Primeiro Trabalho Maior: Elementos fundamentais do processamento de sinais em blocos
Data de entrega: 30/09/2014 até 23:55 pelo PACA

MUITA ATENÇÃO: OS TRABALHOS SÃO INDIVIDUAIS!
Tirar dúvidas "gerais" sobre o Pd no fórum é OK,
compartilhar implementações não é OK.

O objetivo deste trabalho é adquirirmos familiaridade com o funcionamento do mecanismo de processamento de sinais em blocos utilizado na linguagem Pure Data. Reimplementaremos unidades elementares de manipulação e análise destes sinais, como os objetos `[+~]`, `[*~]`, `[osc~]` e `[rms~]`, para ganharmos intuição e traquejo na combinação de objetos DSP, no uso de vetores e estruturas de controle do fluxo de execução (laços, comandos condicionais, etc). Estas implementações serão feitas puramente através de *patches* e objetos disponíveis nas bibliotecas do Pd-extended (ou seja, estão fora de questão implementações como *externals* na linguagem C).

Do ponto de vista computacional, nossas reimplementações serão menos eficientes que suas versões nativas; senão por outra razão, pelo simples fato de que seremos obrigados a copiar os sinais dos `[inlet~]` em vetores, e gerar o sinal de saída em vetores que serão copiados para os respectivos `[outlet~]`. Mas outra vez: o objetivo é ganhar entendimento sobre o processamento em tempo real de sinais segmentados em blocos. Se durante o desenvolvimento surgir na cabeça de vocês a pergunta "isso vale a pena?", lembrem-se que Fernando Pessoa já a respondeu (*Mensagem, Segunda Parte, X – Mar Portuguez*).

1 A teoria da coisa

Sistemas de processamento de sinais em tempo real, como o Pure Data, conseguem gerar a ilusão de processamento ininterrupto a partir da coordenação de alguns elementos, dentre os quais os mais importantes são:

segmentação do fluxo em blocos: cada *patch* e *subpatch* processa fluxos de áudio segmentados em blocos de tamanho determinado; o tamanho de bloco *default* do Pd é de 64 amostras, mas qualquer *patch* ou *subpatch* pode definir outro tamanho através do objeto `[block~]`[1]. O instante correspondente a cada bloco pode ser obtido do objeto `[bang~]`, que emite um disparo a cada início de bloco.

sequenciamento dos objetos DSP: como os objetos de processamento de áudio estão normalmente conectados entre si, é fundamental estabelecer a ordem de processamento a fim de que todas as conexões de áudio em um *patch* trafeguem dados referentes ao ciclo DSP atual. O objetivo é que uma sequência de objetos DSP concatenados em série tenha como resultado a composição dos processamentos individuais aplicados à mesma entrada. O Pd resolve este problema fazendo uma *ordenação topológica* do grafo que representa as conexões de áudio, partindo do pressuposto que o grafo é acíclico (e assim a ordenação topológica é factível). Esta propriedade do grafo é forçada pelo escalonador: qualquer circuito dirigido formado por conexões DSP gera uma mensagem no console avisando que alguns processos associados a objetos DSP não serão escalonados.

escalonamento de processos e prazos: em um *patch* que envolve processamento de áudio, os objetos DSP devem respeitar o prazo associado à duração do bloco em tempo real; por exemplo, blocos de 64 amostras a 44.1 kHz duram $64/44.1 \approx 1.451247$ *ms*, e todas as computações referentes a um bloco devem acontecer em uma fração desse tempo. Teoricamente o Pd funcionaria como um escalonador para os objetos DSP, acionados através de funções de *callback*; na prática, como a intensidade computacional varia de objeto para objeto, o Pd é muito tolerante[2] em relação ao tempo utilizado por cada objeto, não impondo limites a priori.

Estes elementos, sincronizados por um ciclo específico do processamento de sinais, se relacionam também com os demais fluxos de informação que circulam entre os objetos do Pd, que têm a forma de *mensagens* e a característica de serem gerados em instantes arbitrários, não necessariamente em regime periódico. O alinhamento entre fluxos de áudio e mensagens em Pd está sucintamente descrito na seção 2.4 do manual oficial do Pd (`http://puredata.info/docs/manuals/pd`) e mais detalhadamente no capítulo 3 do livro "Theory and Techniques of Electronic Music" de Miller Puckette, disponível no link `http://msp.ucsd.edu/techniques.htm`.

[1] Este objeto permite definir também, além do tamanho do bloco, sobreposições entre blocos e superamostragem dos blocos, mas estas possibilidades não serão tratadas neste trabalho. As implementações propostas devem funcionar para qualquer tamanho de bloco, mas não precisam funcionar com sobreposição e superamostragem.

