MRB MATH LIBRARY

DOCUMENTATION

This library is intended for use in lower-cost, slower microcontrollers not dedicated for mathematical calculations (e.g. ATmega/Arduino, ARM Cortex M0 and M4, ESP, Raspberry Pi pico) to speed up floating point operations. It can also be used in DSP to reduce cycle time, wherever cycle time is very critical, but the accuracy of calculations is not so critical. The MRB_MATH library will also significantly speed up the execution time of math calculations on microcontrollers without FPU or without hardware support for the default math.h library. An additional function not included in the math.h library is the function for calculating the RMS and discrete Fourier transformation (DFT) of a signal.

In summary, this library allows faster calculations at the expense of lower precision and higher memory usage.

The documentation describes the functions contained in the MRB_MATH library and compares them with their equivalents from the math.h library. Execution times and precision are compared. The results are summarized in graphs or tables. The possibility to parameterize each function is also described - the MRB_MATH library makes it possible to increase the precision of a function at the cost of additional cycles or FLASH/RAM memory usage.

Functions included in the library

- ***sin_f****(float x)* — *sinus from argument x*
- ***cos_f****(float x)* — *cosinus from argument x*
- ***fast_invsqrt****(float x)* — *fast inverse root square from argument x*
- ***fast_sqrt****(float x)* — *fast root square from argument x*
- ***RMS****(float x)* — *root mean square function (with normal sqrt function)*
- ***fast_RMS****(float x)* — *fast root mean square function (with fast sqrt function)*
- ***rapid_RMS****(float x)* — *rapid root mean square fun. (approach without sqrt)*
- ***DFT****(float x)* — *discrete Fourier transform*

Execution speed overview

Table 1 Comparison of function execution time in terms of cycles on STM32 microcontroller with Cortex M-7 core

Function / Library	**<math.h>**	**“MRB_MATH_LIB.h”**
sinus	762 cycles	30 cycles
cosinus	762 cycles	32 cycles
sqrt	674 cycles	28 cycles
RMS	-	694 cycles
fast RMS	-	46 cycles
rapid RMS	-	13 cycles
DFT	-	25k cycles

Library created by Maciej Brzycki ------ February 2024

Trigonometric functions

Basic sine and cosine functions in the MRB_MATH library are based on the look up table, which will be stored in the microcontroller's flash memory when the program is uploaded. While the program is running, it permanently occupies space in RAM memory. User can define the size of memory usage with *LUTSIZE* parameter. When equal to 1 - look up table will take about 4kB of memory (1000 float values), when 2 – 8kB and when 4 – 16kB.

Important note: the range of the function's arguments is not as large as that of the **sin** function from the math.h library. The argument of the **sin_f** function **should be** in range from the range -2π to 4π. For wider range of arguments **sin_f** and **cos_f** will work, but computing time will be longer.

```
// ***********SIN and COS*********************************//
//You can define lenght of look up table (in kB). The bigger the look up table, the more accurate sin_f and cos_f function are
#define LUTSIZE 1 //4*kB
```

Figure 1 LUT size choice

```
//******************************
#if LUTSIZE == 2 Inactive Preprocessor Block
#else
#define LUT_LENGTH 1000
const float MRB_TL_SINUS_LOOKUP[LUT_LENGTH] =
{
    0.000000, 0.001572, 0.003145, 0.004717, 0.006289, 0.007862, 0.009434, 0.011006, 0.012579, 0.014151,
    0.015723, 0.017295, 0.018867, 0.020439, 0.022011, 0.023583, 0.025155, 0.026727, 0.028299, 0.029871,
```

Figure 2 LUT view - beginning of the declaration

```
float sin_mine = sin_f(x); // Result from sin_f
```

```
float cos_mine = cos_f(x); // Result from cos_f
```

