Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Departamento de Engenharia Elétrica

Relatório para a Disciplina Projeto em Engenharia Elétrica

Projeto e Implementação de Filtros Bi-Dimensionais em MatLab e C para Aritmética de Ponto Flutuante

Jayarama Sundar Santana
jayaramasantana@yahoo.com

Orientador:
Angelo Perkusich

Campina Grande, Fevereiro de 2006

Biblioteca Setorial do CDSA. Fevereiro de 2021.

Sumé - PB

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Departamento de Engenharia Elétrica

Relatório para a Disciplina Projeto em Engenharia Elétrica

Projeto e Implementação de Filtros Bi-Dimensionais em MatLab e C para Aritmética de Ponto Flutuante

Jayarama Sundar Santana
(Aluno)

Angelo Perkusich
(Orientador)

Lista de Abreviaturas e Símbolos

DSP – *Digital Signal Processor* (Processador Digital de Sinal)

IEEE – *Institute of Electrical and Electronics Engineers*

MSE – *Mean Squared Error* (Erro Médio Quadrático)

CMOS – *Complementary Metal Oxide Semiconductor*

CCD – *Charge Coupled Device*

RAM – *Random Access Memory*

Pixel – *Picture Element*

$f(i,j)$ – imagem de referência no ponto (i,j)

$r(i,j)$ – ruído no ponto (i,j)

$a(i,j)$ – versão degradada da imagem de referência no ponto (i,j)

$\hat{f}(i,j)$ – estimativa da imagem de referência no ponto (i,j)

η – vizinhança local de dimensão N x M pixels

$\mu(n,m)$ – média local no ponto (n,m)

$\sigma^2(n,m)$ – variância local no ponto (n,m)

v^2 – variância do ruído

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Programas

1. Introdução

Em geral, o processo de aquisição de um sinal é também acompanhado de ruído, o que distorce a informação original. Em especial, sensores de imagem utilizados em câmeras digitais adicionam um ruído característico à imagem. A fim de reduzir o efeito do ruído, pode-se recorrer à aplicação de filtros digitais bidimensionais [1].

Os filtros digitais bidimensionais realizam cálculos numéricos, os quais podem ser realizados por um computador de propósito geral ou Processadores Digitais de Sinais (DSPs), especializados para tal função.

A fim de se realizar o projeto e implementação de algoritmos para a plataforma de Processadores Digitais de Sinais (DSPs) baseados em aritmética de ponto fixo, pode-se realizar os seguintes passos [2]:

1) Formulação matemática dos algoritmos a serem implementados;
2) Implementação e simulação dos algoritmos em linguagens de alto nível, como o MatLab [3], e posterior implementação em C usando aritmética de ponto flutuante;
3) Implementação e simulação dos algoritmos em linguagens de alto nível, como o MatLab, e posterior implementação em C usando aritmética de ponto fixo;
4) Implementação do código gerado em C usando aritmética de ponto fixo para a plataforma final (DSP de ponto fixo);
5) Utilização de métricas de desempenho.

O uso de uma linguagem de alto nível, tal como o MatLab, e o uso da aritmética de ponto flutuante facilita a verificação dos algoritmos e possibilita maior precisão nos cálculos. Em seguida, realiza-se o porte dos algoritmos da aritmética de ponto flutuante para a aritmética de ponto fixo, tendo em vista o lançamento do aplicativo na plataforma final (DSP).

Esse projeto está dividido em três etapas, a citar:

1) Projeto e implementação de filtros bi-dimensionais em MatLab e C para a aritmética de ponto flutuante;
2) Projeto e implementação de filtros bi-dimensionais em MatLab e C para aritmética de ponto fixo;
3) Aplicação de métricas de desempenho para os filtros digitais;

Na presente atividade, trataremos da primeira etapa do projeto. Dentre os algoritmos de redução de ruído, analisaremos os filtros da média, mediana e Wiener adaptativo.

2. Objetivos

2.1. Objetivos Gerais

Obter conhecimentos acerca do padrão de imagem bitmap de 24-bits, além de projetar e implementar filtros digitais bi-dimensionais.

2.2. Objetivos Específicos

Objetivamos implementar rotinas para a leitura e escrita de arquivos bitmap de 24 bits e três filtros digitais para a redução de ruído em imagens, a citar: filtros da média, mediana e Wiener adaptativo. As implementações serão realizadas em ambiente MatLab e linguagem C utilizando aritmética de ponto flutuante. Finalmente, avaliaremos o desempenho dos filtros em relação ao tempo de processamento.

3. Representação de Números em Ponto-Flutuante

A representação de números reais em ponto-flutuante de precisão simples segundo o IEEE *(Institute of Electrical and Electronics Engineers)* utiliza 32 bits, sendo 1 bit para o sinal, 8 bits para o expoente e 23 bits para a mantissa. A mantissa é um número real no intervalo [1, 2) armazenado como um número binário em potências negativas de 2. A parte inteira (o número real 1) é subentendida. Já o expoente é um número no intervalo de -126 a 127 armazenado com uma polarização de 127, isto é, como um número binário no intervalo de 00000001 a 11111110. Os valores

correspondentes ao expoente igual a 0 ou 255 são reservados para informações adicionais. Logo, a conversão de um número em ponto flutuante de precisão simples para um número real é feita segundo a equação 1 [4].

$$\text{Valor Real} = (-1)^{sinal} \cdot Mantissa \cdot 2^{Expoente-Polarização} \qquad \textbf{(Equação 1)}$$

Por exemplo, consideremos o seguinte número binário de 32 bits:

0 10000111 10101000000000000000000

Podemos notar que:

- O valor real correspondente é um número positivo, pois o primeiro bit é 0;
- O expoente é: $(10000111)_2 = (135)_{10}$;
- A mantissa é: $(10101000000000000000000)_2 = 2^{-1} + 2^{-3} + 2^{-5} = (0{,}65625)_{10}$, que corresponde a $(1{,}65625)_{10}$, pois a parte inteira igual a 1 é subentendida;
- A polarização é: $(127)_{10}$.

Logo, o número real correspondente vale:

$$(-1)^{sinal} \cdot Mantissa \cdot 2^{Expoente-Polarização} = (-1)^0 \cdot 1{,}65625 \cdot 2^{135-127} = 424{,}0$$

A representação de números reais em ponto-flutuante de precisão dupla é similar à representação com precisão simples. A diferença é que são usados 11 bits para o expoente, que varia de -1022 a 1023 usando uma polarização de 1023, e 52 bits são usados para representar a mantissa.

A vantagem da representação de números em ponto flutuante é que números muito pequenos e muito grandes podem ser representados sem grande perda de precisão. Por exemplo, uma variável em ponto flutuante de precisão simples pode representar números Reais no intervalo de $1{,}175494351 \cdot 10^{-38}$ a $3{,}402823466 \cdot 10^{+38}$. Logo, cuidados especiais na implementação do algoritmo, tais como *overflow* e *underflow* de variáveis não são geralmente necessários.

4. Padrão de Imagem Bitmap de 24-bits

No padrão de imagem bitmap de 24-bits, cada unidade de imagem ou pixel (*Picture Element*) é representada no formato RGB (*Red, Green, Blue*). Este consiste da combinação de três cores primárias: Vermelho, Verde e Azul, onde cada cor é representada por um byte (8 bits), logo cada pixel é composto de três bytes ou 24 bits.

A unidade de imagem é representada matematicamente por uma terna (r, g, b), onde cada componente pode variar de 0 a 255. As ternas (0, 0, 0) e (255, 255, 255) representam as cores preta e branca, respectivamente.

Um arquivo de imagem bitmap precisa de um cabeçalho ou *header* onde se encontram informações básicas acerca do mesmo. No padrão bitmap, os primeiros 54 bits constituem o cabeçalho. Na tabela 1, são apresentadas as informações presentes no cabeçalho para cada faixa de bytes.

Tabela 1: Cabeçalho de um arquivo bitmap de 24-bits

Número do byte de início	**Informação**
0	Assinatura
2	Tamanho do arquivo
18	Número de colunas
22	Número de linhas
28	Bits/pixel
46	Número de cores usadas
54	Início dos dados

Note-se que o campo número de cores usadas não é relevante em se tratando de uma imagem com o valor Bits/pixel > 8.

É interessante ressaltar que o número do byte de início representa o byte menos significativo, logo para computar o valor de cada campo, deve-se utilizar a equação 2.

$$Valor = B_0 \cdot 256^0 + B_1 \cdot 256^1 + ... + B_{n-1} \cdot 256^{n-1} + B_n \cdot 256^n \quad \textbf{(Equação 2)}$$

A fim de se ler os valores dos bytes no campo dados, é necessário criar dois laços sucessivos: o primeiro para varrer as linhas e o segundo para varrer as colunas. Fixas a linha e coluna, os bytes devem ser lidos na ordem B, G, R. Esse mapa é esquematizado na figura 1.

Colunas

Linhas	0	1	2	.	.	.	M
N	B G R	.	.	.	.	.	B G R
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
2	.	.	.	.	.	.	.
1	.	.	.	.	.	.	.
0	B G R	.	.	.	.	.	B G R

Figura 1: Campo de dados no padrão bitmap de 24-bits

5. Rotina para Leitura/Escrita de Arquivos Bitmap de 24 bits

Nesta seção, apresentamos rotinas para leitura de arquivos bitmap de 24 bits, o armazenamento das componentes R, G e B em matrizes e sua posterior escrita em um arquivo de saída. Notemos que com as matrizes armazenadas, é possível realizar as filtragens a serem discutidas.

Por hora, não incluímos rotinas para interface com o usuário, as quais serão introduzidas posteriormente.

5.1. Implementação em MatLab

É apresentado a seguir o programa **entradaSaida.m** para manipulação de arquivos de imagem bitmap de 24 bits em MatLab.

Programa 1: entradaSaida.m

```
**********************************************************************************************************
clear
clc

%Inicialização do contador
tic;

%Leitura do arquivo de imagem
RGB = single(imread('entrada.bmp'));

%Armazenamento das componentes R, G e B em matrizes
R=RGB(:,:,1);
G=RGB(:,:,2);
B=RGB(:,:,3);

%Escrita das componentes R, G e B em uma matriz RGB
clear RGB;
RGB(:,:,1)=R;
RGB(:,:,2)=G;
RGB(:,:,3)=B;

%Escrita da matriz RGB em um arquivo
imwrite(uint8(RGB),'saida.bmp','bmp')

%Contagem e impessão do tempo de processamento
t=toc;
sprintf('Tempo para processamento de imagem: %g', t)
**********************************************************************************************************
```

Notemos que o programa utiliza essencialmente as funções imread() e imwrite() para a manipulação do arquivo de imagem. Ressalta-se que a manipulação com os dados do arquivo torna-se transparente através dessas funções, logo é dispensado o cuidado necessário com a manipulação dos dados do arquivo em nível de bytes.

A função imread() tem a seguinte assinatura:

uint8 *RGB* = imread('*nome.bmp*');

Esta função lê o arquivo *nome.bmp* do diretório corrente e armazena as componentes R, G e B na matriz *RGB*, que é do tipo uint8 (ponto-fixo de 8 bits). Esta matriz é de dimensão N x M x 3, onde a componente R é representada por (N,M,1), a componente G por (N,M,2) e a componente B por (N,M,3). Notemos que no programa usamos a função single() sobre o resultado de imread() a fim de converter os dados para ponto-flutuante de precisão simples, o que será útil para o ganho de precisão ao serem feitas manipulações com esses dados.

