissionais.

7 Parte Subjetiva

7.1 Desafios/Frustrações Encontrados

A implementação do sistema apresentou alguns desafios que serão apresentados nesta seção.

O primeiro desafio encontrado foi a falta de uma documentação oficial ou mais organizada que apresentasse a especificação de um arquivo MIDI. A MMA (*MIDI Manufacturers Association*) que define e publica a especificação oficial dos arquivos MIDI não apresenta nenhum conteúdo *online*. Portanto, a informação necessária para implementar a biblioteca que lida com esse tipo de arquivo foi retirada de fontes diversas, algumas não tão confiáveis, ou seja, toda essa informação teve que ser reunida e organizada até chegarmos a uma versão útil desta especificação.

Vale a pena explicitarmos aqui uma das informações que foi muito difícil de ser encontrada. Em um arquivo MIDI temos três tipos de eventos: os eventos MIDI, os "meta" eventos e os eventos chamados de *system exclusive*. Os eventos MIDI aparecem sempre divididos em três partes:

- o *delta time*:
  Já mencionado nesta monografia, se refere ao momento em que o evento deve ser tocado e possui tamanho variável em *bytes*.

- o tipo do evento:
  Diz qual é o evento (*Note On*, *Note Off*, *Pitch Bend*, etc) e possui tamanho fixo (1 *byte*).

- os parâmetros do evento:
  Os eventuais parâmetros podem ter 1 ou 2 *bytes*, dependendo do evento. Por exemplo, o evento do tipo *Note On* vem seguido de dois *bytes*, sendo o primeiro a nota e o segundo a intensidade com a qual ela foi tocada.

O fato mencionado antes, que foi muito difícil de ser encontrado, é que o *byte* referente ao tipo do evento não precisa aparecer sempre quando temos uma seqüência de eventos iguais. Por exemplo, se tivermos uma faixa com diversos eventos do tipo *Note On*, o *byte* identificador pode aparecer somente no primeiro evento e temos que considerá-lo o mesmo para todos os outros. Isso cria a seguinte dificuldade: como identificar uma mudança de evento?

Para isso temos que olhar para o primeiro *bit*. Todos os *bytes* referentes ao tipo do evento MIDI possuem o seu primeiro *bit* igual a 1. Então, olhando para o próximo *byte* após o *delta time*, se o primeiro *bit* for igual a 0, temos que utilizar o evento anterior e considerar o *byte* atual como um parâmetro. Se for igual a 1, temos um evento novo. Isso fica como uma dica para o leitor interessado em implementar uma biblioteca que lida com arquivos MIDI.

Um segundo inconveniente foi a falta de padronização dos arquivos MIDI encontrados. Nem todos eles possuem faixas separadas para cada instrumento, muitos deles possuem uma mesma faixa com diversas mudanças de instrumento. Pode existir algum contexto em que isso seja útil, afinal o padrão permite tais liberdades, mas para viabilizar este projeto isso foi considerado irregular e tais arquivos não foram tratados.

O terceiro desafio foi produzir o acompanhamento. O TiMidity++, por exemplo, possui uma documentação muito ruim e tudo o que foi feito, desde a criação do dispositivo MIDI virtual até chegarmos no áudio saindo das caixas de som, foi muito trabalhoso. Foram retiradas muitas informações de fóruns, que nem sempre estavam corretos ou atualizados. Para chegarmos ao resultado final foram feitos muitos testes e, por isso, acrescentamos mais informações no Apêndice A desta monografia, com o intuito de ajudar outras pessoas que estejam interessadas em lidar com dispositivos MIDI em sistemas GNU/Linux.

7.2 Disciplinas do BCC relevantes para o trabalho

- MAC122 - Princípios de Desenvolvimento de Algoritmos
- MAC323 - Estruturas de Dados

  As matérias acima ensinam a lidar com listas ligadas e como definir estruturas conforme a necessidade do problema.