Figure 3 Use of trigonometric functions

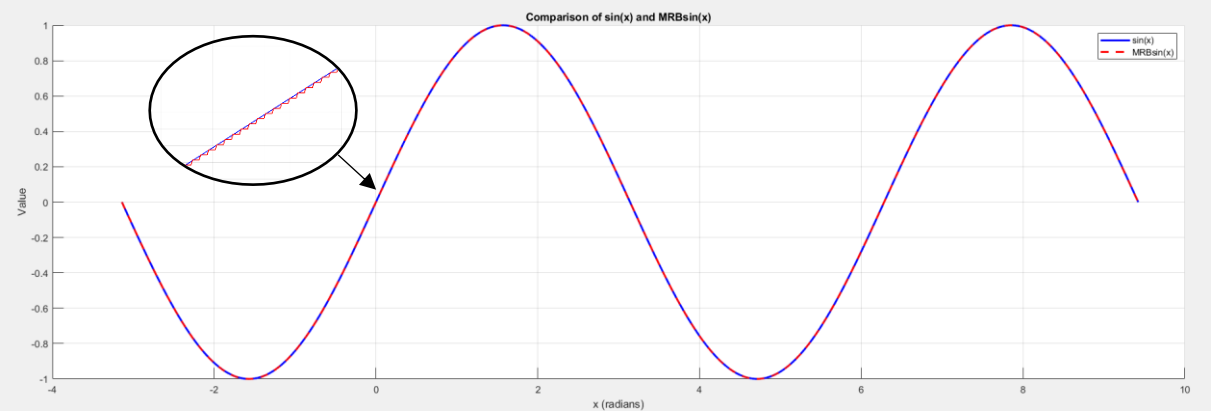

Figure 4 math.h sin(x) function and MRB_MATH_LIB.h sin_f(x) comparison (LUTSIZE = 1)

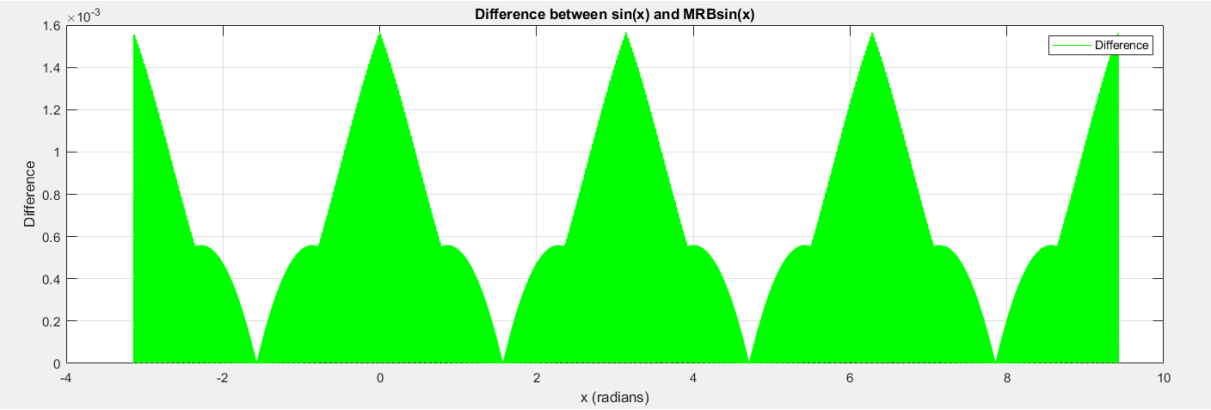

Figure 5 Absolute error between sin and sin_f functions (LUTSIZE = 1)