A função imwrite() tem a seguinte assinatura:

imwrite(*RGB*, '*nome.bmp*', 'modo');

Esta função escreve a matriz de imagem *RGB* (no formato uint8) no arquivo *nome.bmp*. O parâmetro 'modo' é 'bmp' quando tratamos de arquivos bitmap. Notemos que no programa usamos a função uint8() sobre RGB a fim de converter os dados de ponto-flutuante para a formato uint8.

Por fim, utilizamos também um contador no programa utilizando as funções tic e toc, a fim de estabelecer métricas de desempenho. A função tic inicia o temporizador e a função toc retorna o tempo decorrido desde o último tic.

5.2. Implementação em Linguagem C

É apresentado a seguir o programa **entradaSaida.c** para manipulação de arquivos de imagem bitmap de 24 bits em C.

Programa 2: entradaSaida.c

```
*****************************************************************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

//Protótipos das funções
long getInfoImagem(FILE*, long, int);

void main()
{

        FILE *bmpInput, *bmpOutput;
        int rows, columns, r, c;
        long fileSize;
        double initClock, time;

        //Inicia contagem do número de ticks de clock
        initClock=clock();

        //Abrindo arquivos de entrada e saída
        bmpInput = fopen("entrada.bmp", "rb");
        bmpOutput = fopen("saida.bmp", "wb");

        //Computando dados do arquivo BMP de entrada
        columns = (int)getInfoImagem(bmpInput, 18, 4);
        rows = (int)getInfoImagem(bmpInput, 22, 4);
        fileSize = getInfoImagem(bmpInput, 2, 4);

        //Testando se o tamanho do arquivo de entrada é o esperado
```

```
if(fileSize!=(rows*columns*3+54)){
        printf("\nArquivo de entrada incorreto ou com dimensões desalinhadas...\n");
        exit(0);
}


//Copia cabeçalho do arquivo de entrada para o arquivo de saída
{//novo escopo
        unsigned char buffChar[54];
        fseek(bmpInput, 0, SEEK_SET);
        fseek(bmpOutput, 0, SEEK_SET);

        fread(buffChar, sizeof(char), 54, bmpInput);
        fwrite(buffChar, sizeof(char), 54, bmpOutput);
}//fim do novo escopo


//Posicionando os apontadores bmpInput e bmpOutput para o setor de dados
fseek(bmpInput, 54, SEEK_SET);
fseek(bmpOutput, 54, SEEK_SET);

{//novo escopo

        //Alocação Dinâmica de Memória para os arrays Red, Green, Blue e buffChar
        unsigned char *redValue =(unsigned char *) malloc((sizeof(unsigned
char)*rows*columns));
        unsigned char *greenValue = (unsigned char *)malloc((sizeof(unsigned
char)*rows*columns));
        unsigned char *blueValue = (unsigned char *)malloc((sizeof(unsigned char) *
rows * columns));
        unsigned char *buffChar = (unsigned char *)malloc((sizeof(unsigned char) *
rows * columns * 3));

        //Inicialização do indexador de buffChar
        long index = 0;

        //Leitura e armazenamento dos dados
        fread(buffChar, sizeof(unsigned char), rows*columns*3, bmpInput);

        for(r=0; r<=rows - 1; r++){
                for(c=0; c<=columns - 1; c++){

                        int tmp = r*columns + c;

                        blueValue[tmp] = buffChar[index];
                        index++;

                        greenValue[tmp] = buffChar[index];
                        index++;

                        redValue[tmp] = buffChar[index];
                        index++;
                }
        }

        //Reinicialização do indexador de buffChar
        index=0;

        //Escrita dos Arrays Red, Green e Blue ao arquivo de saída
        for(r=0; r<=rows - 1; r++){
```

```
                    for(c=0; c<=columns - 1; c++){

                        int tmp = r*columns + c;
                        buffChar[index]=blueValue[tmp];
                        index++;
                        buffChar[index]=greenValue[tmp];
                        index++;
                        buffChar[index]=redValue[tmp];
                        index++;
                    }
                }
                fwrite(buffChar, sizeof(unsigned char), rows*columns*3, bmpOutput);

        }//fim do novo escopo

        //Fechamento dos arquivos
        fclose(bmpInput);
        fclose(bmpOutput);

        //Cálculo e impressão do tempo gasto para a filtragem
        time=((double)clock()-(double)initClock)/(double)CLK_TCK;
        printf("\nTempo de processamento da imagem: %.3f segundos\n", time);

}

//Rotina de leitura de dados do cabeçalho da imagem
long getInfoImagem(FILE* inputFile, long offset, int numberOfChars)
{
        unsigned char *ptrC;
        long value = 0L;
        unsigned char tmpChar;
        int i;
        float power=1.0;

        ptrC = &tmpChar;

        fseek(inputFile, offset, SEEK_SET);

        //Cálculo do valor por adição de bytes
        for(i=0; i<numberOfChars; i++)  {
           fread(ptrC, sizeof(char), 1, inputFile);
                value = (long)(value + (*ptrC)*(power));
                power=power*256;
        }
        return(value);

}
*****************************************************************************************************************
```

Notemos que em linguagem C, a rotina para a manipulação de arquivos bitmap foi feita em nível de bytes, o que exige conhecimento detalhado acerca do padrão bitmap de 24-bits. Para tal manipulação, utilizamos funções da biblioteca padrão <stdio.h>, a citar fopen(), fclose(), fseek(), fread() e fwrite() [13].

A função fopen() tem a assinatura:

***stream* FILE * fopen (const char *** *filename*, **const char *** *mode*);

Isto é, a função retorna um apontador para o arquivo (*stream*) e os argumentos são o nome do arquivo (*filename*) e o modo (*mode*). No programa usamos os modos "rb", que indica que o arquivo deve ser aberto para leitura em modo binário, e o modo "wb", que indica que o arquivo deve ser aberto para escrita em modo binário.

A função fclose() tem a assinatura:

int fclose (FILE * *stream*);

Esta função fecha o arquivo especificado pelo *stream* ou apontador para o arquivo depois de realizar um *flush* de todos os buffers associados a ela. O retorna da função vale **0** se for realizada com sucesso e **EOF** caso ocorra algum erro.

A função fseek() tem a assinatura:

int fseek (FILE * *stream* **, long** *offset* **, int** *origin* **);**

Esta função posiciona o apontador para arquivo *stream* na posição definida por *offset* a partir da localização dada por *origin*. O retorno da função vale **0** caso a operação tenha sido realizada com sucesso e $\neq$ **0** caso ocorra um erro. Existem três parâmetros para *origin*:

- SEEK_SET – *offset* deve ser calculado a partir do começo do arquivo;
- SEEK_CUR – *offset* deve ser calculado a partir da posição atual do apontador de arquivo;
- SEEK_END - *offset* deve ser calculado a partir do final do arquivo;

A função fread() tem a assinatura:

int fread (void * *buffer*, **size_t** *size*, **size_t** *count*, **FILE *** *stream*);

Esta função lê a partir do apontador de arquivo *stream* e transfere para o *buffer* um número de itens igual a *count* e do tamanho especificado por *size*. Logo, o número de bytes transferidos é igual a *count*size*. O valor de retorno da função é o número de itens lidos. No programa utilizamos um buffer adicional *buffChar* a fim de armazenar todo o campo de dados do arquivo da entrada com uma única chamada de fread(). Esse procedimento gasta mais memória, contudo diminui o número de chamadas da função fread(), logo economizando tempo.

Por fim, a função fwrite() tem a assinatura:

size_t fwrite (const void * *buffer*, size_t *size*, size_t *count*, FILE * *stream*);

Esta função escreve um número de itens iguais a *count* e do tamanho especificado por *size* a partir do bloco de memória apontado por *buffer* para a posição no arquivo apontada por *stream*. No programa escrevemos o conteúdo de *buffChar* com uma única chamada da função fwrite(), logo economizando tempo de processamento.

A fim de estabelecer um temporizador, utilizamos a biblioteca <time.h>. A função clock() retorna o número de *ticks* de relógio desde o início da execução do programa. Já o parâmetro CLK_TCK é uma constante que representa o número de *ticks* por segundo. Logo, para sabermos tempo gasto para executar determinada região do programa, basta fazermos:

$$\text{Tempo} = \frac{\text{ticks atual - ticks referência}}{\text{CLK\_TCK}} \qquad \textbf{(Equação 3)}$$

Vale ressaltar que os apontadores para as componentes R, G e B usados nesse programa são unidimensionais. Entretanto, podemos acessar os valores dos "dados bidimensionais" correspondentes à i-ésima linha e j-ésima coluna do seguinte modo:

$$Valor = *(ptrArray + i*colunas + j) \qquad \textbf{(Equação 4)}$$

Ou de maneira mais simplificada:

$$Valor = ptrArray[i*columns + j] \qquad \textbf{(Equação 5)}$$

6. Filtros Digitais para Imagem

Consideremos que a imagem de referência e o ruído são representados por $f(i, j)$ e $r(i, j)$, respectivamente e que $a(i, j)$ é uma versão degradada da imagem pelo ruído, isto é, $a(i, j) = f(i, j) + r(i, j)$. Com a finalidade de reduzir o ruído em imagens, podemos recorrer ao uso de um filtro digital de imagem, denotado pela transformação $T\{\cdot\}$ [5, 7]. É importante ressaltar que, na maioria dos casos, somente conhecemos o sinal $a(i, j)$, isto é, a versão degradada da imagem de referência. Logo a saída do filtro digital pode ser representada por:

$$\hat{f}(i, j) = T\{a(i, j)\} \qquad \textbf{(Equação 6)}$$

Onde $\hat{f}(i, j)$ representa o sinal de saída, isto é, uma estimativa da imagem de referência.

O desempenho do filtro digital está fortemente associado aos modelos utilizados para caracterizar a imagem e o ruído. Os filtros não-adaptativos, como os da média e mediana, em geral suprimem as componentes de alta freqüência da imagem, logo produzindo suavização. Já os filtros adaptativos, como o de Wiener, tendem a preservar as características originais da imagem, como as bordas e detalhes.

Nesse estudo, estamos considerando dois tipos de ruído: o ruído aditivo Gaussiano branco, comum em sensores de imagem (especialmente o sensor CMOS), e o ruído impulsivo (*salt and pepper*), comuns em erros de bit de transmissões e *pixels* com mal-função em sensores [12].

Nos itens seguintes, abordamos a implementação de três filtros digitais, a citar: filtros da média, mediana e Wiener adaptativo.

6.1. Filtro da Média

O filtro digital da média consiste em substituir cada *pixel* da imagem pela média dos *pixels* em uma vizinhança N x M. Notemos que se trata de uma filtragem bastante simples e que ocorre um processo de suavização, isto é, supressão das componentes de alta freqüência do sinal. Em se tratando de uma imagem colorida, esse processamento deve ser feito independentemente para cada componente de cor. Esse filtro pode ser representado pela equação 7.

$$\hat{f}(n,m) = \frac{1}{NM} \sum_{i,j \in \eta} a(i,j) \qquad \textbf{(Equação 7)}$$

Onde $\hat{f}(n,m)$ é a saída do filtro no pixel (n.m), *a(i,j)* é a entrada do filtro e η é uma vizinhança local N x M ao redor do *pixel (n,m)*. Note-se que $i, j \varepsilon \eta$, ou seja, varrem a vizinhança local N x M em torno do pixel (n,m). Notemos que utilizamos uma notação simplificada na equação 7, equivalendo à equação 8.

$$\hat{f}(n,m) = \frac{1}{NM} \sum_{i=n-floor\left(\frac{N-1}{2}\right)}^{i=n+ceil\left(\frac{N-1}{2}\right)} \sum_{j=m-floor\left(\frac{M-1}{2}\right)}^{j=m+ceil\left(\frac{M-1}{2}\right)} a(i,j) \qquad \textbf{(Equação 8)}$$

Onde *ceil(x)* representa o arredondamento de x para o inteiro mais próximo maior ou igual a x, e *floor(x)* representa o arredondamento de x para o inteiro mais próximo menor ou igual a x.