- MAC211 - Laboratório de Programação I
- MAC242 - Laboratório de Programação II

  As duas matérias acima foram as primeiras a apresentar metologias para a criação de projetos de grande porte. Foi nessas matérias que vimos a importância de fazer controle de versão do código (usando Subversion, por exemplo) e como fazer uma boa documentação (usando Doxygen, por exemplo).

- MAC338 - Análise de Algoritmos

  Para este projeto esta matéria é a mais importante com certeza absoluta. Foi nela que vimos o método de Programação Dinâmica e o problema da maior subseqüência comum, que formam o núcleo do sistema apresentado aqui.

- MAC422 - Sistemas Operacionais

  Não podemos deixar de fora a matéria acima que nos permitiu entender melhor como funcionam os sistemas GNU/Linux e isso foi útil, por exemplo, na hora de criar módulos, como o de MIDI virtual utilizado na saída do sistema.

- MAC337 - Computação Musical

  A única disciplina optativa que foi relevante para o desenvolvimento deste projeto foi a disciplina de Computação Musical. Ela ofereceu uma visão geral muito boa da área e apresentou detalhes sobre análise, processamento e síntese de áudio que, apesar de não serem utilizadas diretamente, tiveram a sua importância para o projeto.

7.3 Planos Futuros e Aprimoramentos do Sistema

O sistema implementado possui muito espaço para melhoras. Segue uma lista de algumas delas:

- permitir a entrada de instrumentos quaisquer, utilizando algum método de transcrição melódica;
- criar uma interface gráfica para tornar a execução mais amigável, onde poderíamos mostrar a partitura original da música utilizando o *software* LilyPond, por exemplo;
- criar dois modos de execução diferentes: um para estudo e outro para performance;
- motrar na tela os erros cometidos pelo músico, quando o mesmo estiver rodando o sistema em modo de estudo;
- lidar melhor com trinados e glissandos que, por não estarem explicitamente anotados na partitura, seriam tratados como erros pela versão atual do sistema;
- abrir um maior espaço para improvisações do solista, permitindo uma maior flexibilidade de uso em situações de performance ao vivo;

7.4 Agradecimentos

O desenvolvimento deste projeto não teria acontecido sem a orientação do Prof. Marcelo Queiroz que, em diversas reuniões desde o final de 2008, sempre foi muito atencioso, tirando minhas dúvidas, ajudando a organizar as minhas idéias, acompanhando o projeto bem de perto. Por esses e outros motivos eu sou muito grato!

A MIDI em sistemas GNU/Linux

Neste apêndice iremos mostrar alguns detalhes importantes de como lidar com MIDI em sistemas GNU/Linux.

A.1 Leitura de instrumentos MIDI

Quando temos um instrumento MIDI conectado a um sistema GNU/Linux conseguimos lê-lo a partir de um arquivo localizado em */dev*. Seu nome possui o seguinte formato: */dev/midi#*, onde # é o número do dispositivo. Abaixo temos um código de exemplo que mostra um jeito muito simples de obter as informações a partir de um instrumento MIDI.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/poll.h>

int main() {
    struct pollfd fds[1];
    unsigned char bytes[3];
    char* midi_device = "/dev/midi1";
    int r, fd = open(midi_device, O_RDONLY);

    if (fd == -1) {
        printf("error: %s not found...\n", midi_device);
        return 1;
    }

    fds[0].fd = fd;
    fds[0].events = POLLIN;

    while (1) {
        r = poll(fds, 1, 1000);

        if (r == -1) {
            printf("error!\n");
        } else if (r == 0) {
            printf("timeout!\n");
        } else {
            if (fds[0].revents & POLLIN) {
                read(fd, bytes, 3);
                printf("|%d|%d|%d|\n",
                        bytes[0], bytes[1], bytes[2]);
            }
        }
    }

    close(fd);

    return 0;
```

```
}
```

Observe que não foram utilizadas funções mais conhecidas, como *fopen()* e *fread()*. Essa escolha foi feita pela necessidade de trabalharmos com *file descriptors* e o motivo principal será mostrado mais adiante.

As funções utilizadas são explicadas abaixo:

```
int open(const char *path, int oflag, ... );
```

A função *open()* pertence à biblioteca *fcntl.h*. Ela recebe o nome do arquivo e uma *flag* que determina que modo em que esse arquivo será acessado. Ela devolve o *file descriptor* referente ao arquivo que queremos abrir.