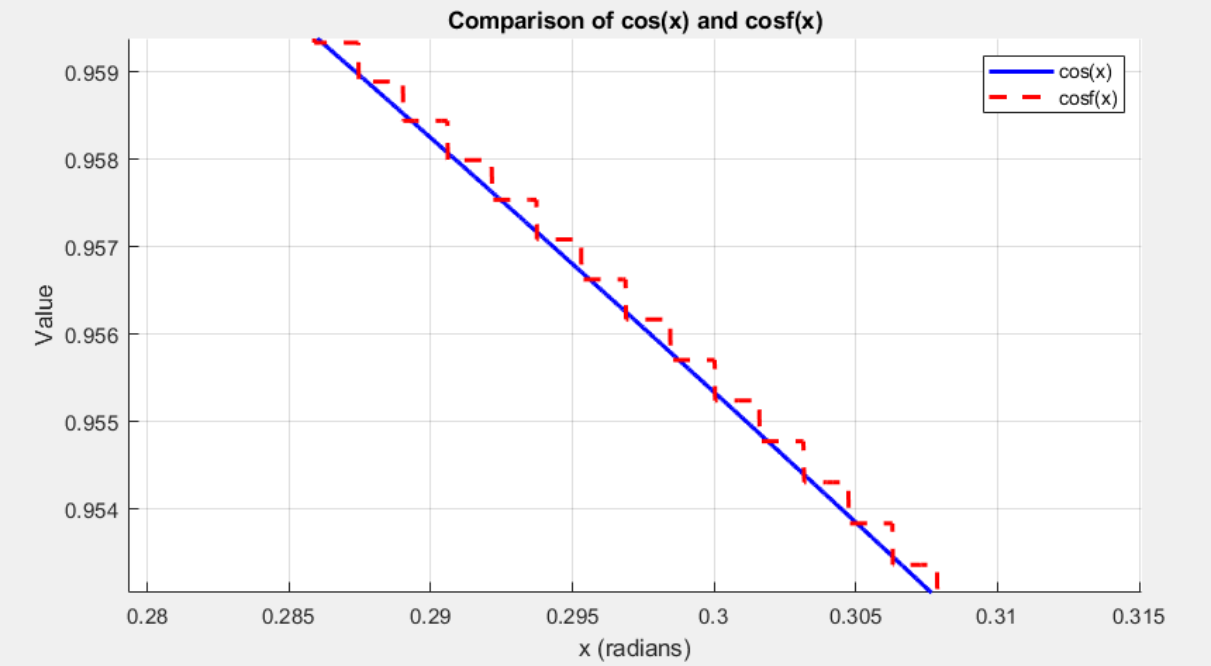

Figure 6 cos and cos_f comparison - zoom in

Taylor series trigonometric functions

In library, there are also trigonometric functions that are estimated from Taylor series (*sin_t* and *cos_t*). The library calculates sinus based on first four elements of Taylor series, as this is sufficient to estimate values from -π and π.

$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \cdot \frac{x^{2n+1}}{(2n+1)!}$$

$$\text{sin\_t}(x) = x\left(1 - x^2\left(\frac{1}{3!} - x^2\left(\frac{1}{5!} - x^2\left(\frac{1}{7!} - \frac{x^2}{9!}\right)\right)\right)\right)$$

Sin_t formula was improved to do as few multiplications as possible and to not use divisions at all (F3, F5 etc. are 1 by factorial of particular number):

```
x = arg * arg;
result_value = arg * (1 - x * F3 * (1 - x * F5 * (1 - x * F7 * (1 - x * F9))));
```

Figure 7 Implemented formula with 4 elements of Taylor series

Functions *sin_t* and *cos_t* are slower than trigonometric functions which use LUT (*sin_f* and *cos_f)*, but they occupy much less space in memory (as LUT is not declared). Important thing to notice is lacking precision for arguments near -π and π. For some arguments near those boundaries, value can exceed 1, what must be considered when testing the stability of the system. Functions *sin_t* and *cos_t* are always available, however user can define LUTSIZE as 0 and quit using the sin_f and cos_f functions, so lookup table won't occupy memory.

```
// **********SIN and COS**********************************//
//You can define lenght of look up table (in kB). The bigger the look up table, the more accurate sin_f and cos_f function are
#define LUTSIZE 0 //4*kB
```