6.1.1. Implementação em MatLab

É apresentada a seguir a rotina **filtroMedia.m** que implementa a função de filtro da média em MatLab. Essa função tem a seguinte assinatura:

***Media* = filtroMedia(*G, window*);**

Onde *G* é a matriz bidimensional de entrada, *window* é a dimensão da janela e *Media* é a matriz bidimensional que representa o retorno da função. Por praticidade, estamos considerando uma vizinhança local *window* x *window*.

Programa 3: filtroMedia.m

```
**********************************************************************************************************
function mean=filtroMedia(g,window)

N = size(g,1);
M = size(g,2);

invsqwindow=single(1/(window*window));

mean=single(zeros(N,M));

%Definição dos limites da janela
lowerWindow=floor((window-1)/2);
upperWindow=ceil((window-1)/2);

%Laços que varrem as linhas e colunas
for n=1:N
   for m=1:M
      %Laços para a soma dos pixels em uma vizinhança local window x window
      for i=n-lowerWindow:n+upperWindow
         for j=m-lowerWindow:m+upperWindow
            %Teste para verificar se o pixel excede as bordas
            if(~(i<1 || j<1 || i>N || j>M))
               mean(n,m)=mean(n,m)+g(i,j);
            end
         end
      end
      %Cálculo da média
      mean(n,m)=mean(n,m)*invsqwindow;
   end
end
**********************************************************************************************************
```

6.1.2. Implementação em Linguagem C

É apresentada a seguir a rotina **filtroMedia.c** que implementa a função de filtro da média em C. Essa função tem a seguinte assinatura:

void filtroMedia(unsigned char **Array*, int *window*, int *rows*, int *columns*)

Onde **Array* é um apontador para uma lista de valores, *window* é a dimensão da vizinhança *window* x *window*, *rows* são o número de linhas e *columns* são o número de colunas. Notemos que **Array* é modificado pela função, passando a conter o resultado da filtragem da média.

Programa 4: filtroMedia.c

```
*****************************************************************************************************
#include <stdlib.h>

//Rotina do filtro da Média
void filtroMedia(unsigned char *inpArray,int window, int rows, int columns){

        //Definição de variáveis
        int i,j,n,m,tmp,tmp2,lowerWindow,upperWindow;
        int N = rows;
        int M = columns;
        float invsqwindow;
        float mean;

        //Definição de parâmetro
        invsqwindow=1/(float)(window*window);

        //Definição dos limites da janela
        lowerWindow=(int)(window-1)/2;
        upperWindow=(int)((window-1)/2.0+0.5);

        //Laços que varrem as linhas e colunas
        for (n=0;n<N;n++){
                for (m=0; m<M; m++){
                        //Laços para a soma dos pixels em uma vizinhança local
                        // window x window
                        tmp=n*columns+m;
                        mean=0;
                        for(i=n-lowerWindow; i<=n+upperWindow;i++){
                                for(j=m-lowerWindow; j<=m+upperWindow;j++){
                                        //Teste para verificar se o pixel excede as bordas
                                        if(!(i<0 || j<0 || i>N-1 || j>M-1)){
                                                tmp2=i*columns + j;
                                                mean=mean+(float)inpArray[tmp2];
                                        }
                                }
                        }
                        //Cálculo da média e arrendondamento (adicionar 0.5
                        //e realizar o typecast para int)
                        inpArray[tmp]=(unsigned char)(mean * invsqwindow+0.5);

                }
        }
}
*****************************************************************************************************
```

6.2. Filtro da Mediana

O filtro da mediana baseia-se em uma técnica de processamento de sinais não-linear em que cada *pixel* é substituído pela mediana em uma vizinhança N x M [1]. Este filtro é particularmente útil na supressão de ruídos impulsivos, como por exemplo, o "*salt and pepper*". Esse filtro pode ser representado pela equação 9.

$$\hat{f}(n,m) = MED\{a(i,j)\}, \text{ com } i,j \in \eta \qquad \textbf{(Equação 9)}$$

Onde $\hat{f}(n,m)$ é a saída do filtro, *a(i,j)* é a entrada do filtro e $MED\{a(i,j)\}$ representa a mediana tomada em η, isto é, em uma vizinhança local N x M ao redor do *pixel (n,m).*

A mediana de uma seqüência discreta de N valores, para N ímpar, é o valor de um elemento pertencente à seqüência tal que $\frac{N-1}{2}$ elementos são menores ou iguais ao valor e $\frac{N-1}{2}$ elementos são maiores ou iguais ao valor.

Por exemplo, consideremos a matriz:

$$A = \begin{bmatrix} 9 & 2 & 3 \\ 6 & 8 & 4 \\ 7 & 5 & 1 \end{bmatrix}$$

Logo $MED\{a(2,2)\}$ em uma vizinhança 3 x 3 é 5.

Para calcular a mediana computacionalmente, basta ordenar uma seqüência e calcular o seu valor central. Caso a seqüência de valores seja par, não existe valor central, mas podemos arbitrar a mediana como sendo um dos dois elementos do par central da seqüência.

6.2.1. Implementação em MatLab

É apresentada a seguir a rotina **filtroMediana.m** que implementa o filtro da mediana em MatLab. Essa função tem a assinatura similar ao do filtro da média e é apresentada a seguir.

***Mediana* = filtroMediana(*G*, *window*);**

Programa 5: filtroMediana.m

```
**********************************************************************************************************
function median=filtroMediana(g,window)

N = size(g,1);
M = size(g,2);

invsqwindow=single(1/(window*window));
median=single(zeros(N,M));

%Array temporário para armazenamento de valores
tempArray=zeros(window*window,1);
tmpArrayIndex=1;

%Definição dos limites da janela
lowerWindow=floor((window-1)/2);
upperWindow=ceil((window-1)/2);

%Laços que varrem as linhas e colunas
for n=1:N
   for m=1:M
      %Laços para o armazenamento de valores no tempArray em uma vizinhança
      %local window x window
      tmpArrayIndex=1;
      for i=n-lowerWindow:n+upperWindow
         for j=m-lowerWindow:m+upperWindow
            %Teste para verificar se o pixel excede as bordas
            if(~(i<1 || j<1 || i>N || j>M))
               tempArray(tmpArrayIndex)=g(i,j);
               tmpArrayIndex=tmpArrayIndex+1;
            end
         end
      end

      %Ordenameno de tempArray com o uso de dois indexadores (i e j)
      i =1;
      j=2;
      while i<=tmpArrayIndex-2
          while j<=tmpArrayIndex-1
             if(tempArray(i)>tempArray(j))
                tmp=tempArray(i);
                tempArray(i)=tempArray(j);
                tempArray(j)=tmp;
             end
             j=j+1;
          end
          i=i+1;
          j=i+1;
      end

      %A mediana é o valor central da seqüência ordenada (seqüência com número
      %ímpar de elementos) ou é o maior valor do par de valores centrais
      %da seqüência ordenada (seqüência com número par de elementos)
      median(n,m)=tempArray(ceil((tmpArrayIndex-1)/2));

   end
end
**********************************************************************************************************
```

6.2.2. Implementação em Linguagem C

É apresentada a seguir a rotina **filtroMediana.c** que implementa a função de filtro da mediana em C. Essa função tem a assinatura similar ao do filtro da Média e é apresentada a seguir.

void filtroMediana(unsigned char **Array*, int *window*, int *rows*, int *columns*)

Programa 6: filtroMediana.c

```
*****************************************************************************************************************
#include <stdlib.h>

//Rotina do filtro da Mediana
void filtroMediana(unsigned char *inpArray,int window, int rows, int columns){

        //Definição de variáveis
        int i,j,n,m,tmp,tmp2,tmpArrayIndex,lowerWindow,upperWindow;
        int N = rows;
        int M = columns;
        unsigned char *tempArray;

        //Alocação Dinâmica de Memória
        tempArray = (unsigned char *)malloc((sizeof(unsigned char) * window * window));

        //Definição dos limites da janela
        lowerWindow=(int)(window-1)/2;
        upperWindow=(int)((window-1)/2.0+0.5);

        //Laços que varrem as linhas e colunas
        for (n=0;n<N;n++){
                for (m=0; m<M; m++){
                        //Laços para o armazenamento de valores no tempArray em uma
                        //vizinhança local window x window
                        tmp=n*columns+m;
                        tmpArrayIndex=0;
                        for(i=n-lowerWindow; i<=n+upperWindow;i++){
                                for(j=m-lowerWindow; j<=m+upperWindow;j++){
                                        //Teste para verificar se o pixel excede as bordas
                                        if(!(i<0 || j<0 || i>N-1 || j>M-1)){
                                                tmp2=i*columns + j;
                                                tempArray[tmpArrayIndex]=inpArray[tmp2];
                                                tmpArrayIndex++;
                                        }
                                }
                        }

                        //Ordenameno de tempArray com o uso de dois indexadores (i e j)
                        i=0;
                        j=1;
                        while(i<=tmpArrayIndex-2){
                                while(j<=tmpArrayIndex-1){
                                        if(tempArray[i]>tempArray[j]){
                                                tmp2=tempArray[i];
                                                tempArray[i]=tempArray[j];
```

```
                                        tempArray[j]=tmp2;
                                    }
                                    j++;
                                }
                                i++;
                                j=i+1;
                            }
                            //A mediana é o valor central da sequencia ordenada (seqüência com
                            // número ímpar de elementos) ou é o maior valor do par de valores
                            //centrais da seqüência ordenada (seqüência com número par de
                            //elementos)
                            inpArray[tmp]=tempArray[(tmpArrayIndex-1)/2];
                        }
                    }
}
```

****

6.3. Filtro de Wiener Adaptativo

O filtro digital de Wiener (algoritmo proposto por Lee), ao contrário dos outros filtros estudados anteriormente, é adaptativo, ou seja, o mesmo varia a filtragem da imagem se baseando na estatística local do sinal (média e variância) em uma vizinhança local N x M. Este filtro se baseia na minimização do erro quadrático entre a imagem de referência $f(n,m)$ e sua estimativa $\hat{f}(n,m)$, além de modelar o ruído como sendo Gaussiano branco, logo apresentando um excelente desempenho para tal ruído. O filtro de Wiener adaptativo pode ser representado pelas equações 10, 11 e 12 [3, 7].

$$\mu(n,m) = \frac{1}{NM} \sum_{i,j \in \eta} a(i,j) \qquad \textbf{(Equação 10)}$$

$$\sigma^2(n,m) = \left(\frac{1}{NM} \sum_{i,j \in \eta} a^2(i,j) \right) - \mu^2(n,m) \qquad \textbf{(Equação 11)}$$

$$\hat{f}(n,m) = \mu(n,m) + \frac{\max\left(0, \sigma^2(n,m) - v^2\right)}{\sigma^2(n,m)} \left(a(n,m) - \mu(n,m)\right)$$

(Equação 12)

Onde $\hat{f}(n,m)$ é a saída do filtro, *a(i,j)* é a entrada do filtro (imagem ruidosa), v^2 é variância do ruído, $\mu(n,m)$ e $\sigma^2(n,m)$ são a média e variância local, respectivamente, em η, isto é, em uma vizinhança local N x M ao redor do *pixel (n,m)*.

