RFT : a simplified fast real-time sliding DFT algorithm

Claude Baumann, Director of the Boarding School "Convict Episcopal de Luxembourg[1]", 5, avenue Marie-Thérèse, m.b. 913, L-2019 Luxembourg[2] claude.baumann@education.lu

September 15th 2005 (updated November 15th 2005)

Abstract

This paper presents a variant of the Discrete Fourier Transform (DFT)[3] that is particularly destined for the implementation in embedded applications, which are characterized by limited resources in terms of memory and computing speed. Often these applications are highly specialized and require therefore an optimal relationship between costs and performance. It surely is of great interest to dispose of algorithms that may combine effectiveness, simplicity, code-shortness and implementation ease. The presented table-based sliding algorithm has all these qualities. As the name suggests, it executes the Fourier analysis on the single sample rather than on an N-sized array of samples, drastically increasing the rate at which the Fourier coefficients are produced in comparison to other DFT methods. This makes it an excellent choice for real-time applications. The algorithm is based on a recursive interpretation of the DFT equations. Furthermore, a look-up table with an astute indexing replaces the time-intensive calls of trigonometric approximation functions. An optimal adaptation of the data types guarantees highest precision and result stability.

1. Introduction

Probably the most frequently operated task in modern computers in any way is the Discrete Fourier Transform (DFT). In signal processing the DFT is used to analyze discrete time signals. Far from being trivial, the DFT is known to be quite complex both in program design and execution. Of the numerous available algorithms the Cooley-Tukey Algorithm[4] is the most commonly used. To express the execution efficiency, the algorithm is called the Fast Fourier Transform (FFT).

If N designates the number of samples that the FFT is supposed to take into consideration, the complexity $\boldsymbol{O}(N)$ may be expressed with $N\log_2 N$ as the result of a *divide an conquer* strategy. Compared to the complexity N^2 of a non-optimized DFT, the gain is considerable. However the disadvantage for real-time applications is the fact that the FFT must be applied to a whole N-sized dataset, where N must be a power of 2. If no channel switching procedure is added, the FFT coefficients are updated at a maximum speed of $f_a = f_s/N$, where f_s is the sampling frequency, regarded the supposition that the FFT computation has been achieved during $1/f_a$.[5]

By contrast, the present simplified sliding algorithm -that we'd like to call *Real-time Fourier Transform* (RFT)- has the particularity that the computing speed does not depend on N, but only on the number of required harmonics. (Remember that in the FFT both values cannot be separated!) The high-speed analysis is operated after each data-sample and considers nothing else but the differences that are generated through the disappearance of the oldest data-value and the incoming of the most recent value, which means that it is not necessary to wait for N values to be read. If the short computation is faster than the data acquisition time of the signal processing device, f_a will equal f_s. Furthermore the algorithm avoids recalculating trigonometric functions all over again, which is one of the principal sources of execution slow-down. Instead the program fixes once at start N cosine and sine values into a constant look-up table (LUT). Finally a considerable speed gain can be obtained, if the choice of the data types is made carefully in function of N, since only a finite number of trigonometric

function-points are appearing. Of course this choice must also consider the initial signal precision that might be given as an ADC resolution.

The RFT-algorithm is amazingly short and easy to implement into any kind of environment, which makes it particularly interesting for embedded devices. The possible applications of the algorithm are as numerous as for the FFT, but the gain may be significant for both the design and the execution in real-time signal processing, especially if the number k_{max} of observed harmonics is small. Additionally, the algorithm can easily be extended to include special data windowing like Hanning-filtering destined to overcome problems that are linked to the discontinuities at the edges of the observed data sets. However, with the RFT algorithm windowing functions in the time domain cannot be applied. Instead corresponding low-pass filter functions must be used in the frequency domain. If the algorithm was exploited as a non-sliding DFT[6], the complexity $\boldsymbol{O}(N)$ can be estimated to $k_{max} \cdot N$, representing a performance that is comparable to the one delivered by the *Goertzel algorithm*[7]. If $k_{max} < \log_2 N$, the algorithm is even faster than the FFT.