Figure 8 No lookup table. Using only Taylor series based trigonometric functions

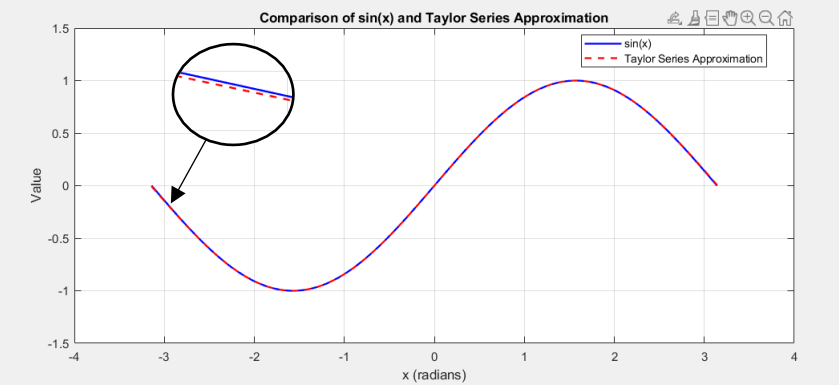

Figure 9 sin and sin_t comparison

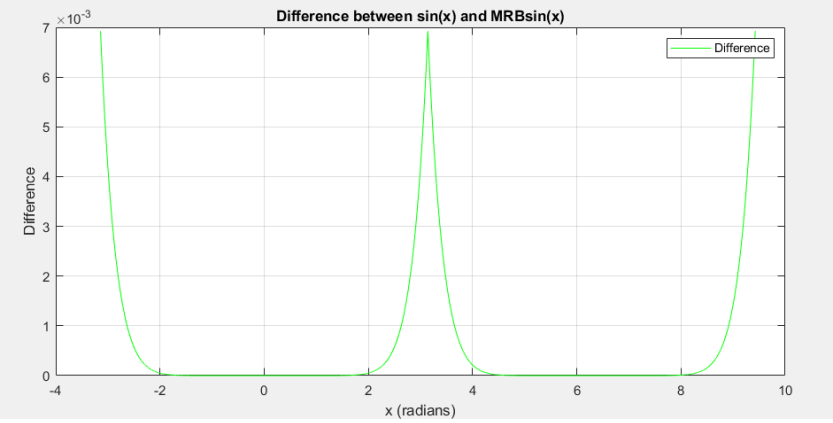

Figure 10 Difference between sin and sin_t functions

Table 2 Mean absolute difference between sin and sin_f (or cos and cos_f) with 1e-5 angle step.

LUTSIZE(n*4kB)	**1n**	**2n**	**4n**	**0 (Taylor series)**
Mean Absolute error	0.000326	0.000163	0.000082	0.000582

Square root function

The fast square root function is based on inverse square root Quake's algorithm. The Quake algorithm avoids directly calculating the square root. Instead, it uses a fast approximation, combined with Newton's method for refinement. The key trick is in using bit-level manipulation of the floating-point number to produce a rough initial guess.

```
//Quake algorithm
float fast_invsqrt(float number) {
    long i;
    float x2, y;
    const float threehalfs = 1.5F;
    if (number <= 0.0f) return 0;
    x2 = number * 0.5F;
    y = number;
    i = *(long*)&y;                         // Treating float number as hex
    i = 0x5f3759df - (i >> 1);              // Mantysa hack - emulating the ^(-1/2) operation
    y = *(float*)&i;                        // Back to float coding
    int j = 0;
    // Additional iterations for better accuracy
    for (j = 0; j < SQRT_ACCURACY; ++j) { // Adjust the number of iterations as needed
        y = y * (threehalfs - (x2 * y * y)); // Newton's iteration
    }
    // Additional iterations can be added here for better accuracy

    return y;
}
```

Figure 11 Quake algorithm

Step-by-step Quake's algorithm description:

Approximation using a "magic number":

- The number *x* is first reinterpreted as an integer (using **(long int*)&x*), which allows bit manipulation – e.g. float number *1.0f* would be reinterpreted as *1065353216*.
- This integer is then modified using a "magic constant" **0x5f3759df**, which was empirically determined. The expression *i = 0x5f3759df - (i >> 1)* performs the bit manipulation to get an initial approximation of $\frac{1}{\sqrt{x}}$.
- This step essentially tricks the floating-point representation into giving an initial estimate that's close to the correct result.

Convert the bits back to a float:

- After the bit manipulation, the result is cast back into a floating-point value (using **(float*)&i*), so now *y* is a rough approximation of $\frac{1}{\sqrt{x}}$.

Refinement using Newton's method:

The result is then refined with one iteration of Newton's method to improve the approximation. This step reduces the error in the approximation. The formula used is:

$$x = x \cdot (1{,}5 - 0{,}5 \cdot x^2 \cdot y)$$

After that, result of Quake's algorithm is multiplied by input value, so division is avoided again (which increases execution speed a lot). Instead $sqrt = 1 \,/\, (invsqrt)$ function does:

$$sqrt = x \cdot (invsqrt)$$

```
//To avoid harsh dividing, we use formula
// x * (1/sqrt(x)) = sqrt(x)
float fast_sqrt(float number) {
    return number * fast_invsqrt(number);
}
```

Figure 12 Inverse-Inverse square root

User can adjust the function by changing *SQRT_ACCURACY* parameter. Increasing SQRT_ACCURACY also increases linearly execution speed of the function. The relative error (relative to **sqrt** function from the math.h library) and the execution time of the function for a given parameter value are summarized below. The error was presented in relative form due to the very large range of values tested by the **fast_sqrt** function.

```
// ***********SQRT*****************************************//
// fast qrt accuracy is directly connected with number of cycles needed to count it
// increasing sqrt_accuracy by one effects linearly increasing execution time of fast_sqrt
#define SQRT_ACCURACY 5
```