Quando não se conhece o ruído a priori, ν^2 pode ser calculado como a média de todas as variâncias locais $\sigma^2(n,m)$.

Para analisarmos os casos limítrofes do filtro de Wiener, podemos reescrever a equação 12, chegando à equação 13.

$$\hat{f}(n,m) = \left[1 - \frac{\max\left(0, \sigma^2(n,m) - \nu^2\right)}{\sigma^2(n,m)}\right]\mu(n,m) + \frac{\max\left(0, \sigma^2(n,m) - \nu^2\right)}{\sigma^2(n,m)}a(n,m)$$

(Equação 13)

Podemos verificar que:

Quando $\sigma^2(n,m) >> \nu^2$, $\frac{\max\left(0, \sigma^2(n,m) - \nu^2\right)}{\sigma^2(n,m)} \to 1$ e logo, $\hat{f}(n,m) \to a(n,m)$.

Quando $\sigma^2(n,m) \to \nu^2$, $\frac{\max\left(0, \sigma^2(n,m) - \nu^2\right)}{\sigma^2(n,m)} \to 0$ e logo, $\hat{f}(n,m) \to \mu(n,m)$.

Logo, notamos que quando há uma grande homogeneidade na vizinhança, isto é, quando a variância é pequena, o filtro realiza maior suavização da imagem, isto é, a saída tende à média. Já quando a variância é grande, o filtro realiza menor suavização, ou seja, a saída tende à imagem ruidosa.

6.3.1. Implementação em MatLab

É apresentada a seguir a rotina **filtroWiener.m** que implementa o filtro de Wiener em MatLab. Essa função tem a assinatura similar à dos filtros já apresentados:

Wiener **= filtroWiener(*G*, *window*);**

Por praticidade, estamos considerando que o ruído é estimado pelo programa.

Programa 7: filtroWiener.m

```
**********************************************************************************************************
function output=filtroWiener(g,window)
```

```
N = size(g,1);
M = size(g,2);

invsqwindow=single(1/(window*window));
mean=single(zeros(N,M));
var=single(zeros(N,M));
output=single(zeros(N,M));
noise=single(0);

%Definição dos limites da janela
lowerWindow=floor((window-1)/2);
upperWindow=ceil((window-1)/2);

%Laços que varrem as linhas e colunas
for n=1:N
   for m=1:M
    %Laços para o cálculo da média e variância locais
    %em uma vizinhança local window x window
      for i=n-lowerWindow:n+upperWindow
         for j=m-lowerWindow:m+upperWindow
            %Teste para verificar se o pixel excede as bordas
            if(~(i<1 || j<1 || i>N || j>M))
               mean(n,m)=mean(n,m)+g(i,j);
               var(n,m)=var(n,m)+g(i,j)*g(i,j);
            end
         end
      end
      mean(n,m)=mean(n,m)*invsqwindow;
      var(n,m)=var(n,m)*invsqwindow-mean(n,m)*mean(n,m);
      %Estimativa do ruído
      noise=noise+var(n,m);
   end
end

noise=noise/(N*M);

%Laço para o cálculo da expressão:
%output(n,m)= mean(n,m) + max(0, var(n,m - noise))/var * [g(n,m) - mean(n,m)];
for n=1:N
   for m=1:M
      output(n,m)=g(n,m)-mean(n,m);
      g(n,m)=var(n,m)-noise;
      if(g(n,m)<0)
         g(n,m)=0;
      end

      %Caso ocorra divisao por 0, a saída será igual à média
      if(var(n,m)==0)
         output(n,m)=mean(n,m);
      else
         output(n,m)=output(n,m)/var(n,m)*g(n,m)+mean(n,m);
      end
   end
end
```

****

6.3.2. Implementação em Linguagem C

É apresentada a seguir a rotina **filtroWiener.c** que implementa a função de filtro de Wiener em C. Essa função tem a assinatura similar à dos filtros já apresentados:

void filtroWiener(unsigned char **Array*, int *window*, int *rows*, int *columns*)

Programa 8: filtroWiener.c

```
*****************************************************************************************************
#include <stdlib.h>

//Rotina do filtro de Wiener
void filtroWiener(unsigned char *inpArray,int window, int rows, int columns){

        //Definição de variáveis
        int i,j,n,m,tmp,tmp2,lowerWindow,upperWindow;
        int N = rows;
        int M = columns;
        float noise=0;
        float invsqwindow;
        float *ptrTmpValue;
        float tmpValue=0.0;
        float *ptrOutput;
        float output=0.0;
        float *var, *mean;

        //Inicialização dos apontadores
        ptrOutput=&output;
        ptrTmpValue=&tmpValue;

        //Alocação Dinâmica de Memória
        var = (float*)malloc((sizeof(float) * rows * columns));
        mean = (float *)malloc((sizeof(float) * rows * columns));

        //Definição de parâmetro
        invsqwindow=(float)1/(float)(window*window);

        //Definição dos limites da janela
        lowerWindow=(int)(window-1)/2;
        upperWindow=(int)((window-1)/2.0+0.5);

        //Laços que varrem as linhas e colunas
        for (n=0;n<N;n++){
                for (m=0; m<M; m++){
                        //Laços para o cálculo da média e variância locais
                        //em uma vizinhança local window x window
                        tmp=n*columns+m;
                        mean[tmp]=0;
                        var[tmp]=0;
                        for(i=n-lowerWindow; i<=n+upperWindow;i++){
                                for(j=m-lowerWindow; j<=m+upperWindow;j++){
                                        //Teste para verificar se o pixel excede as bordas
                                        if(!(i<0 || j<0 || i>N-1 || j>M-1)){
                                                tmp2=i*columns + j;
                                                mean[tmp]=mean[tmp]+(float)inpArray[tmp2];
```

```
                                        var[tmp]=var[tmp]+(float)inpArray[tmp2] *
                                        (float)inpArray[tmp2];
                                }
                        }
                }
                mean[tmp]=mean[tmp] * invsqwindow;
                var[tmp]=var[tmp] * invsqwindow-mean[tmp]* mean[tmp];

                //Estimativa do ruído
                noise=noise+var[tmp];
        }
    }

    noise=(float)noise/(float)(N*M);

    //Laço para o cálculo da expressão:
    //output(n,m)= mean(n,m) + max(0, var(n,m) - noise))/var(n,m) * [a(n,m) - mean(n,m)];
    for(n=0; n<N; n++){
        for (m=0; m<M; m++){

            tmp=n*columns+m;

            *ptrOutput=(float)(inpArray[tmp]-mean[tmp]);
            *ptrTmpValue=(float)(var[tmp]-noise);
            if(tmpValue<0){
                tmpValue=0;
            }

            *ptrOutput=(float)(mean[tmp] + *ptrOutput/var[tmp] * (*ptrTmpValue));

            //Para arrendondar um número, adicionar 0.5
            //e realizar o typecast para int
            inpArray[tmp]=(unsigned char)(*ptrOutput+0.5);

        }
    }
}
```

7. Interface com o Usuário

Nesta seção apresentamos a interface do programa com o usuário. Por praticidade, desenvolvemos uma interface do tipo "texto", isto é, orientado a console para os programas desenvolvidos em MatLab e C.

7.1. Ambiente MatLab

O programa **filtro_main.m**, apresentado a seguir, realiza a interface com o usuário em ambiente MatLab, além de efetuar as rotinas de leitura e escrita de arquivos bitmap já discutidas na seção 5. Notemos que este programa deve ser executado a partir da pasta contendo os arquivos **filtroMedia.m, filtroMediana.m, filtroWiener.m** e o arquivo bitmap a ser filtrado. Finalizada a execução do programa, o arquivo de saída

com a imagem filtrada estará disponível na pasta corrente. Na figura 2, é apresentado um exemplo típico da interface com o programa.

```
***Filtros Digitais Bidimensionais***

Digite o nome arquivo de entrada (Ex. entrada.bmp): entrada.bmp

Digite o nome do arquivo de saída (Ex. saida.bmp): saida.bmp

Digite o tamanho da janela (inteiro): 3

Digite:
1- para filtragem da Média;
2- para filtragem da Mediana;
3- para filtragem de Wiener;

>> 3

Tempo para processamento da imagem: 0.671 segundos
```

Figura 2: Exemplo da interface com o programa filtro_main.m

Programa 9: filtro_main.m

```
%*************************************************************************************************************
clear
clc

%Interface com o usuário
disp('***Filtros Digitais Bidimensionais***');
inputFile = input('\nDigite o nome arquivo de entrada (Ex. entrada.bmp): ','s');
outputFile = input('\nDigite o nome do arquivo de saída (Ex. saida.bmp): ','s');
window = input('\nDigite o tamanho da janela (inteiro): ');

filterType=input('\nDigite:\n1- para filtragem da Média;\n2- para filtragem da Mediana;\n3- para
filtragem de Wiener;\n\n>> ');

if(filterType < 1 || filterType >3)
   disp('Opção inválida. O filtro de Wiener foi escolhido.');
        filterType = 3;
end

%Inicialização do contador
tic;

%Leitura do arquivo de imagem e armazenamento em ponto-flutuante de
%precisão simples
RGB = single(imread(inputFile));

%Armazenamento das componentes R, G e B em matrizes
R=RGB(:,:,1);
G=RGB(:,:,2);
```

```
B=RGB(:,:,3);

%Escolha do tipo de filtro
switch filterType

   case 1
      R=filtroMedia(R,window);
      G=filtroMedia(G,window);
      B=filtroMedia(B,window);

   case 2
      R=filtroMediana(R,window);
      G=filtroMediana(G,window);
      B=filtroMediana(B,window);

   case 3
      R = filtroWiener(R,window);
      G = filtroWiener(G,window);
      B = filtroWiener(B,window);

end

%Escrita das componentes R, G e B em uma matriz RGB
RGB(:,:,1)=R;
RGB(:,:,2)=G;
RGB(:,:,3)=B;

%Escrita da matriz RGB em um arquivo;
imwrite(uint8(RGB),outputFile,'bmp')

%Contagem e impessão do tempo de processamento
t=toc;
disp(' ');
disp(['Tempo para processamento de imagem: ', num2str(t)]);
*****************************************************************************************************
```

7.2 Linguagem C

O programa **filtro_main.c**, apresentado a seguir, realiza a interface com o usuário em linguagem C, além de efetuar as rotinas de leitura e escrita de arquivos bitmap já discutidas. A fim de gerar o programa executável **filtro.exe**, criamos um projeto no ambiente Visual C++ incluindo os arquivos **filtro_main.c**, **filtroMedia.m**, **filtroMediana.m** e **filtroWiener.m**. Notemos que este programa deve ser executado na mesma pasta do arquivo bitmap a ser filtrado. Finalizada a execução do programa, o arquivo de saída com a imagem filtrada estará disponível na pasta corrente. Na figura 3, é apresentado um exemplo típico da interface com o programa.