2. Some DFT fundamentals

The DFT certainly is one of the workhorses in modern computing. Its principal activity may be summarized as the description of periodicity in digital data. Especially used in digital signal processing, it serves as a powerful signal-analyzer procedure extracting the harmonics that compose the original signal. Basically, the DFT follows the theorem that any periodic function fulfilling the *Dirichlet*-criteria[8] may be expressed as an infinite summation of sinus and cosines:

$$g(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} \left[a_k \cos kx + b_k \sin kx \right]$$

$$a_0 = \frac{1}{\pi} \int_0^{2\pi} g(x) dx$$

$$a_k = \frac{1}{\pi} \int_0^{2\pi} g(x) \cos kx \, dx$$

$$b_k = \frac{1}{\pi} \int_0^{2\pi} g(x) \sin kx \, dx$$

The value $a_0/2$ can be interpreted as the mean-value of all the windowed $g(x)$[9]. The values a_k and b_k are called the coefficients of harmonics or Fourier-coefficients (sometimes ‘twiddle factors’). In many cases a complex number representation of the Fourier series is preferable. The a_k values represent the real components and b_k the imaginary parts. However, for the description of the present algorithm the trigonometric representation is the better choice.

Signals generally are expressed as time-functions[10]. The coefficients are then described as:

$$g(t) = \frac{a_0}{2} + \sum_{k=1}^{\infty} \left[a_k \cos k\varpi t + b_k \sin k\varpi t \right]$$

$$a_0 = \frac{1}{T} \int_0^T g(t) dt$$

$$a_k = \frac{1}{T} \int_0^T g(t) \cos(k\varpi t) dt, \quad \varpi = \frac{2\pi}{T}$$

$$b_k = \frac{1}{T} \int_0^T g(t) \sin(k\varpi t) dt$$

In practical digital signal processing the time-domain data-values are obviously discrete and the equations may be rewritten as finite summations rather than integrals:

$$T = N\Delta t, \quad t = i\Delta t$$

$$a_0 = \frac{1}{N} \sum_{i=0}^{N-1} g(i\Delta t)$$

$$a_k = \frac{1}{N} \sum_{i=0}^{N-1} g(i\Delta t) \cos(k\varpi \cdot i\Delta t)$$

$$b_k = \frac{1}{N} \sum_{i=0}^{N-1} g(i\Delta t) \sin(k\varpi \cdot i\Delta t)$$

$$\varpi = \frac{2\pi}{N\Delta t}$$

$$\Rightarrow \begin{cases} a_k = \frac{1}{N} \sum_{i=0}^{N-1} g(i\Delta t) \cos\left(ki \cdot \frac{2\pi}{N} \right) \\ b_k = \frac{1}{N} \sum_{i=0}^{N-1} g(i\Delta t) \sin\left(ki \cdot \frac{2\pi}{N} \right) \end{cases} \Leftrightarrow \begin{cases} a_k = \sum_{i=0}^{N-1} g(i\Delta t) \frac{\cos\left(ki \cdot \frac{2\pi}{N} \right)}{N} \\ b_k = \sum_{i=0}^{N-1} g(i\Delta t) \frac{\sin\left(ki \cdot \frac{2\pi}{N} \right)}{N} \end{cases}$$

3. FIFO signal buffer with circular index

The RFT algorithm uses a simple windowing First In First Out (FIFO) data array that always accepts N data values. Practically the buffer must initially be zeroed. A circular index-pointer selects the current data-value. If a new data value is entered to the buffer-array, the old value is overwritten and the index is incremented modulo N.

At any sampling time $t_i + uT = i\Delta t + uT$, where $i \in [0,N-1]$ and $u \in [0,+\infty[$, both being integer-values, let y_i be the old data value that is going to disappear of the window with the arrival of the new data value y_i'. The formulas may be transformed as follows:

$$\begin{cases} a'_k = a_k + (y'_i - y_i)\dfrac{\cos\left(ki \cdot \dfrac{2\pi}{N}\right)}{N} \\ b'_k = b_k + (y'_i - y_i)\dfrac{\sin\left(ki \cdot \dfrac{2\pi}{N}\right)}{N} \end{cases}$$

$$i \leftarrow (i+1) \bmod N$$

These recursive equations demonstrate that the Fourier coefficients may be computed at any new sampling moment by only considering the difference of the recent and the old value multiplied with the respective trigonometric functions. The most costly calculations obviously are the sine and the cosine values. But it can easily be shown that during the circular operations the same trigonometric values reappear over and over again.

The recursion that is employed in the present algorithm reminds other sliding DFT methods that all are characterized by the fact that the transform is computed after each single sample period instead of every N samples[11].