Figure 13 Adjusting fast quare root accuracy - in default equal to 5

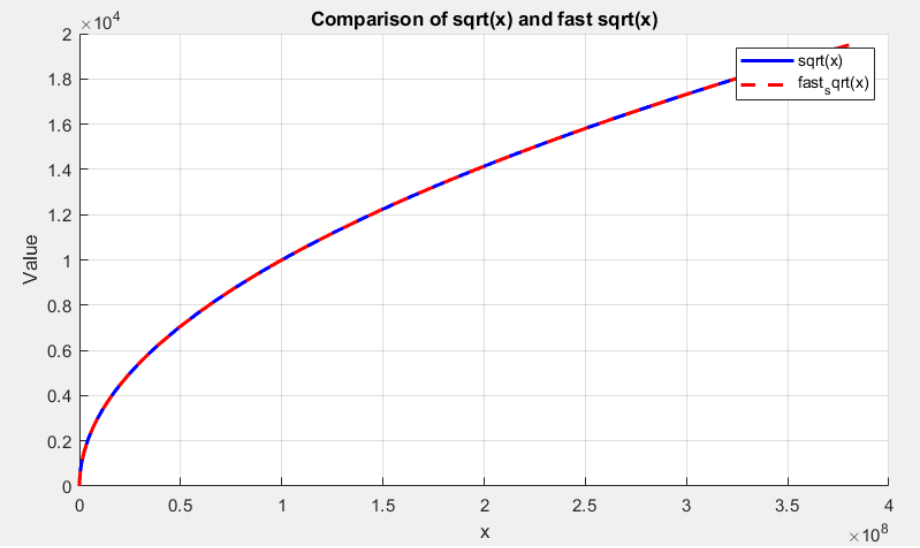

Figure 14 sqrt from math.h and fast_sqrt comparison

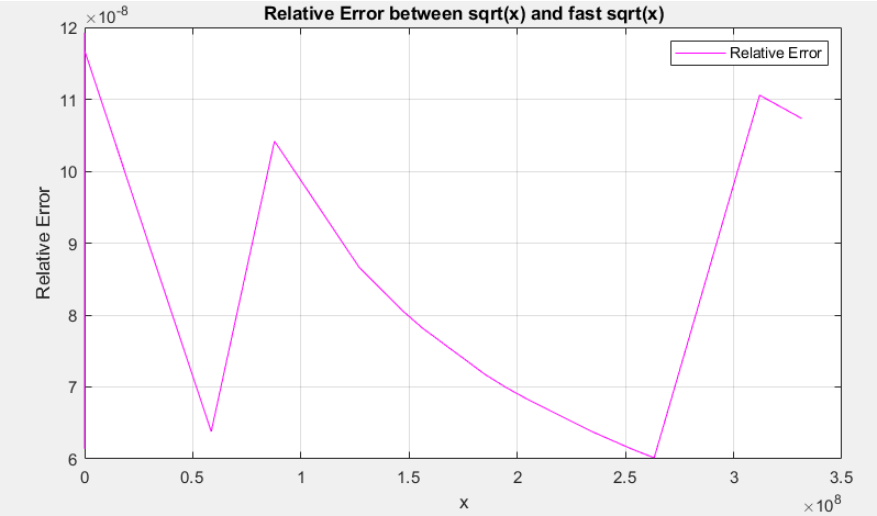

Figure 15 Relative error [%] of fast_sqrt function in comparison to sqrt function from math.h library

Table 3 Mean relative error of fast_sqrt function with SQRT_ACCURACY parameter change

SQRT_ACCURACY	Mean relative error	Function execution speed
2	1,6e-4 [%]	17 cycles
3	3,5e-6 [%]	21 cycles
5	2,3e-6 [%]	26 cycles
10	2,2e-6[%]	44 cycles
20	2,1e-6[%]	90 cycles

Root mean square function

As is this one of the most commonly needed tools in digital signal processing, the root mean square function (RMS) has also been added to MRB_MATH library. Three variants of the RMS function are included in this library: normal, fast and rapid. The normal variant is based on the sqrt function included in the math.h library. The fast variant is based on the fast_sqrt function contained in the MATH_MRB library. The rapid variant uses mathematical relationships assuming a perfectly sinusoidal waveform, so it does not use division or root operations. The accuracy of the various variants and their execution times for different signals is summarized below.