```
ssize_t read(int fildes, void *buf, size_t nbyte);
```

A função *read()* pertence à biblioteca *unistd.h*. Ela recebe um *file descriptor*, um *buffer* onde ela irá escrever e o tamanho desse *buffer*. Ela devolve um valor não negativo, em caso de sucesso, e -1, em caso de erro na leitura.

```
int close(int fildes);
```

A função *close()* pertence à biblioteca *unistd.h*. Ela recebe um *file descriptor* e libera o acesso ao arquivo correspondente.

```
int poll(struct pollfd[], nfds_t, int);
```

A função *poll()* pertence à biblioteca *sys/poll.h*. O primeiro parâmetro dessa função é um vetor de estruturas do tipo *pollfd*.

```
struct pollfd {
    int fd;
    short int events;
    short int revents;
};
```

Essa estrutura apresenta três membros: o primeiro é um *file descriptor*, o segundo é uma *flag* que define o tipo de evento que queremos detectar e o terceiro é uma *flag* que define o tipo de evento que realmente ocorreu.

A biblioteca *sys/poll.h* possui algumas *flags* pré-definidas como a POLLIN, utilizada no exemplo. Ela representa o caso em que temos dados prontos para serem lidos. O segundo parâmetro que a função *poll()* recebe é um inteiro com a quantidade de membros que o vetor acima possui e o terceiro parâmetro é um *timeout*, isto é, o tempo que função espera para o evento ocorrer.

A.2 Síntese de Áudio utilizando o TiMidity++

Conforme vimos na Seção 5, utilizamos o programa chamado TiMidity++ para sintetizar áudio a partir dos eventos MIDI. Antes de inicializarmos o TiMidity++ precisamos criar um dispositivo MIDI virtual utilizando o comando abaixo:

```
sudo modprobe snd_virmidi
```

O comando acima cria os seguintes dispositivos novos em */dev*.

```
admmidi1
amidi1
dmmidi1
midi1
mixer1
```

O dispositivo que nos interessa é o *midi1*. Outra forma de verificar os dispositivos MIDI disponíveis é utilizar o seguinte comando:

```
pmidi -l
```

```
rppbodo@woodstock:~$ pmidi -l
 Port     Client name                      Port name
 14:0     Midi Through                     Midi Through Port-0
 20:0     Virtual Raw MIDI 1-0             VirMIDI 1-0
 21:0     Virtual Raw MIDI 1-1             VirMIDI 1-1
 22:0     Virtual Raw MIDI 1-2             VirMIDI 1-2
 23:0     Virtual Raw MIDI 1-3             VirMIDI 1-3
```

Utilizando o *pmidi* podemos verificar, inclusive, quais as portas referentes a cada dispositivo (abaixo veremos a importância disso).

Agora podemos inicializar o TiMidity++ conforme queremos, com o seguinte comando:

timidity -iAD -Os –realtime-priority=99

Os parâmetros utilizados têm os seguintes significados:

- -iAD:
  O parâmetro "-iA" inicializa o TiMidity++ como um seqüenciador MIDI cliente e o "-D" o coloca para rodar em *background*.

- -Os:
  O parâmetro "-Os" faz com que a saída do TiMidity++ seja enviada para o ALSA.

- –realtime-priority=99:
  O parâmetro "–realtime-priority=99" determina que o TiMidity++ tenha prioridade alta, evitando assim algum tipo de atraso na produção do áudio.

Neste ponto temos um dispositivo MIDI virtual funcionando e o TiMidity++ rodando em *background*.

