
 Universidade do Minho	Módulo 1 Introdução ao caso de estudo: Convolução	
--	--	---

Convolução

Este módulo apresenta o caso de estudo que será usado ao longo deste semestre. Trata-se de uma operação de convolução que aplica um filtro a todos os píxeis de uma imagem.

Cada píxel na imagem final será igual à média dos 9 píxeis originais na sua vizinhança. Esta operação é decomposta em 2 passagens: primeiro calcula-se a média com o vizinho da esquerda e da direita (filtro horizontal), e, na 2ª passagem calcula-se a média com os vizinhos de cima e de baixo (filtro vertical).

Uma imagem é representada no mundo digital como um *array* bidimensional com uma determinada largura (*W* de *width*) e altura (*H* de *height*). Se a cor de cada ponto da imagem, designado por píxel, for representado por um inteiro, em C será algo declarado como:

```
int i[H][W];
```

Em termos muito simples a convolução consiste em aplicar um filtro a CADA píxel da imagem *i*, gerando uma nova imagem *h*. Matematicamente a convolução é representada pelo operador ***, logo:

```
int h[H][W];
h = i * f; /* NOTA: aqui * representa a convolução;
               Este operador não existe em C e não deve ser
               confundido com a multiplicação */
```

A Figura 1 ilustra como é calculado cada elemento $h[y][x]$, concretizando para o caso particular de $h[4][3]$. Imagine que o filtro *f* está centrado no píxel $i[y][x]$ da imagem (neste caso $i[4][3]$) e de seguida cada elemento de *i*, que é sobreposto pelo filtro, é multiplicado pelo respetivo elemento de *f*. A soma de todos estes produtos é o valor a atribuir a $h[y][x]$.

NOTA: na verdade existe uma subtilidade relativamente ao filtro. É suposto a soma de todos os valores de *f* ser igual a 1 – diz-se que o filtro está normalizado. Como declaramos o filtro *f* como sendo um *array* de inteiros dificilmente a soma dos seus elementos pode ser 1. É necessário somar os valores de todos os elementos do filtro e, antes de escrever o resultado descrito no parágrafo anterior em $h[y][x]$, dividi-lo por esta soma. A soma de todos os elementos do filtro chama-se a constante normalizadora. No caso da média de 3 píxeis esta constante é 3.

A equação abaixo apresenta em notação matemática, e para o caso de um filtro de 3x1, a expressão da convolução. Note que esta operação é repetida para todos os píxeis da imagem.

$$h[y][x] = (i[y][x-1] + i[y][x] + i[y][x+1]) / 3;$$

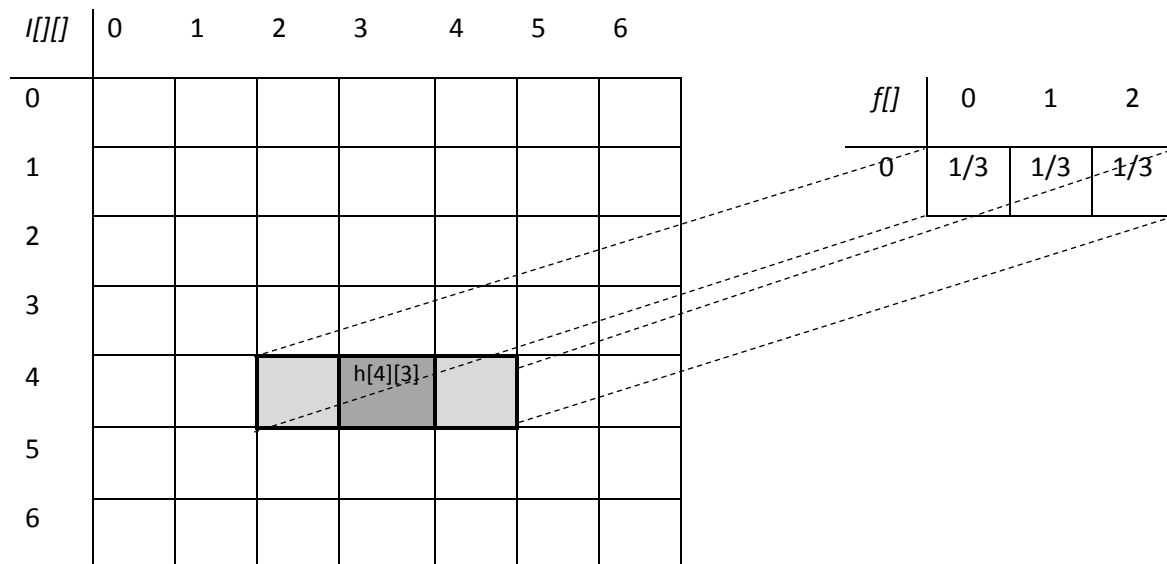


Figura 1 - Região de i (imagem) usada para calcular $h[4][3]$: é usada uma janela com a dimensão 3x1, isto é, a mesma dimensão do filtro $f[]$.