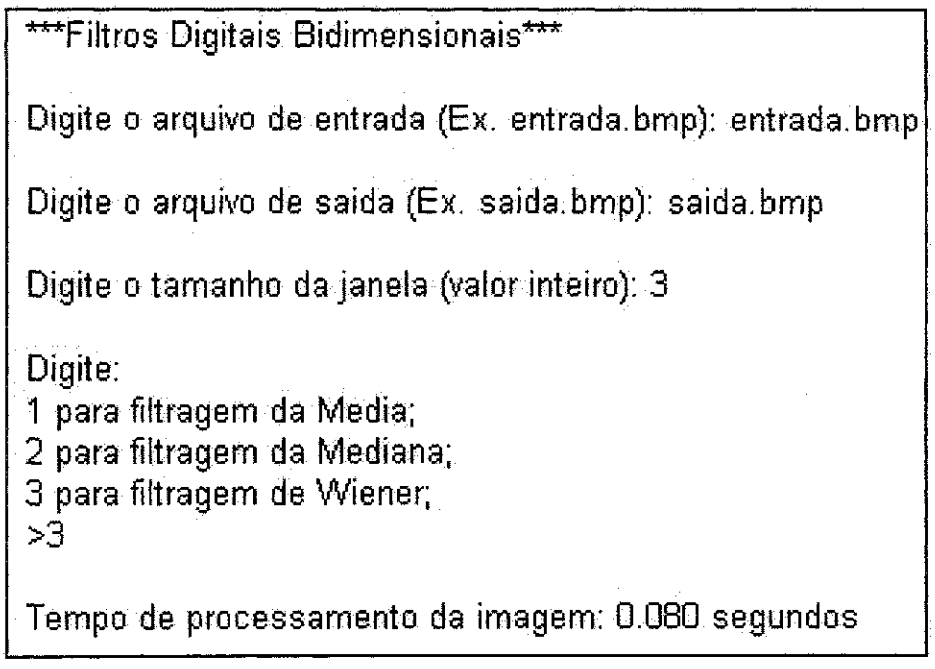

Figura 3: Exemplo da interface com o programa filtro.exe

Programa 10: filtro_main.c

```
*****************************************************************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

//Protótipos das funções
long getInfoImagem(FILE*, long, int);
void filtroMedia(unsigned char *, int, int, int );
void filtroMediana(unsigned char *, int, int, int );
void filtroWiener(unsigned char *, int, int, int );

void main()
{
        FILE *bmpInput, *bmpOutput;
        char arquivoEntrada[20];
        char arquivoSaida[20];
        int rows, columns, r, c, filterType;
        int window=0;
        long fileSize;
        double initClock, time;

        //Interface com o usuário
        printf("\n***Filtros Digitais Bidimensionais***\n");
        printf("\nDigite o arquivo de entrada (Ex. entrada.bmp): ");
        scanf("%s", arquivoEntrada);
        printf("\nDigite o arquivo de saida (Ex. saida.bmp): ");
        scanf("%s", arquivoSaida);
        printf("\nDigite o tamanho da janela (valor inteiro): ");
        scanf("%d", &window);

        //Escolha do tipo de filtro
        printf("\nDigite:\n");
        printf("1 para filtragem da Media;\n");
```

```
        printf("2 para filtragem da Mediana;\n");
        printf("3 para filtragem de Wiener;\n");
        printf(">");
        scanf ("%d", &filterType);

        //Caso a opção seja inválida, o filtro de Wiener é escolhido
        if(filterType < 1 || filterType > 3){
                printf("\nOpcao invalida. O filtro de Wiener foi escolhido.\n");
                filterType = 3;
        }

        //Inicia contagem do número de ticks de clock
        initClock=clock();

        //Abrindo arquivos de entrada e saída
        bmpInput = fopen(arquivoEntrada, "rb");
        bmpOutput = fopen(arquivoSaida, "wb");

        //Computando dados do arquivo BMP de entrada
        columns = (int)getInfoImagem(bmpInput, 18, 4);
        rows = (int)getInfoImagem(bmpInput, 22, 4);
        fileSize = getInfoImagem(bmpInput, 2, 4);

        //Testando se o tamanho do arquivo de entrada é o esperado
        if(fileSize!=(rows*columns*3+54)){
                printf("\nArquivo de entrada incorreto ou com dimensões desalinhadas...\n");
                exit(0);
        }

        //Copia cabeçalho do arquivo de entrada para o arquivo de saída
        {//novo escopo
        unsigned char buffChar[54];
        fseek(bmpInput, 0, SEEK_SET);
        fseek(bmpOutput, 0, SEEK_SET);

        fread(buffChar, sizeof(char), 54, bmpInput);
        fwrite(buffChar, sizeof(char), 54, bmpOutput);
        }//fim do novo escopo

        //Posicionando os apontadores bmpInput e bmpOutput para o setor de dados
        fseek(bmpInput, 54, SEEK_SET);
        fseek(bmpOutput, 54, SEEK_SET);

        {//novo escopo

                //Alocação Dinâmica de Memória para os arrays Red, Green, Blue e buffChar
                unsigned char *redValue =(unsigned char *) malloc((sizeof(unsigned
char)*rows*columns));
                unsigned char *greenValue = (unsigned char *)malloc((sizeof(unsigned
char)*rows*columns));
                unsigned char *blueValue = (unsigned char *)malloc((sizeof(unsigned char) *
rows * columns));
                unsigned char *buffChar = (unsigned char *)malloc((sizeof(unsigned char) *
rows * columns * 3));

                //Inicialização do indexador de buffChar
                long index = 0;

                //Leitura e armazenamento dos dados
                fread(buffChar, sizeof(unsigned char), rows*columns*3, bmpInput);
```

```
for(r=0; r<=rows - 1; r++){
        for(c=0; c<=columns - 1; c++){

                int tmp = r*columns + c;

                blueValue[tmp] = buffChar[index];
                index++;

                greenValue[tmp] = buffChar[index];
                index++;

                redValue[tmp] = buffChar[index];
                index++;
        }
}

//Aplicação do filtro escolhido
switch(filterType){

case 1:
        filtroMedia(redValue, window, rows, columns);
        filtroMedia(greenValue, window, rows, columns);
        filtroMedia(blueValue, window, rows, columns);
break;

case 2:
        filtroMediana(redValue, window, rows, columns);
        filtroMediana(greenValue, window, rows, columns);
        filtroMediana(blueValue, window, rows, columns);
break;

case 3:
        filtroWiener(redValue, window, rows, columns);
        filtroWiener(greenValue, window, rows, columns);
        filtroWiener(blueValue, window, rows, columns);
break;
}

//Reinicialização do indexador de buffChar
index=0;

//Escrita dos Arrays Red, Green e Blue ao arquivo de saída
for(r=0; r<=rows - 1; r++){
        for(c=0; c<=columns - 1; c++){

                int tmp = r*columns + c;
                buffChar[index]=blueValue[tmp];
                index++;
                buffChar[index]=greenValue[tmp];
                index++;
                buffChar[index]=redValue[tmp];
                index++;
        }
}
fwrite(buffChar, sizeof(unsigned char), rows*columns*3, bmpOutput);

}//fim do novo escopo
```

```
        //Fechamento os arquivos
        fclose(bmpInput);
        fclose(bmpOutput);

        //Cálculo e impressão do tempo gasto para a filtragem
        time=((double)clock()-(double)initClock)/(double)CLK_TCK;
        printf("\nTempo de processamento da imagem: %.3f segundos\n", time);

}

//Rotina de leitura de dados do cabeçalho da imagem
long getInfoImagem(FILE* inputFile, long offset, int numberOfChars)
{
        unsigned char *ptrC;
        long value = 0L;
        unsigned char tmpChar;
        int i;
        float power=1.0;

        ptrC = &tmpChar;

        fseek(inputFile, offset, SEEK_SET);

        //Cálculo do valor por adição de bytes
        for(i=0; i<numberOfChars; i++)  {
           fread(ptrC, sizeof(char), 1, inputFile);
                value = (long)(value + (*ptrC)*(power));
                power=power*256;
        }
        return(value);

}
```

****

8. Métricas de Desempenho

Nesta seção analisamos as métricas de desempenho referentes ao tempo de processamento dos filtros em ambiente MatLab e linguagem C. Faremos também um comentário acerca da qualidade das imagens filtradas, apresentando o MSE (*Mean Squared Error* - Erro Médio Quadrático) de cada componente de cor da imagem filtrada em relação à imagem de referência. O MSE é definido pela equação 14.

$$MSE = \frac{1}{N \cdot M} \sum_{i=1}^{N} \sum_{j=1}^{M} \left(\hat{f}(i,j) - f(i,j) \right)^2 \qquad \textbf{(Equação 14)}$$

Onde N e M referem-se às dimensões da imagem.

A fim de realizar as métricas de desempenho, utilizamos dois arquivos bitmap de 24 bits com resolução de 352 x 288 pixels (figuras 4 e 5). A figura 4 foi capturada com

uma *Webcam* Genius (sensor CMOS) modelo VideoCAM NB [8]. Notemos a presença de ruído. Já a figura 5 foi fotografada usando uma câmera digital Olympus (sensor de imagem CCD) referência u-miniD, Stylus V [9] e recortada para a resolução já citada. Notemos que a presença de ruído não é tão perceptível.

A figura 6 corresponde à imagem apresentada na figura 4 com uma adição de ruído Gaussiano Branco com média nula e variância igual a 0,01. Já a figura 7 corresponde à imagem apresentada na figura 5 com a adição do ruído tipo "*salt and pepper*", isto é, ruído impulsivo, com densidade (d) igual a 0,02. A imagem é afetada com o ruído em aproximadamente d*(N*M), onde N e M se referem às dimensões da imagem.