```
rppbodo@woodstock:~$ pmidi -l
 Port     Client name                      Port name
 14:0     Midi Through                     Midi Through Port-0
 20:0     Virtual Raw MIDI 1-0             VirMIDI 1-0
 21:0     Virtual Raw MIDI 1-1             VirMIDI 1-1
 22:0     Virtual Raw MIDI 1-2             VirMIDI 1-2
 23:0     Virtual Raw MIDI 1-3             VirMIDI 1-3
128:0     TiMidity                         TiMidity port 0
128:1     TiMidity                         TiMidity port 1
128:2     TiMidity                         TiMidity port 2
128:3     TiMidity                         TiMidity port 3
```

Queremos agora conectá-los e, para isso, vamos utilizar o comando:

aconnect

```
rppbodo@woodstock:~$ aconnect 20:0 128:0
rppbodo@woodstock:~$ Requested buffer size 32768, fragment size 8192
ALSA pcm 'default' set buffer size 32768, period size 8192 bytes
```

Após todos os passos mostrados acima, podemos escrever no dispositivo MIDI virtual e a saída será sintetizada para as caixas de som. Abaixo temos um código de exemplo que mostra um jeito muito simples de tocarmos uma nota utilizando um evento MIDI.

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    unsigned char bytes[3];
    char* midi_device = "/dev/midi1";
    int fd = open(midi_device, O_WRONLY);

    if (fd == -1) {
        printf("error: %s not found...\n", midi_device);
        return 1;
    }

    bytes[0] = 144;
    bytes[1] = 60;
    bytes[2] = 127;
    write(fd, bytes, 3);

    sleep(1);

    bytes[0] = 128;
    bytes[1] = 60;
    bytes[2] = 127;
    write(fd, bytes, 3);

    close(fd);

    return 0;
}
```

As funções *open()* e *close()* já foram explicadas anteriormente.

ssize_t write(int fildes, const void *buf, size_t nbyte);

A função *write()* pertence à biblioteca *unistd.h*. Ela recebe um *file descriptor*, um *buffer* no qual ela irá escrever e o tamanho desse *buffer* e devolve um valor não negativo, em caso de sucesso, e -1, em caso de erro na escrita.

O motivo para usarmos esse conjunto de funções é o seguinte: no exemplo acima para escrevermos os *bytes* no dispositivo MIDI de saída poderíamos usar a função *fwrite()*, mas ela não tem ação imediata, ou seja, teríamos que usar a função *fwrite()* sempre seguida da função *fflush()*, para garantir uma escrita imediata. A função *write()*, por outro lado, possui ação imediata e por isso consideramos sua utilização uma solução mais elegante.

Como vimos anteriormente o primeiro *byte* determina o evento do tipo *Note On*, o segundo determina qual nota MIDI será tocada e o terceiro

determina a intensidade com a qual ela será tocada. Após enviarmos um *Note On* temos que enviar um *Note Off* para a nota não ficar tocando durante um longo período. Então, o primeiro *byte* determina o evento do tipo *Note Off*, o segundo determina qual a nota MIDI e o terceiro determina a velocidade com a qual a tecla será solta.

Referências

[1] ROADS, Curtis. The Computer Music Tutorial. The MIT Press, 1996.

[2] ROWE, Robert. Interactive Music Systems. The MIT Press, 1993.

[3] DANNENBERG, Roger. An On-Line Algorithm for Real-Time Accompaniment. Proceedings of the International Computer Music Conference ICMC'1985, Computer Music Association, 1985, p. 193-198.

[4] DANNENBERG, Roger; MUKAINO, Hirofumi. New Techniques for Enhanced Quality of Computer Accompaniment. Proceedings of the International Computer Music Conference ICMC'1988, Computer Music Association, 1988, p. 243-249.

[5] PHILLIPS, Dave. Linux Music & Sound. No Starch Press, 2000.

[6] CORMEN, Thomas; LEISERSON, Charles; RIVEST, Ronald; STEIN, Clifford. Introduction to Algorithms. MIT Press And McGraw-Hill, 2001.