Resumindo, o valor de $h[y][x]$ é dado por uma média pesada da vizinhança de $i[y][x]$, sendo que os pesos são os valores de f e a dimensão da vizinhança é igual à dimensão de f .

```

1
2 void convolve (int h[], int i[], int W, int H, int buf[]) {
3     register int x, y;
4
5     // filtro horizontal
6     for (x=1 ; x<(W-1) ; x++) { // for each column of I
7         for (y=0 ; y<H ; y++) { // for each row of I
8             buf[y*W+x] = (i[y*W+x-1] + i[y*W+x] + i[y*W+x+1]) / 3;
9         } // y loop
10    } // x loop
11
12    // filtro vertical
13    for (x=1 ; x<(W-1) ; x++) { // for each column of buf
14        for (y=1 ; y<(H-1) ; y++) { // for each row of I
15            h[y*W+x] = (buf[(y-1)*W+x] + buf[y*W+x] + buf[(y+1)*W+x]) / 3;
16        } // y loop
17    } // x loop
18 }

```

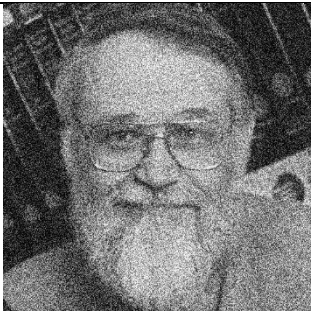


O excerto de código acima implementa a convolução 3x3 sem qualquer otimização. Algumas notas sobre este código:

- A função `convolve()` recebe como parâmetros os apontadores para a zona de memória (*buffer*) onde deve ser guardado o resultado (h) e o *buffer* onde se encontra a imagem original (i). Recebe também as dimensões da imagem (W e H) e ainda um *buffer* (buf) para armazenar os resultados intermédios, isto é, o resultado da aplicação do filtro horizontal.
- Apesar de os *buffers* h , buf e i conterem dados que sabemos representarem quantidades bidimensionais o acesso é feito considerando cada *buffer* como uma estrutura unidimensional. Assim $h[y][x]$ é acedido como $h[y*W+x]$, sendo y a linha a aceder, x a coluna e W a largura (número de colunas) da estrutura bidimensional;

O filtro apresentado é do tipo passa-baixo e elimina altas frequências, isto é, tende a suavizar as imagens diluindo os contornos dos objetos. A Tabela 1 exemplifica a aplicação deste filtro à imagem

de Brian Kernighan (um dos inventores originais da linguagem C). A imagem original foi contaminada com ruído; os filtros eliminam parcialmente esse ruído. Nota que quanto maior a área do filtro, mais ruído é eliminado. Como não há almoços de borla, filtros maiores atenuam de forma mais significativa os contornos dos objetos e levam mais tempo a ser executados.

Tabela 1 - Exemplos de aplicação da convolução.

	
Imagem Brian Kernighan com ruído branco	
	
Box filter (média) de 3x3	Box filter (média) de 11x11

Sessão Laboratorial

Arranque a sua máquina usando a imagem do Fedora Core 13, inicie sessão usando as credenciais abaixo:

User: diguest

Pass: diguest

e inicie o gestor de janelas (`startx`).

Descarregue da Blackboard o código associado a este módulo e construa o executável (`make`).

O executável gerado permite aplicar a convolução atrás discutida a uma imagem, no formato PPM (extensão `.pgm` que é o sub-formato para gray scale).

Corra o executável escrevendo:

```
./convolve AC_images/brian_kernighan_noisy.pgm result.pgm
```

No ecrã foi apresentado o tempo de execução em micro-segundos (`usecs`). Visualize a imagem gerada e que está armazenada no ficheiro `result.pgm` (use por exemplo a aplicação `gimp` do

Linux ou o próprio interface gráfico do gestor de ficheiros para invocar a aplicação de visualização configurada por omissão).

1. Ao longo deste semestre usaremos a biblioteca PAPI (**P**erformance **A**pplication **P**rogramming **I**nterface) para obter várias informações sobre o desempenho dos nossos programas.

Examine o código da função `main()` (em `main.cpp`) e descreva a forma como estamos a medir o tempo de execução da rotina `convolve()`. Note que a função `PAPI_get_real_usec()` devolve o tempo real (*wall clock time*) em micro-segundos (usecs) decorrido desde um dado momento de referência desconhecido.

2. O código fornecido mede o tempo de execução de `convolve()` apenas uma vez. Esta não é a metodologia mais adequada. Modifique o código da função `main()` introduzindo um ciclo que permita medir o tempo de execução de `convolve()` *N* vezes (*N* definido com um `#define`) e que reporte uma estatística mais apropriada (exemplo: média, mínimo ou mediana).
3. É muitas vezes útil reportar o desempenho de um programa em função do número de ciclos de relógio consumidos, em vez do tempo absoluto. Sabendo que a função `PAPI_get_real_cyc()` devolve o número de ciclos do relógio decorridos desde um dado momento de referência desconhecido, modifique o seu programa para que, além do tempo, reporte também o número de ciclos.
4. O número absoluto de ciclos medido na alínea anterior poderá não ser muito útil. Uma métrica mais útil é o *CyclesPerElement* (CPE), isto é, o número de ciclos do relógio consumidos por elemento do conjunto de dados. No caso das imagens um elemento é um pixel. Modifique o seu código por forma a reportar, além do tempo e do total de ciclos, o CPE. Anote os resultados para imagens de diferentes tamanhos (exemplo: `AC_images/brian_kernighan_noisy.pgm` e `AC_images/VolcanosSunsetBW.pgm`)
5. Altere a `Makefile` para passar a usar o nível de optimização `-O2`. Recompile o código (não se esqueça de escrever `make clean` antes de `make`) e meça de novo os mesmos resultados da alínea anterior. Comente.
6. Considere o código da função `convolve()` (em `convolve.cpp`). Concorda que se obtém um programa funcionalmente equivalente se trocarmos a ordem dos ciclos `for()`, isto é, em vez de fazer uma travessia *column-wise* da imagem (índice *x* no ciclo externo) fizermos uma travessia *row-wise* (índice *y* no ciclo externo)? Altere a ordem dos ciclos `for`, recompile e comente os resultados.