[2] Isso torna possível que um processo acabe congelando o Pd, eventualidade que será gerenciada pelo processo *pd-watchdog*, que tem poder para matar o Pd se o problema não se resolver em pouco tempo.

A fim de estabelecer uma metodologia clara de desenvolvimento deste trabalho, restringiremos os objetos DSP que poderemos usar na construção dos nossos objetos. Para copiar em vetores os dados de cada sinal da entrada, usaremos a sequência `[inlet~]` –> `[tabsend~ nomedovetor]`, que escreve a cada ciclo DSP o sinal recebido no vetor nomeado; analogamente usaremos `[tabreceive~ nomedovetor]` –> `[outlet~]` para cada saída. Além destes, o objeto `[bang~]` já mencionado dispara uma vez a cada início de bloco e será usado para iniciar cada ciclo de computação, que deve ser realizado exclusivamente por objetos de controle (objetos sem "~").

2 Primeiros passos: `[soma~]` e `[multiplica~]`

Estes objetos, que serão implementados em arquivos independentes (soma~.pd e multiplica~.pd), devem ter o funcionamento semelhante aos objetos nativos `[+~]` e `[*~]` e seguem uma mesma estrutura: copiar as duas entradas para vetores $x[n]$ e $y[n]$, construir um laço para varrer os índices de 0 a $N-1$ (N é o tamanho de bloco) e produzir um terceiro vetor $z[n]$ com os valores $x[n]+y[n]$ (`[soma~]`) ou $x[n]*y[n]$ (`[multiplica~]`). Os laços podem ser construídos "na unha" (como ilustra o *patch* 2.control.examples/05.counter.pd da documentação que acompanha o Pd) ou usando objetos mais sofisticados, como `[Uzi]`, `[until]` ou `[counter]`. Os objetos `[tabread]` e `[tabwrite]` podem ser utilizados para acessar as componentes dos vetores.

Note que todos os objetos construídos devem se ajustar ao contexto, especificamente ao tamanho de bloco N do *patch* onde estiverem inseridos (do mesmo modo que suas versões nativas `[+~]` e `[*~]`). Para isso é necessário computar o tamanho do bloco e dimensionar os vetores correspondentemente. O *patch* 5.reference/switch~-help.pd mostra (no subpatch messages-to-switch) uma maneira de medir o tempo entre blocos (que em milissegundos é igual a N/44.1) usando `[bang~]` e `[timer]`; para redimensionar vetores, veja o *patch* 2.control.examples/15.array.pd. Note que não faz sentido redimensionar o vetor a cada bloco (alocação de memória é um processo custoso): para isso você deve "filtrar" os números produzidos para só aproveitá-los quando o valor do tamanho de bloco mudar (há um jeito elegante de fazer isso usando `[trigger]` e `[sel]`).

3 Unidade geradora: `[oscilador~]`

Este objeto será parecido com o `[osc~]`, com a exceção de que não precisaremos nos preocupar com o argumento de inicialização da frequência[3]. O interesse desse objeto está no mecanismo de geração do sinal senoidal $y[n]$ da saída a partir de um sinal de entrada $x[n]$ que pode definir a frequência instantânea a cada amostra. Para isso devemos entender como relacionar $x[n]$ e $y[n]$; é importante observar que a expressão $y[n] = \cos(2\pi * x[n] * n/44100)$ falha miseravelmente (será que a razão disso ficará clara após a leitura do texto que se segue?).

Para permitir que a frequência instantânea do oscilador varie a qualquer momento, inclusive de maneira descontínua, a solução é expressar $y[n]$ em função da fase (ângulo) do oscilador no instante $n-1$, que denotaremos por $\phi[n-1]$. Assim, considerando que a frequência instantânea no intervalo de tempo $[n-1, n]$ é de $x[n]$ Hz teremos

$$\phi[n] = (\phi[n-1] + 2\pi * x[n]/44100) \mod 2\pi,$$

onde "mod 2π" é o operador que obtém o ângulo equivalente no intervalo $[0, 2\pi]$. Com isso o novo valor do oscilador será

$$y[n] = \cos(\phi[n]).$$

Apesar da fase ϕ estar indicada como se fosse um vetor, note que basta uma variável para representá-la: a atualização de $\phi[n-1]$ para $\phi[n]$ pode sobrescrever o valor anterior da mesma variável, pois ele já não será mais usado. Note também que esta variável garantirá que as emendas de bloco funcionem: o valor armazenado na variável na última amostra de um bloco será usado como $\phi[n-1]$ para calcular a fase do oscilador na primeira amostra do bloco seguinte. A declaração e inicialização dessa variável pode ser feita com um objeto `[float]`, que por *default* vale 0 quando o objeto é criado.