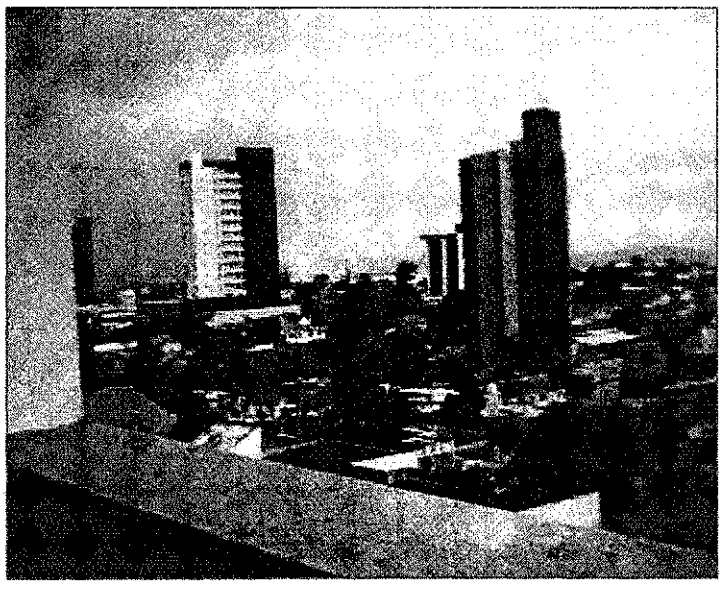

Figura 4: Imagem de referência captada com uma *Webcam* CMOS

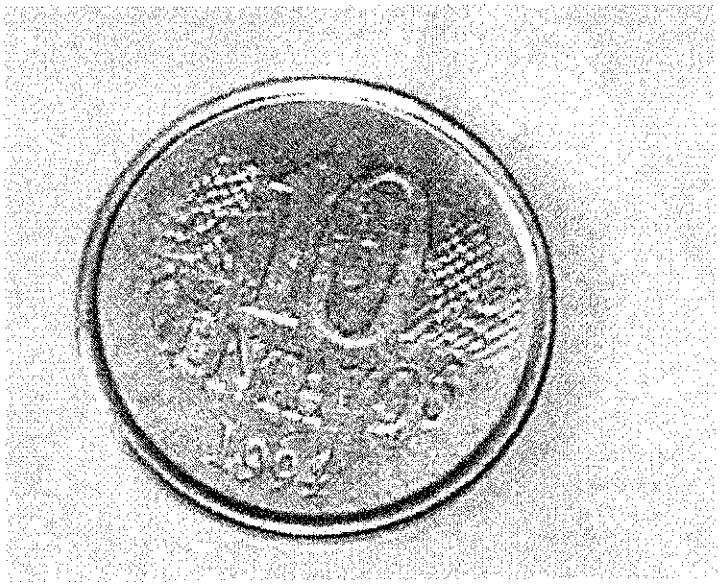

Figura 5: Imagem de referência captada com uma câmera CCD

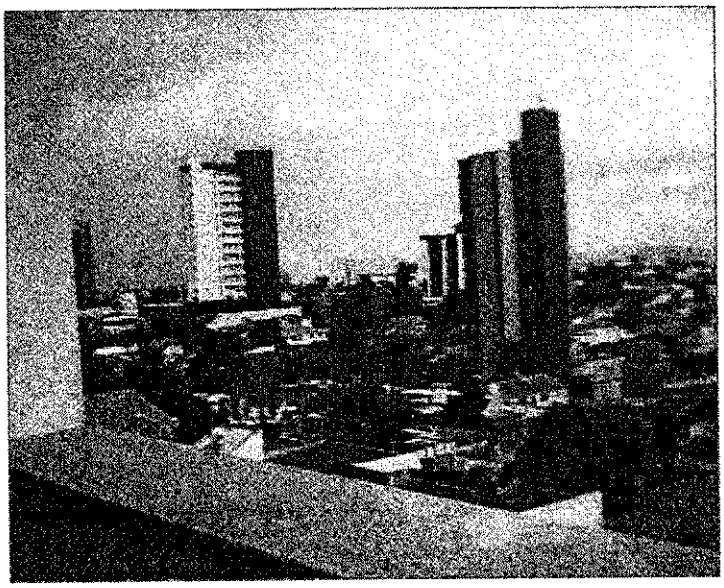

Figura 6: Ruído Gaussiano Branco adicionado à imagem apresentada na figura 4

Figura 7: Ruído "*salt and pepper*" adicionado à imagem apresentada na figura 5

O MSE de cada componente de cor das imagens ilustradas nas figuras 6 e 7, em relação às imagens originais (figuras 4 e 5), está apresentado na tabela 4. Observemos que neste caso estamos calculando o MSE das imagens ruidosas em relação às imagens originais, isto é, $MSE = \frac{1}{N \cdot M} \sum_{i=1}^{N} \sum_{j=1}^{M} (a(i,j) - f(i,j))^2$.

Tabela 2: Erro Médio Quadrático de cada componente de cor

MSE		
	Figura 6 (Ruído Gaussiano)	**Figura 7 (Ruído *salt and pepper*)**
Componente *Red*	552,51	466,90
Componente *Green*	619,61	443,60
Componente *Blue*	490,92	432,33

O tempo de processamento da rotina de leitura/escrita e de cada filtro foi calculado como sendo a média aritmética de 10 execuções usando a opção *profile* do compilador. Padronizamos a figura 4 como arquivo de entrada e o parâmetro referente ao tamanho da janela deslizante igual a três, ou seja, *window* = 3. Os tempos de processamento dos filtros excluem as rotinas de leitura/escrita de arquivo.

Os testes foram realizados em um *notebook* PC TOSHIBA [10] com processador Intel Pentium M de 1,86 GHz, 1 GB de RAM e rodando o sistema operacional Microsoft Windows XP Home Edition SP2 [11].

8.1. Rotinas de Leitura/Escrita

A série de tempos de processamento da rotina de leitura, armazenamento de dados e escrita de arquivos, implementados em MatLab e C, está apresentada nas tabelas 3 e 4, respectivamente.

Tabela 3: Tempos de processamento da rotina de Entrada/Saída em MatLab

	Iteração										
	1ª	**2ª**	**3ª**	**4ª**	**5ª**	**6ª**	**7ª**	**8ª**	**9ª**	**10ª**	**Média**
Tempo (s)	0,030	0,040	0,050	0,041	0,040	0,08	0,05	0,03	0,04	0,03	**0,043**

Tabela 4: Tempos de processamento da rotina de Entrada/Saída em linguagem C

	Iteração										
	1ª	**2ª**	**3ª**	**4ª**	**5ª**	**6ª**	**7ª**	**8ª**	**9ª**	**10ª**	**Média**
Tempo (ms)	3,638	3,480	3,551	3,577	3,459	3,709	3,383	3,419	3,605	3,380	**3,520**

8.2. Filtro da Média

As imagens resultantes do filtro da média estão ilustradas nas figuras 8 e 9.

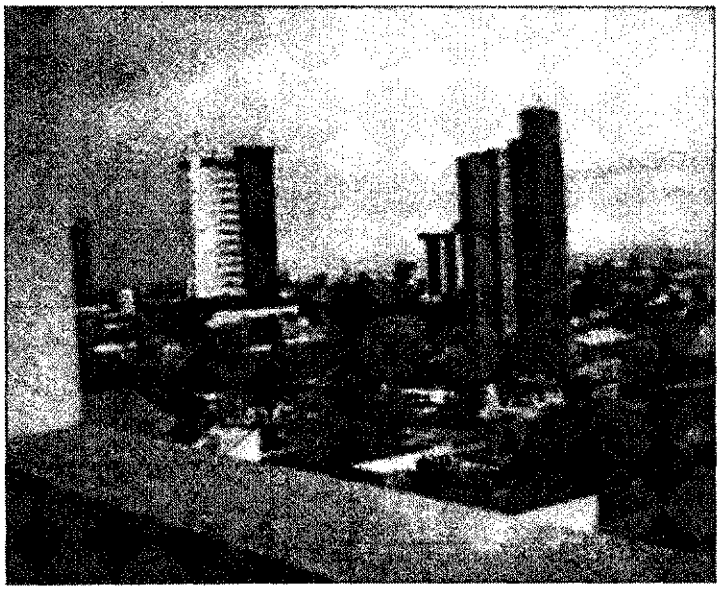

Figura 8: Filtro da média aplicado à imagem apresentada na figura 6

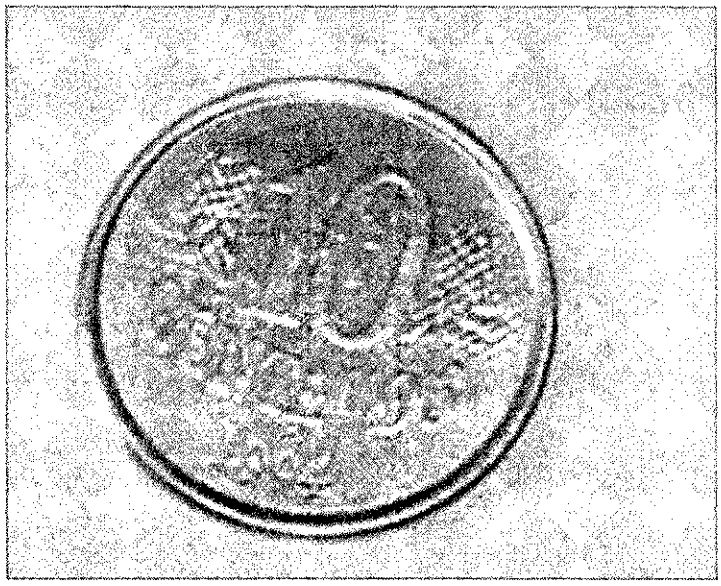

Figura 9: Filtro da média aplicado à imagem apresentada na figura 7

O MSE de cada componente de cor das imagens ilustradas nas figuras 8 e 9, em relação às imagens originais (figuras 4 e 5), está apresentado na tabela 5.

Tabela 5: Erro Médio Quadrático de cada componente de cor

MSE		
	Figura 8 (Ruído Gaussiano)	**Figura 9 (Ruído *salt and pepper*)**
Componente *Red*	303,38	221,16
Componente *Green*	243,83	217,37
Componente *Blue*	283,20	216,46

A partir das figuras 8 e 9, além das tabelas 2 e 5, podemos concluir que houve uma redução do ruído nas imagens, contudo houve uma suavização dos contornos.

A série de tempos de processamento deste filtro, implementados em MatLab e C, está apresentada nas tabelas 6 e 7, respectivamente.

Tabela 6: Tempos de processamento do filtro da média em MatLab

	Iteração										
	1ª	**2ª**	**3ª**	**4ª**	**5ª**	**6ª**	**7ª**	**8ª**	**9ª**	**10ª**	**Média**
Tempo (s)	0,541	0,591	0,541	0,591	0,581	0,541	0,551	0,540	0,540	0,541	**0,556**

Tabela 7: Tempos de processamento do filtro da média em linguagem C

	Iteração										
	1ª	**2ª**	**3ª**	**4ª**	**5ª**	**6ª**	**7ª**	**8ª**	**9ª**	**10ª**	**Média**
Tempo (ms)	33,61	30,01	29,84	29,74	29,57	33,82	30,00	29,72	35,54	29,24	**31,11**

A razão entre os tempos de processamento em MatLab e C é de 17,87. Isso representa um enorme ganho de desempenho em linguagem C.

O número de quadros por segundo que pode ser processado em linguagem C, excluindo o tempo médio da rotina de leitura/escrita, é de:

$$n = \frac{1}{31{,}11 \cdot 10^{-3}} = 32{,}14$$

8.3. Filtro da Mediana

As imagens resultantes do filtro da mediana estão ilustradas nas figuras 10 e 11.

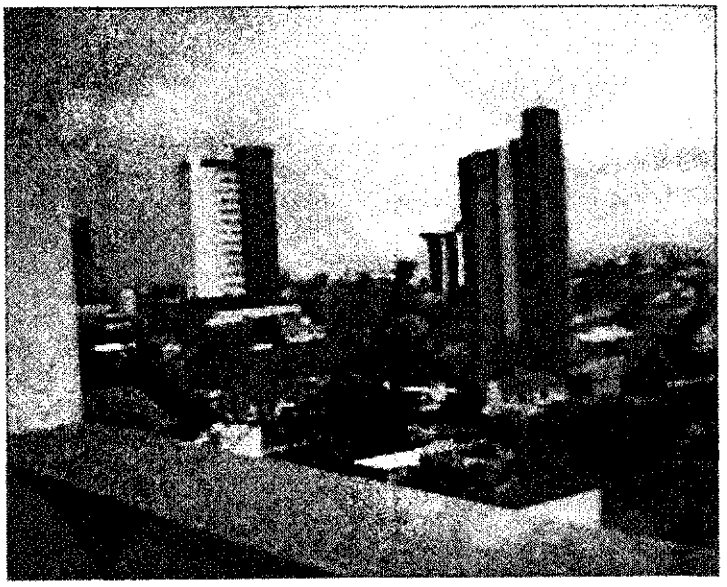

Figura 10: Filtro da mediana aplicado à imagem apresentada na figura 6

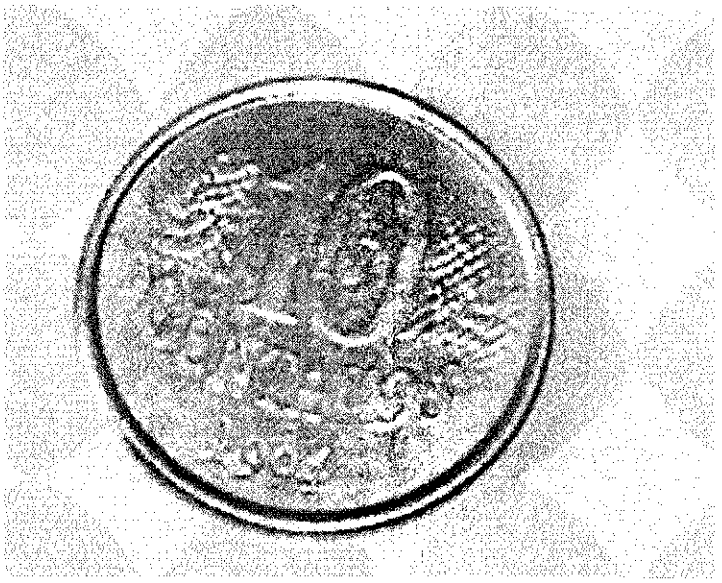

Figura 11: Filtro da mediana aplicado à imagem apresentada na figura 7

O MSE de cada componente de cor das imagens ilustradas nas figuras 10 e 11, em relação às imagens originais (figuras 4 e 5), está apresentado na tabela 8.

