Functions.h

Este documento apresenta a descrição das funções frequentemente utilizadas no processamento digital de sinal

|  |  |
| --- | --- |
| **função** | **descrição** |
| [**Filtragem 500Hz, 50hz, 5hz**](#_Filtragem_500Hz,_50hz,) | Filtragem de um sinal à frequencia de amostragem de 10kHz |
| [**Transformada ABC (AB) ( alfa/beta**](#_Transformada_ABC_(AB)) | Transformada de um sinal trifásico (simples ou composto) nas componentes alfa e beta |
| [**Atraso de um sinal**](#_Atraso_de_um) | Atraso de um sinal |
| [**Cálculo do True RMS de um sinal**](#_Cálculo_do_True) |  |

# Filtragem 500Hz, 50hz, 5hz

Filtro de primeira ordem, com os coeficientes para a frequência de amostragem de 10kHz

### Cabeçalhos:

**#define** NUM\_FILTERS 15 //Number of filters used

////// FUNCTION PROTOTYPES

**void** **init\_filters**(**void**);

**float** **filter\_500hz**(**float** input\_value, **int** filter\_number);

**float** **filter\_50hz**(**float** input\_value, **int** filter\_number);

**float** **filter\_5hz**(**float** input\_value, **int** filter\_number);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\* Filters' variables \*\*\*\*\*\*\*\*\*\*\*\*/

**float** value\_ant1[NUM\_FILTERS];

**float** value\_filt\_ant1[NUM\_FILTERS];

**float** value\_filt[NUM\_FILTERS];

**void** **init\_filters**(**void**) {

**int** aux\_i = 0;

**for** (aux\_i = 0; aux\_i < NUM\_FILTERS; aux\_i++) {

value\_ant1[aux\_i] = 0;

value\_filt\_ant1[aux\_i] = 0;

value\_filt[aux\_i] = 0;

}

}

### Código das funções

**float** **filter\_5hz**(**float** input\_value, **int** filter\_number) {

value\_filt[filter\_number] = ((0.001568334083281 \* input\_value + 0.001568334083281 \* value\_ant1[filter\_number] + 0.996863331833438 \* value\_filt\_ant1[filter\_number]));

value\_ant1[filter\_number] = input\_value;

value\_filt\_ant1[filter\_number] = value\_filt[filter\_number];

**return** value\_filt[filter\_number];

}

**float** **filter\_50hz**(**float** input\_value, **int** filter\_number) {

value\_filt[filter\_number] = ((0.015466291403103 \* input\_value + 0.015466291403103 \* value\_ant1[filter\_number] + 0.969067417193793 \* value\_filt\_ant1[filter\_number]));

value\_ant1[filter\_number] = input\_value;

value\_filt\_ant1[filter\_number] = value\_filt[filter\_number];

**return** value\_filt[filter\_number];

}

**float** **filter\_500hz**(**float** input\_value, **int** filter\_number) {

value\_filt[filter\_number] = ((0.136728735997320 \* input\_value + 0.136728735997320 \* value\_ant1[filter\_number] + 0.726542528005361 \* value\_filt\_ant1[filter\_number]));

value\_ant1[filter\_number] = input\_value;

value\_filt\_ant1[filter\_number] = value\_filt[filter\_number];

**return** value\_filt[filter\_number];

}

### Exemplo de utilização:

VA = filter\_500hz(VA, 0);

VB = filter\_500hz(VB, 1);

VC = filter\_500hz(VC, 2);

# Transformada ABC (AB) 🡪 alfa/beta

Funções para a transformadas de componentes simples ou compostas para alfa/beta

### Cabeçalhos:

**#define** K23 0.66666666666666666666666666666667f

**#define** Ksqr32 0.86602540378443864676372317075294f

**#define** K23sqr3 0.38490017945975050967276585366797f

**float** V\_abc[3]={0.0f, 0.0f, 0.0f}; // AC voltages

**float** I\_abc[3]={0.0f, 0.0f, 0.0f}; // AC currents

**float** V\_AB[2]={0.0f, 0.0f}; // alpha beta voltage

**float** I\_AB[2]={0.0f, 0.0f}; // alpha beta current

**void** **abc\_AB**(**float** \*abc, **float** \*AB);

**void** **abc\_AB\_Dy**(**float** \*abc, **float** \*AB);

### Código das funções

**void** **abc\_AB**(**float** \*abc, **float** \*AB){

// Conventional conversion into orthogonal components

\*AB=K23\*(\*abc-0.5f\*\*(abc+1)-0.5f\*\*(abc+2));