The user must set the parameters of the measured RMS signal in the preprocessor directives before using any of this functions (this must be done in the preprocessor, in order to avoid using the malloc() function to create a sample buffer). Important information is signal measuring frequency, base frequency for which RMS should be calculated and buffer size (should be counted as *measuring frequency divided by base frequency*). *RMS_HANDLERS* corresponds to number of buffer arrays that are declared for this scope. Each signal should have different buffer. That means, e.g. for three phase system, there are three current measurements and three voltage measurements, so *RMS_HANDLERS* should be equal to 6.

```
// **********RMS*****************************************//
//For proper RMS calculation, you have to adjust
#define MEASURING_FREQUENCY 10000
#define BASE_FREQ 50
#define BUFFER_SIZE 200 // (MEASURING_FREQUENCY / BASE_FREQ)
#define RMS_HANDLERS 6
//Buffer size has be counted manually before compilation, so there will be no need of using malloc()
```

Figure 16 Setting parameters of RMS functions

```
#define current_A 0
#define current_B 1
#define current_C 2
#define voltage_A 3
#define voltage_B 4
#define voltage_C 5
current[0] = rapid_RMS(ADC1[i], current_A);
current[1] = rapid_RMS(ADC2[i], current_B);
current[2] = rapid_RMS(ADC3[i], current_C);
voltage[0] = rapid_RMS(ADC4[i], voltage_A);
voltage[1] = rapid_RMS(ADC5[i], voltage_B);
voltage[2] = rapid_RMS(ADC6[i], voltage_C);
```

Figure 17 Approach for use of RMS function with multiple occurrences

Table 4 Execution speed comparison for different RMS functions

Execution speed	
RMS	694 cycles
fast_RMS	46 cycles
rapid_RMS	13 cycles

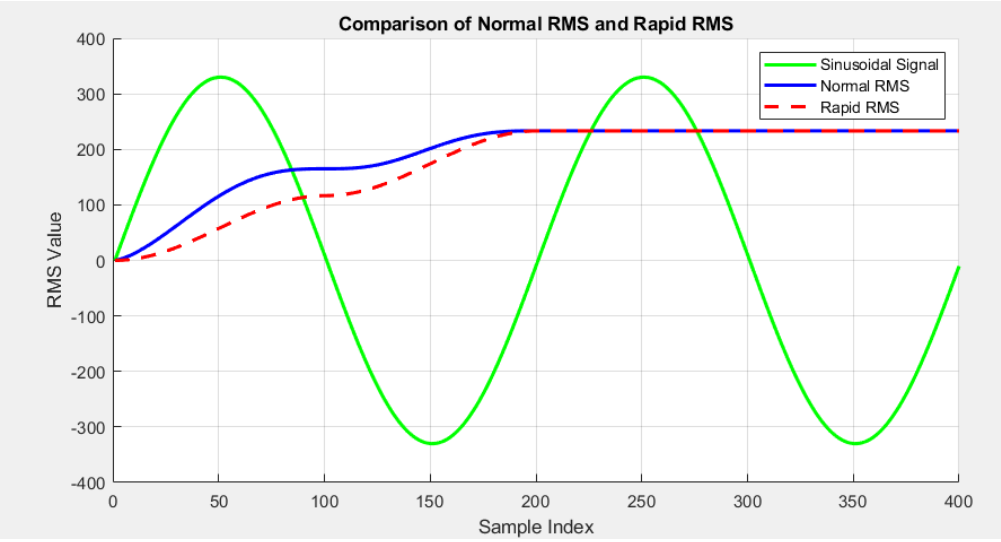

Figure 18 Comparison of transient state of normal RMS and rapid RMS for ideal sinusoidal signal