Tabela 8: Erro Médio Quadrático de cada componente de cor

MSE		
	Figura 10 (Ruído Gaussiano)	**Figura 11 (Ruído *salt and pepper*)**
Componente *Red*	283,44	77,13
Componente *Green*	255,20	76,92
Componente *Blue*	250,40	76,86

Na figura 11, é bastante evidente a redução do ruído "*salt and pepper*". Também notamos uma redução do ruído na figura 10. Em relação ao filtro da média, podemos notar uma maior nitidez das bordas. A comparação das tabelas 2 e 8 confirma a redução do ruído.

A série de tempos de processamento para este filtro está apresentada nas tabelas 9 e 10, respectivamente.

Tabela 9: Tempos de processamento do filtro da mediana em MatLab

	Iteração										
	1ª	**2ª**	**3ª**	**4ª**	**5ª**	**6ª**	**7ª**	**8ª**	**9ª**	**10ª**	**Média**
Tempo (s)	0,911	0,902	0,901	0,911	0,912	0,902	0,901	0,891	0,901	0,912	**0,904**

Tabela 10: Tempos de processamento do filtro da mediana em linguagem C

	Iteração										
	1ª	**2ª**	**3ª**	**4ª**	**5ª**	**6ª**	**7ª**	**8ª**	**9ª**	**10ª**	**Média**
Tempo (ms)	91,40	90,20	90,51	92,31	90,19	92,71	92,36	90,61	94,49	92,64	**91,74**

A razão entre os tempos de processamento em MatLab e C é de 9,85.

O número de quadros por segundo que pode ser processado em linguagem C, excluindo o tempo médio da rotina de leitura/escrita, é de:

$$n = \frac{1}{91{,}74 \cdot 10^{-3}} = 10{,}90 .$$

8.4. Filtro de Wiener

As imagens resultantes do filtro de Wiener estão ilustradas nas figuras 12 e13.

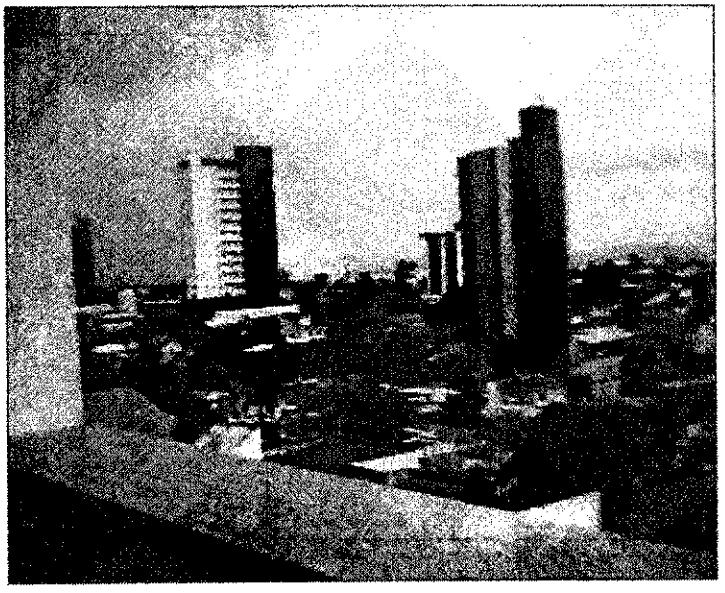

Figura 12: Filtro de Wiener aplicado à imagem apresentada na figura 6

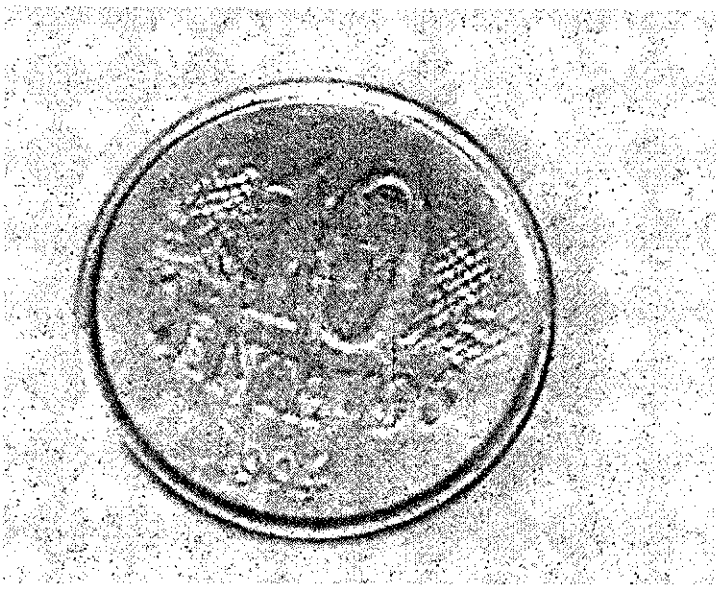

Figura 13: Filtro de Wiener aplicado à imagem apresentada na figura 6

O MSE de cada componente de cor das imagens ilustradas nas figuras 12 e 13, em relação às imagens originais (figuras 4 e 5), está apresentado na tabela 11.

Tabela 11: Erro Médio Quadrático de cada componente de cor

MSE		
	Figura 12 (Ruído Gaussiano)	**Figura 13 (Ruído *salt and pepper*)**
Componente *Red*	183,24	372,72
Componente *Green*	157,74	356,38
Componente *Blue*	176,92	348,34

Na figura 12, além de haver uma expressiva redução do ruído, notamos uma maior nitidez nas bordas, o qual é uma excelente característica do filtro de Wiener. Entretanto, na figura 13, não notamos grande melhoria na imagem. Notemos que o fato do ruído "*salt and pepper*" ser impulsivo implica em variâncias locais elevadas nas vizinhanças contendo o ruído, logo, o filtro de Wiener tende a preservar a imagem original, como discutido na seção 6.3. A comparação das tabelas 2 e 11 confirma a redução do ruído.

A série de tempos de processamento para este filtro está apresentada nas tabelas 12 e 13, respectivamente.

Tabela 12: Tempos de processamento do filtro de Wiener em MatLab

	Iteração										
	1ª	**2ª**	**3ª**	**4ª**	**5ª**	**6ª**	**7ª**	**8ª**	**9ª**	**10ª**	**Média**
Tempo (s)	0,661	0,661	0,681	0,691	0,671	0,671	0,661	0,671	0,671	0,681	**0,672**

Tabela 13: Tempos de processamento do filtro da de Wiener em linguagem C

	Iteração										
	1ª	**2ª**	**3ª**	**4ª**	**5ª**	**6ª**	**7ª**	**8ª**	**9ª**	**10ª**	**Média**
Tempo (ms)	58,60	61,13	58,13	58,03	59,84	57,61	57,52	60,07	57,97	58,10	**58,70**

A razão entre os tempos de processamento em MatLab e C é de 11,45.

O número de quadros por segundo que pode ser processado em linguagem C, excluindo o tempo médio da rotina de leitura/escrita, é de:

$$n = \frac{1}{58{,}70 \cdot 10^{-3}} = 17{,}04 .$$

8.5. Resumo

Na tabela 14, apresentamos um breve resumo dos tempos médios de processamento de imagem dos três filtros em ambiente MatLab e linguagem C. Já nas figuras 14 e 15, apresentamos as figuras 6 e 7 submetidas a todos os filtros discutidos.

Tabela 14: Resumo dos tempos médios de processamento de imagem

	Filtro da Média	**Filtro da Mediana**	**Filtro de Wiener**
Ambiente MatLab	0,556 s	0,904 s	0,672 s
Ambiente C	0,0311 s	0,0917 s	0,0587 s

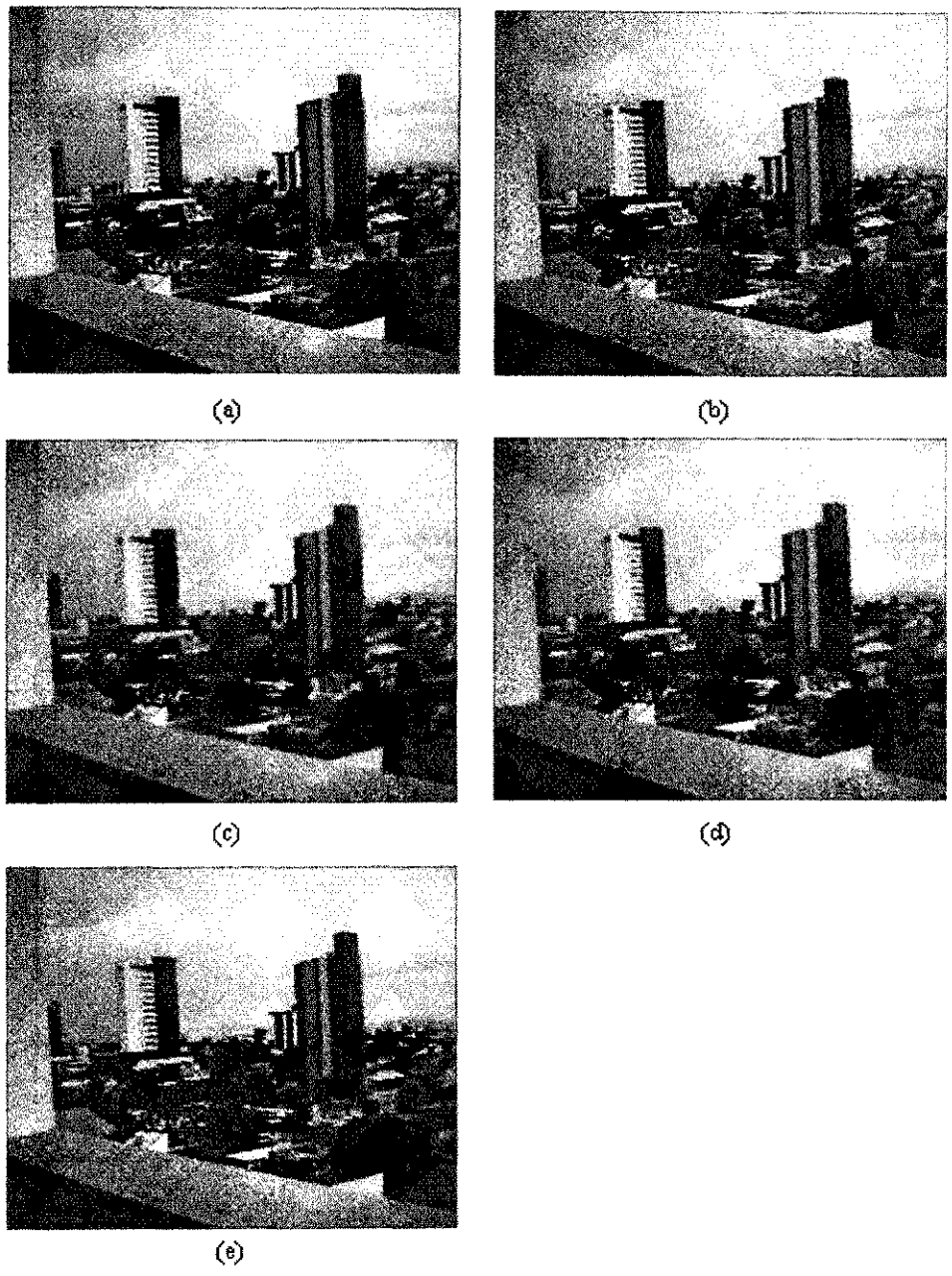

Figura 14: (a) Imagem ilustrada na figura 4 (de referência); (b) Imagem ilustrada na figura 6 (com ruído gaussiano branco); (c) Filtragem da média; (d) Filtragem da mediana; (e) Filtragem de Wiener adaptativo