\*(AB+1)=K23\*(Ksqr32\*\*(abc+1)-Ksqr32\*\*(abc+2));

}

**void** **abc\_AB\_Dy**(**float** \*abc, **float** \*AB){

// Conversion into orthogonal components converting triangle to star

\*AB=K23sqr3\*(Ksqr32\*\*abc-Ksqr32\*\*(abc+2));

\*(AB+1)=K23sqr3\*(-0.5f\*\*abc+\*(abc+1)-0.5f\*\*(abc+2));

}

### Exemplo de utilização:

I\_abc[0] = IAaux;

I\_abc[1] = IBaux;

I\_abc[2] = ICaux;

abc\_AB(I\_abc, I\_AB);

V\_abc[0] = VCaux; //VAB --> corrigida sequência de fases

V\_abc[1] = VAaux; //VBC --> corrigida sequência de fases

V\_abc[2] = VBaux; //VCA --> corrigida sequência de fases

abc\_AB\_Dy(V\_abc, V\_AB);

i\_alfa = I\_AB[0];

i\_beta = I\_AB[1];

v\_alfa = -V\_AB[0]; //garantir sequência de fases positiva; garantir que

//v\_alfa se encontra em fase com i\_alfa

v\_beta = V\_AB[1];

# Atraso de um sinal

Função para realizar o atraso de um sinal

### Cabeçalhos:

**#define** SIZE\_VECTOR 200 // número de samples a atrasar (200\*100u = 20ms)

**double** signal\_input, signal\_output;

int16\_t position = 0;

**double** delayed\_v\_alfa[SIZE\_VECTOR], delayed\_i\_alfa[SIZE\_VECTOR];

**double** v\_alfa\_delayed, i\_alfa\_delayed;

### Código das funções

### Exemplo de utilização:

// geração da posição para o array circular

position = (position>(SIZE\_VECTOR-2)) ? 0 : position + 1;

// Atraso de v\_alfa

signal\_input = v\_alfa;

signal\_output = delayed\_v\_alfa[position];

delayed\_v\_alfa[position] = signal\_input;

v\_alfa\_delayed = signal\_output;

// Atraso de i\_alfa

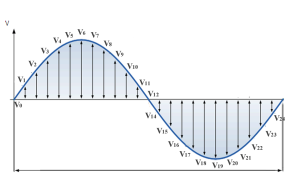
signal\_input = i\_alfa;

signal\_output = delayed\_i\_alfa[position];

delayed\_i\_alfa[position] = signal\_input;

i\_alfa\_delayed = signal\_output;

# Cálculo do True RMS de um sinal

Função para realizar o cálculo do verdadeiro valor eficaz. Esta função utiliza a obtenção do atraso do sinal, atraso de um ciclo de rede, e realiza o cálculo da seguinte forma:

### Cabeçalhos:

**#define** SIZE\_VECTOR 200 // número de samples a atrasar (200\*100u = 20ms)

**double** signal\_input, signal\_output;

int16\_t position = 0;

**double** delayed\_v\_alfa[SIZE\_VECTOR], delayed\_i\_alfa[SIZE\_VECTOR];

**double** v\_alfa\_delayed, i\_alfa\_delayed;

**double** integral\_v\_alfa = 0.0f, integral\_i\_alfa = 0.0f;

**float** v\_rms, i\_rms;

### Código das funções

### Exemplo de utilização:

/\*\*\*\* Atraso de 20ms (200 amostras) \*\*\*\*\*\*/

position = (position>(SIZE\_VECTOR-2)) ? 0 : position + 1;

// Atraso de v\_alfa

signal\_input = v\_alfa;

signal\_output = delayed\_v\_alfa[position];

delayed\_v\_alfa[position] = signal\_input;

v\_alfa\_delayed = signal\_output;

// Atraso de i\_alfa

signal\_input = i\_alfa;

signal\_output = delayed\_i\_alfa[position];

delayed\_i\_alfa[position] = signal\_input;

i\_alfa\_delayed = signal\_output;

// Cálculo True RMS

integral\_v\_alfa = integral\_v\_alfa + (v\_alfa\*v\_alfa - v\_alfa\_delayed\*v\_alfa\_delayed);

integral\_i\_alfa = integral\_i\_alfa + (i\_alfa\*i\_alfa - i\_alfa\_delayed\*i\_alfa\_delayed);

v\_rms = **sqrt**(integral\_v\_alfa \* UM\_SIZE\_VEC);

i\_rms = **sqrt**(integral\_i\_alfa \* UM\_SIZE\_VEC);

v\_rms = filter\_5hz(v\_rms, 12);

i\_rms = filter\_5hz(i\_rms, 13);