4 Lembrando o passado: `[amplitude~]`

O objetivo desse objeto é produzir um sinal de saída RMS$[n]$ contendo a amplitude RMS do sinal de entrada $x[n]$, definida como

$$\mathrm{RMS}[n] = \sqrt{\frac{\sum_{i=0}^{M-1} x[n-i]^2}{M}};$$

[3]Em particular, se quisermos emular `[osc~ 440]` teremos que mandar uma mensagem `|440<` para o nosso `[oscilador~]`.

aqui M representa o tamanho da janela de observação, que será passado como argumento de inicialização do objeto e acessado através da variável \$1 (por exemplo fazendo `[amplitude~ 1024]`). Observe que o valor da amplitude RMS é atualizado a cada amostra[4] a partir dos valores $x[n], x[n-1], \ldots, x[n-M+1]$.

Para cada novo bloco de tamanho N é necessário ter armazenados $M-1$ valores antigos do sinal (que entrarão no cômputo de RMS[0]), além de ser necessário armazenar os N novos valores da entrada. Para isso, será necessário dimensionar um vetor de "memória" do sinal, de tamanho $N+M-1$, que será usado como *buffer* circular. Este vetor não pode ser nem o vetor da entrada nem o vetor da saída, pois os objetos `[tabsend~]` e `[tabreceive~]` sempre copiam blocos de tamanho N a partir da posição 0 dos vetores associados.

Uma implementação ingênua da expressão RMS[n] levaria tempo proporcional a M para obter cada valor RMS[n], e tempo proporcional a $N * M$ para processar um bloco de tamanho N. Entretanto, tal cômputo pode ser radicalmente reduzido (para um custo proporcional a N por bloco) ao se observar que

$$\mathrm{RMS}[n] = \sqrt{\frac{M * \mathrm{RMS}[n-1]^2 - x[n-M]^2 + x[n]^2}{M}};$$

para viabilizar essa implementação, convém usar uma memória de tamanho $N+M$ (ao invés de $N+M-1$) e também observar que o vetor de saída já guardará para o próximo bloco o valor RMS[-1] (ou seja, o último valor RMS[$N-1$] do bloco anterior, necessário para computar o RMS[0] do bloco atual).

DICAS:

Os nomes de vetor devem ser sempre prefixados com "\$0−", que corresponde a um identificador único de cada instância do *patch*, permitindo o uso de várias cópias do mesmo objeto em um mesmo *patch*.

Para implementar o operador "mod 2π", notem que $\alpha \bmod 2\pi = \alpha - 2\pi \lfloor \frac{\alpha}{2\pi} \rfloor$ (essa expressão vale para $\alpha \geq 0$, e produz um resultado também ≥ 0).

Laços sem fim podem congelar o Pd: é sempre melhor testar antes o funcionamento dos laços com um `|bang<` (ou Ctrl-Shift-B) acionado manualmente, e só conectar o `[bang~]` quando estiverem seguros de seu funcionamento. Salvem sempre seus *patches* antes de ligar o DSP do Pd: uma vez congelado, vocês muito provavelmente perderão o trabalho realizado depois do último salvamento.

Como uma sugestão, façam o desenvolvimento sempre de forma incremental: testem separadamente cada parte do *patch*, desde a aquisição dos sinais de entrada nos vetores e cópia dos vetores de saída, passando pelos laços e contas individuais, mudanças de tamanho de bloco, etc.

Para testar seus objetos, vocês podem combinar pequenas unidades para síntese usando os objetos nativos `[+~]`, `[*~]` e `[osc~]`, e duplicar o mecanismo com os objetos `[soma~]`, `[multiplica~]` e `[oscilador~]`, comparando auditivamente os resultados. Estes *patches* podem ser compartilhados no PACA sem constrangimento: apenas certifiquem-se de não enviar junto as suas implementações dos objetos pedidos. Dica: copie o *patch* que você deseja compartilhar para um diretório diferente, rode o Pd e certifique-se que os objetos `[soma~]`, `[multiplica~]` e `[oscilador~]` ficam tracejados e geram mensagens de erro "... couldn't create".

Notem que o tamanho de bloco pode mudar no *superpatch* em tempo de execução, e não exclusivamente no momento da criação do objeto de vocês. Notem também que o tamanho de bloco do patch que contém o `[dac~]` ou `[output~]` deve ser de 64 amostras para a saída de áudio funcionar; assim, para variar o tamanho de bloco nos testes, será necessário criar algum *subpatch* dentro do *patch* de teste.

No caso do objeto `[amplitude~]` a comparação com o `[rms~]` será aproximada, já que este último só produz um valor por bloco de análise. Vocês podem inspecionar o valor produzido pelo objeto de vocês conectando-o a um `[snapshot~]`, que deve ser acionado com um `|bang<` para produzir o valor da amostra.

Bom trabalho!

[4] Isso difere dos objetos nativos `[env~]` e `[rms~]` que só produzem um valor de amplitude para cada janela de análise.