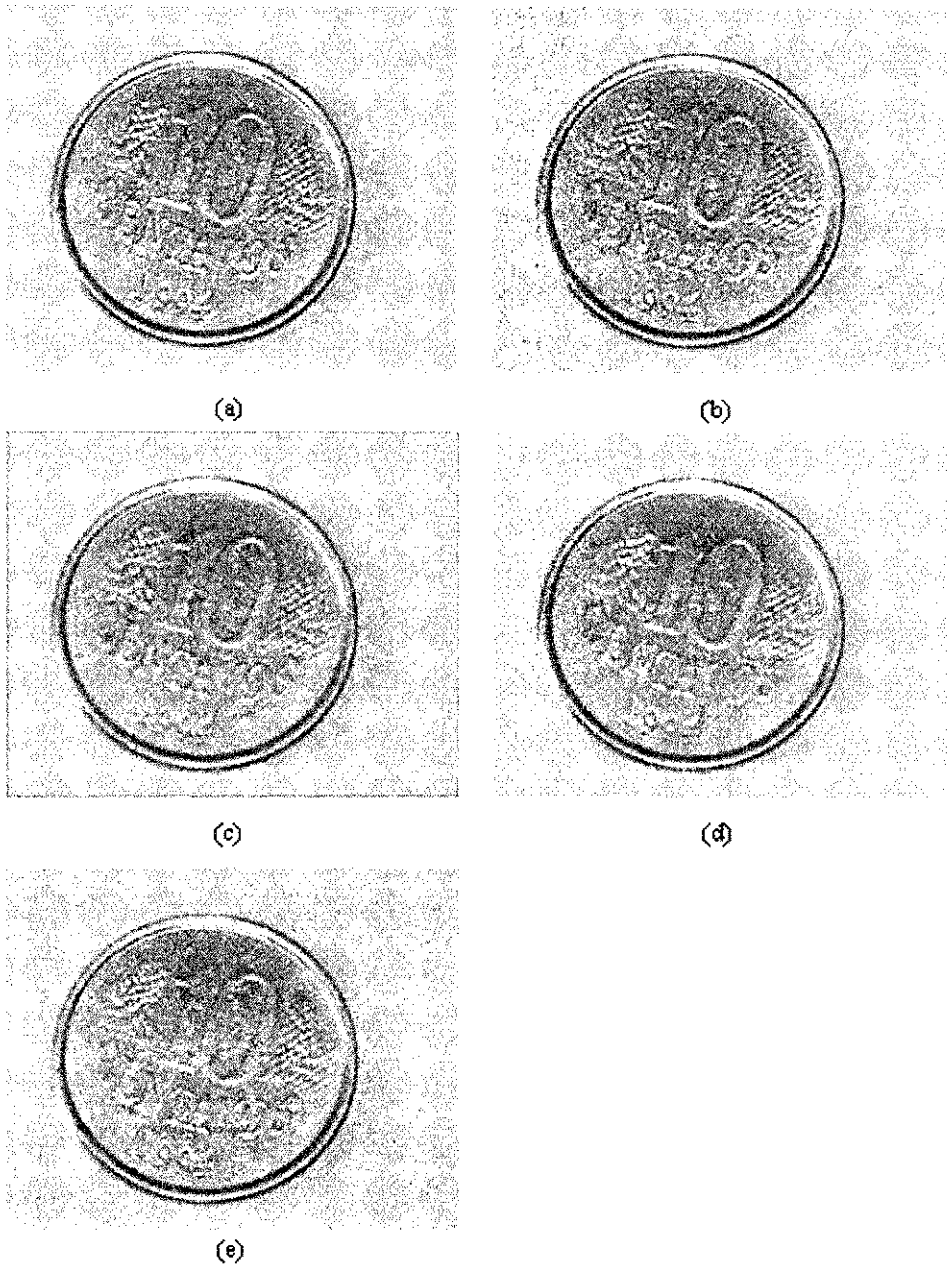

Figura 15: (a) Imagem ilustrada na figura 5 (de referência); (b) Imagem ilustrada na figura 7 (com ruído *salt & pepper*); (c) Filtragem da média; (d) Filtragem da mediana; (e) Filtragem de Wiener adaptativo

Na tabela 15, apresentamos um resumo dos MSEs das imagens ilustradas nas figuras 4 e 5 em relação às imagens ruidosas e em relação às imagens submetidas a todos os filtros discutidos.

Tabela 15: Erro Médio Quadrático de cada componente de cor em relação às imagens originais

MSE			
		Ruído Gaussiano	**Ruído *salt and pepper***
Imagens Ruidosas	**Componente *Red***	552,51	466,90
	Componente *Green*	619,61	443,60
	Componente *Blue*	490,92	432,33
Filtro da Média	**Componente *Red***	303,38	221,16
	Componente *Green*	243,83	217,37
	Componente *Blue*	283,20	216,46
Filtro da Mediana	**Componente *Red***	283,44	77,13
	Componente *Green*	255,20	76,92
	Componente *Blue*	250,40	76,86
Filtro da Wiener	**Componente *Red***	183,24	372,72
	Componente *Green*	157,74	356,38
	Componente *Blue*	176,92	348,34

9. Conclusões

Implementamos com sucesso, em ambiente MatLab e linguagem C, os três algoritmos de redução de ruído propostos, a citar, os filtros digitais bidimensionais da média, mediana e de Wiener adaptativo.

Verificamos que, sendo o MatLab uma linguagem de alto nível, a implementação e verificação algorítmica tornou-se mais fácil. Por exemplo, detalhes na implementação de rotinas para a manipulação de dados em arquivos é altamente simplificada e não exige a manipulação em nível de bytes. Já a implementação em C exige conhecimento detalhado acerca do padrão de imagem bitmap de 24-bits e a manipulação de dados em nível de bytes.

Nas métricas de desempenho realizadas, constatamos que a implementação em linguagem C é muito mais eficiente do que em ambiente MatLab, o que era esperado haja vista que o MatLab é uma linguagem interpretada, ao contrário de C, que é uma linguagem compilada. Dentre os filtros avaliados, constatamos que o da média é o mais rápido e o da mediana é o mais lento. Já o filtro de Wiener é um pouco mais lento que o filtro da média.

Em se tratando do ruído Gaussiano branco, típico em sensores de imagem CMOS e CCD, o filtro de Wiener apresenta os melhores resultados qualitativos, pois o ruído é atenuado, porém conservando mais as bordas e detalhes da imagem, isto é, as componentes de alta freqüência do sinal original. Esse fato é confirmado pelos dados da tabela 15, em que o MSE das componentes de cor é a menor dos três filtros considerados. Por outro lado, o filtro da média apresenta os piores resultados, pois a diminuição do ruído é acompanhada de muita suavização e perda da definição das bordas. No filtro da mediana, o efeito de suavização e perda de definição das bordas é um pouco menor que no filtro da média, porém continua perceptível.

Já considerando a imagem com ruído impulsivo (*salt and pepper*), o filtro da mediana apresenta os melhores resultados qualitativos, pois o ruído é completamente removido e ocorre pouca suavização da imagem. Esse fato é confirmado pelos dados da

tabela 15, em que o MSE das componentes de cor é a menor dos três filtros considerados O filtro da média atenua o ruído, mas o mesmo fica notório, além de ocorrer suavização na imagem filtrada. O filtro de Wiener é pouco eficiente para este tipo de ruído, haja vista que o ruído é impulsivo, isto é, de alta freqüência, logo o filtro tende a preservar a imagem ruidosa original.

Um resumo dos principais resultados é apresentado na tabela 16.

Tabela 16: Resumo dos principais resultados

		Filtro da Média	**Filtro da Mediana**	**Filtro de Wiener**
Tempo de Processamento	**Ambiente MatLab**	0,556 s	0,904 s	0,672 s
	Ambiente C	0,0311 s	0,0917 s	0,0587 s
	Rapidez	O mais rápido	O mais lento	Intermediária
Filtragem do Ruído	**Gaussiano Branco**	O menos indicado; Muita suavização e perda da definição das bordas imagem.	Suavização da imagem e perda da definição das bordas (menor que no filtro da média).	O mais indicado; Menor suavização da imagem e maior preservação das bordas.
	Salt and Pepper	O ruído impulsivo é pouco removido e ocorre suavização da imagem.	O mais indicado; Há pouca suavização e o ruído impulsivo é completamente removido.	O menos indicado; O ruído impulsivo é pouco removido;

Ressaltamos que esta etapa serve de base para a segunda etapa do projeto (referida na Introdução deste trabalho), na qual os algoritmos serão implementados em ponto fixo e lançados na plataforma final (DSP).

Indubitavelmente, o estudo realizado proporcionou uma excelente oportunidade a fim de aprofundar e consolidar os conhecimentos na área de Processamento Digital de Sinais.

10. Referências Bibliográficas

[1] PRATT, William K., 2001. *Digital Image Processing: PIKS Inside*. 3rd Edition. John Willey & Sons.

[2] KUO, Sen M. e LEE, Bob H. *Real-Time Digital Signal Processing: Implementations, Applications and Experiments with the TMS320C55X*. John Wiley & Sons, LTD., 2001.

[3] MATHWORKS, The, 2006. *MATLAB: Image Processing Toolbox User's Guide*. The MathWorks, Inc.

[4] CENTER, Pittsburgh Supercomputing. *The IEEE Standard for Floating Point Arithmetic*. Site: http://www.psc.edu/general/software/packages/ieee/ieee.html (outubro de 2005)

[5] SMITH III, Julius O., 2004. *Introduction to Digital Filters with Audio Applications*. Department of Music, Stanford University. Site: http://ccrma.stanford.edu/~jos/filters/Definition_Filter.html (outubro de 2005)

[6] GREEN, Bill, 2002. *24-Bit BMP Raster Data Tutorial & Grayscaling*. Site: http://www.pages.drexel.edu/%7Eweg22/colorBMP.html (outubro de 2005)

[7] CHAN, Phil. *One-Dimensional Processing for Adaptive Image Restoration*. Research Laboratory of Electronics, Massachusetts Institute of Technology, 1984.

[8] GENIUS, 2004. *KYE Systems Corp.*

[9] OLYMPUS, 2005. *Olympus Imaging Corporation.*

[10] TOSHIBA, 2005. *Toshiba America Information Systems, Inc.*

[11] MICROSOFT, 2005. *Microsoft Windows XP SP2*. Microsoft Corporation.

[12] BAR, Leah et al. *Image Deblurring in the Presence of Salt-and-Pepper Noise*. Tel Aviv University.

[13] NETWORK, The C++ Resources, 2005. *C++ Reference*. Site: http://www.cplusplus.com/ref/ (outubro de 2005)