4. Look-up table (LUT)

Normally in the FFT, k and i have the same dimension, which means that the result of the FFT is an N dimensioned array, where the Fourier coefficients are reproduced twice. In the RFT, this redundancy can be omitted by considering k_{max} = N/2. Choosing N as an even number will prevent the loss of the last harmonic. The RFT algorithm allows restricting k_{max} to any number ≤N/2, if less harmonics are likely to be observed, or even to a selection of interest, if so desired. This leads to the conclusion that the set-dimension of those values from which trigonometric functions are calculated is necessarily limited to $N^2/2$. But, since the functions are 2π-periodic, we have:

$$\forall ki,$$

$$\frac{ki}{N} = (ki \ DIV \ N) + \frac{(ki \bmod N)}{N}$$

$$\Rightarrow \sin\left(ki \cdot \frac{2\pi}{N}\right) = \sin\left((ki \ DIV \ N)2\pi + \frac{(ki \bmod N) \cdot 2\pi}{N}\right) = \sin\left((ki \bmod N) \cdot \frac{2\pi}{N}\right)$$

The trivial inequality $ki \bmod N < N$ proves that no more than N different sine values appear in the computations. (Since *k* may equal 1 and *i* covers the range 0..N-1, there are in fact exactly N values!) The same is true for the cosine values. Instead of operating the time-intensive trigonometric calculations, it is therefore a better idea, to set up a N-sized look-up table (LUT) containing the sine and cosine function values and have an index j= $ki \bmod N$ point to the respective value. However:

$$\forall x, \quad \cos x = \sin\left(x + \frac{\pi}{2}\right)$$

$$\Rightarrow \quad \forall ki, \quad \cos\left(ki \cdot \frac{2\pi}{N}\right) = \sin\left(\left(ki + \frac{N}{4}\right) \cdot \frac{2\pi}{N}\right)$$

If N is chosen as a multiple of 4, a second index $q = [ki+(N \text{ div } 4)] \bmod N$ can be used to point to the cosine, while *j* points to the sine, and only one LUT is required containing the sine values alone. Since N is constant during the whole operation, the constant LUT can be set up once at program start as follows:

$\forall N \in \{4,8,12,16.....\}$

$\forall i \in \{0,1,2,3...N-1\},$

$$LUT(i) = \frac{\sin\left(i \cdot \frac{2\pi}{N}\right)}{N}$$

and the Fourier coefficients are expressed as:

$\forall i \in \{0,1,2,3...N-1\}$

$\forall u \in \{0,1,2,3....\}$

wait for next sampling period $t_i + uT$

$a'_0 = a_0 + (y'_i - y_i) \cdot LUT(N \ div \ 4)$

iterate $k := 1 \ to \ k_{max}$

$$\begin{cases} j = ki \bmod N \\ q = [j + (N \ div \ 4)] \bmod N \\ a'_k = a_k + (y'_i - y_i) LUT(q) \\ b'_k = b_k + (y'_i - y_i) LUT(j) \end{cases}$$

$i \leftarrow (i+1) \bmod N$

Thus, at any sampling moment t_i+uT, only a very restricted number of computations must be executed.

5. A simple implementation of the RFT in pseudo-code

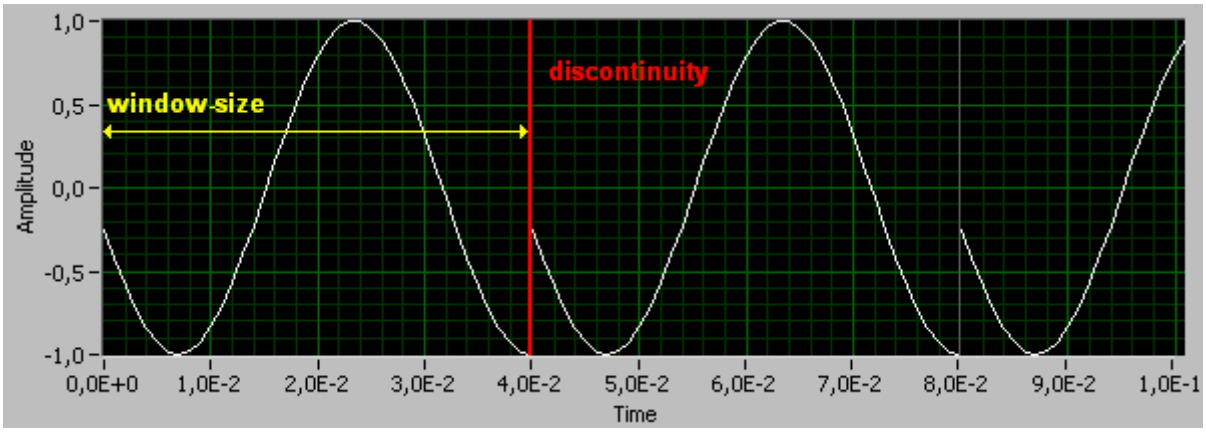
```
|=======================|
|  constant definition:  |
|=======================|

integer:    N                       //number of samples; multiple of 4
            COS_OFFSET = N DIV 4    //shifts by PI/2
            K_MAX                   //number of harmonics <=N/2

|=======================|
|  variable definition:  |
|=======================|

integer:    i     //index (samples)
            k     //index (harmonics)
            j     //index (points to the sine-value in the LUT
            q     //index (points to the cosine-value in the LUT
            new_val, old_val, diff  //temporary variables

array 0..N-1 of integer:     f     //ring-buffer for signal data
array 0..N-1 of real:        LUT   //look-up-table for sin(i*2*PI/N)/N
array 0..K_MAX of real:      A     //Re-coefficients
array 0..K_MAX of real:      B     //Im-coefficients; B[0] never used !

|=======================|
|     initialization     |
|=======================|

for i=0 to N-1 do
```

```
{
      LUT[i]:=SIN(i*2*PI/N)/N       //initialize LUT
      f[i]:=0                       //zero data buffer
}

for k=0 to K_MAX do
{
      A[k]:=0                       //zero coefficients
      B[k]:=0
}

i=0                                 //start with 0th data-value

|=======================|
|     RFT-algorithm     |
|=======================|

while TRUE do
{
      wait for time-event           //timer ticks dt to produce regular
                                    //hard real-timing or timer interrupt
      new_val:=get sample from signal channel
      old_val:=f[i]                 //the value that is going to disappear
      f[i]:=new_val                 //store current sample
      diff:=new_val-old_val         //produce difference
      A[0]:=A[0]+diff*LUT[COS_OFFSET]    //this divides by N !!!

      For k=1 to K_MAX do           //do all the harmonic bins
      {
            j:=k*i MOD N            //fix sine-index
            q:=(j+COS_OFFSET) MOD N //fix cosine-index
            A[k]:=A[k]+diff*LUT[q]  //only add difference multiplied by
            B[k]:=B[k]+diff*LUT[j]  //trig-functions
      }
      i:=(i+1) MOD N                //next sample
}

|=======================|
|     End of program    |
|=======================|
```