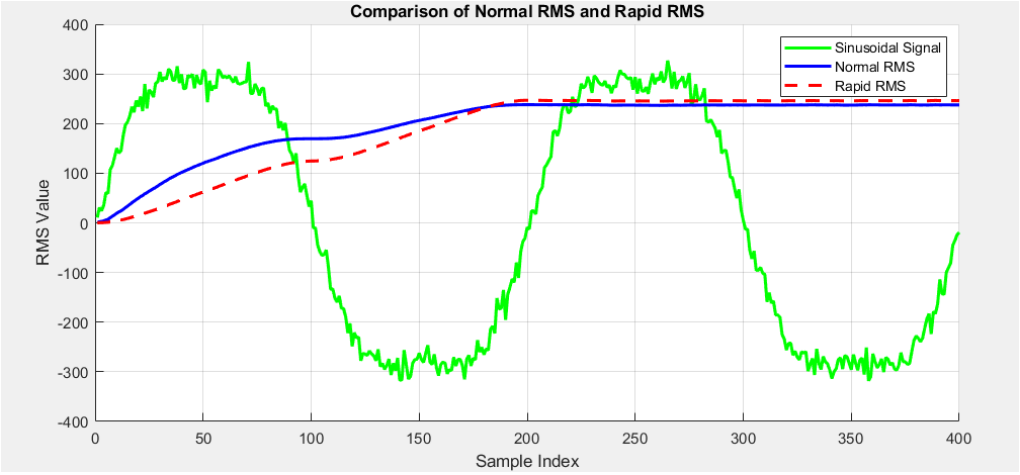

Figure 19 Comparison for noisy signal with third harmonic injected

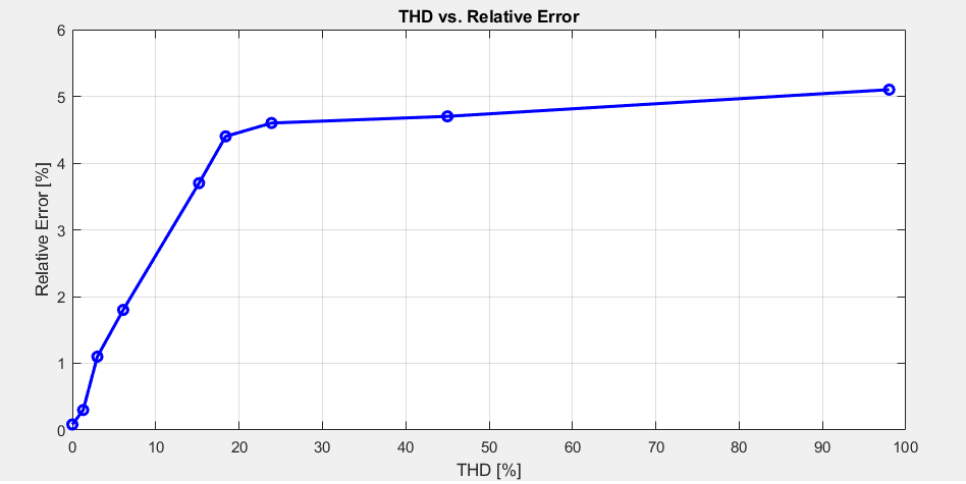

Figure 20 Relative error of rapid_RMS in comparison to normal RMS

Discrete Fourier transform function

The last tool in the library is a discrete Fourier transform function designed for real time and non-real time computing on a microcontroller. Samples can be provided each cycle or can be buffered and given to DFD in selected time. Before using, user have to adjust preprocessor parameters: *DFT_HANDLERS, SAMPLING_FREQ, BASE_FREQ, MAX_HARMONIC* and *DFT_RESULUTION*. Macro *MAX_HARMONIC* corresponds with the highest frequency to which algorithm will count. The more harmonics we want to calculate, the longer computing will be needed. *DFT_ RESULUTION* sets the difference between samples in **DFT** results. With accuracy equal to one, results are received with accuracy to one harmonic (results table will look like: h1, h2, h3 etc.). With accuracy equal to two results table will look like: h1, h1.5, h2 etc. Along with increasing resolution, the buffer array is getting bigger and computing time increases. By default, this function uses trigonometric function from math.h (sinf and cosf) as they are more optimized for wider range of angle arguments than MRB_MATH functions – by definition the range of angles for Fourier transform would be up to $2 \cdot \pi \cdot f_N$[1]. If your microcontroller doesn't have FPU you should uncomment *#define NO_FPU* line.

DFT_HANDLERS corresponds to the number of buffer arrays that are declared for this scope. Each signal should have a different buffer. That means if user wants to make online DFT for 3 signals, for optimal memory usage, this parameter should be equal to 3. To store DFT results user also must declare array with proper length (at least half the buffer plus one).

```
// **********DISCRETE FOURIER TRANSFORM******************//
//For proper DFT calculation, you have to adjust these parameters
#define DFT_HANDLERS 1
#define SAMPLING_FREQ 10000
#define DFT_BASE_FREQ 50
#define MAX_HARMONIC 8
#define DFT_RESULUTION 10
//#define NO_FPU
```

Figure 21 Definition of DFT parameters

```
float P1[DFT_BUFFER_SIZE / 2 + 1];
float P2[DFT_BUFFER_SIZE / 2 + 1];

for (int i = 0; i < 2 * DFT_BUFFER_SIZE; i++) {
    float t = i * 0.0001;
    float y = generate_sine_wave(f, t);
    float y2 = generate_sine_wave(2*f, t);
    DFT(y, P1, 0);
    DFT(y2, P2, 1);
}
```

Figure 22 Multiple occurrence of DFT function and result array definition (offline use of DFT)

```
296 __interrupt void
297 cpuTimer0ISR(void)
298 {
299   cpuTimer0IntCount++;
300     // Define signal parameters
301     t = t + delta_t;
302     float y = generate_sine_wave(f, t);
303     DFT(y, P1, 0);
```

Figure 23 Online use of DFT function

[1] Nyquist frequency

DFT is very costly in terms of computing. Execution speed of DFT function highly depends on buffer size (which is determined by the sampling frequency and resolution) and amount of harmonics which will be calculated. The **DFT** function called in program with a sampling frequency most of the time collects samples into a buffer and will perform calculations only after it is full. For a frequency of 50 Hz, the most costly actions for the processor will be performed every 20ms. After the first buffer is full, the buffer is cycled and the next calculation will be performed for a completely new set of samples.

Table 5 Exuction speed of DFT - comparison for different sampling and resolution parameters

Buffer size	Execution speed
20 (1kHz sampling, DFT resolution 1)	25k cycles
100 (5kHz sampling, DFT resolution 1)	48k cycles
200 (10kHz sampling, DFT resolution 2)	139k cycles

Figures 19 and 20 shows Fourier spectrums under various test conditions. Figure 19 shows Fourier spectrum of tested 50,5Hz sinusoidal signal contaminated with 2^{nd} , 3^{rd} , 5^{th} and 7^{th} harmonics. Test results were similar for all frequency sampling conditions.

Second figure presents difference in results between different resolutions. Signal for second test was additionally contaminated with non-integer harmonics (1.1n, 1.2n, 1.5n, 1.6n and 1.8n) every with an amplitude equal to 5% of 1^{st} harmonic amplitude.

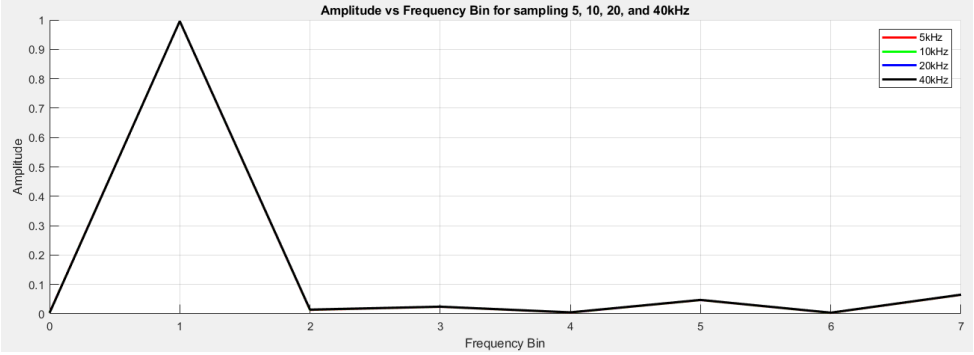

Figure 24 Fourier spectrum - DFT test for non-ideal 50,5Hz sinusoidal signal and 2,3,5,7n harmonics

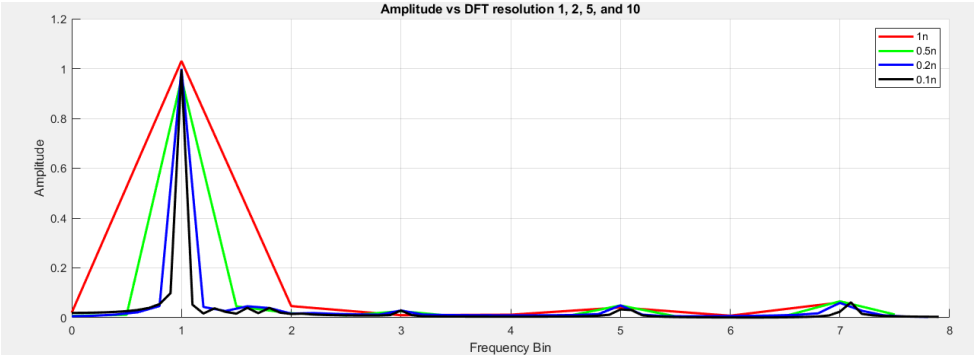

Figure 25 Fourier spectrum - DFT test for different DFT_RESOLUTION parameters