6. Discontinuities

Any DFT encounters the fundamental problem of discontinuities. Suppose that an undesired single discontinuity happens in the original signal. The FFT will return the disturbed set of coefficients once, but in the presented RFT the twiddle factors will remain invalid for the duration of the propagation of the discontinuity. Nonetheless the transform won't be wrong for a longer time than for the FFT. For both methods the invalid state will exactly last for $1/f_b$, where f_b is the FFT bin width and $f_b=f_s/N$.

Another type of discontinuity may be related to the *windowing problem*. The origin of the difficulty is the fact that in most of the cases it cannot be assured that the signal windows, which are selected for the DFT exactly match the signal periodicity. The result is a misinterpretation of the periodic signal as if there was a regular discontinuity producing spectral disturbances.



Picture 1: Illustration of the discontinuity that is generated by the DFT windowing.

A large palette of windowing functions has been developed to overcome these problems (Hanning window, Hamming window, Bartlett window, Blackman, Kaiser, Welch, Dolph-Chebychev…) Normally those functions are applied to the time domain data in order to attenuate the signal at the edges of the window. Unfortunately with the RFT, the point of discontinuity slides over the whole window as the result of the ring buffer. Therefore, the windowing functions cannot be applied to the time domain data, but the filtering effect of the windowing procedure must be operated to the frequency domain. The good news is that because the RFT only considers half of the usual frequency interval, there is no doubling of the harmonics as known from the FFT. The consequence is that chosen low-pass filters can easily replace the rather complicated windowing functions. Since there exists an even larger palette of applicable high-speed recursive or non-recursive digital filters, the RFT may assure the same windowing features than the FFT with less effort.

However, simply enlarging the window size, defined by the number of data values N, can also help solving the windowing problem. By this way, many of the spectral disturbances are suppressed in the larger data pool. Remember that for the RFT the time to process the Fourier coefficients is independent of the window size N. This means that enlarging the data sets does not slow down the processing. Nonetheless, especially for embedded applications with limited memory resources, the side effect is that a larger amount of memory is needed.

7. Optimizing the algorithm

The RFT needs a certain computer memory space. First of all there is the ring buffer that must accept N signal data values. The LUT also requires N values. Finally the Fourier coefficients arrays need N/2 locations each. Besides this room, several bytes must be reserved for the few indices that are needed to run the algorithm. However this very limited space together with the extreme code shortness are in no relation to the memory that must be engaged for the FFT, since this particular algorithm is based on a *divide and conquer* strategy that is repeatedly calling a parameterized recursive function to a certain depth. It must be noticed that the impressive data size produced by recursive function calls are normally invisible to the common user. But compilers must reserve the required memory for the activation record of each call.

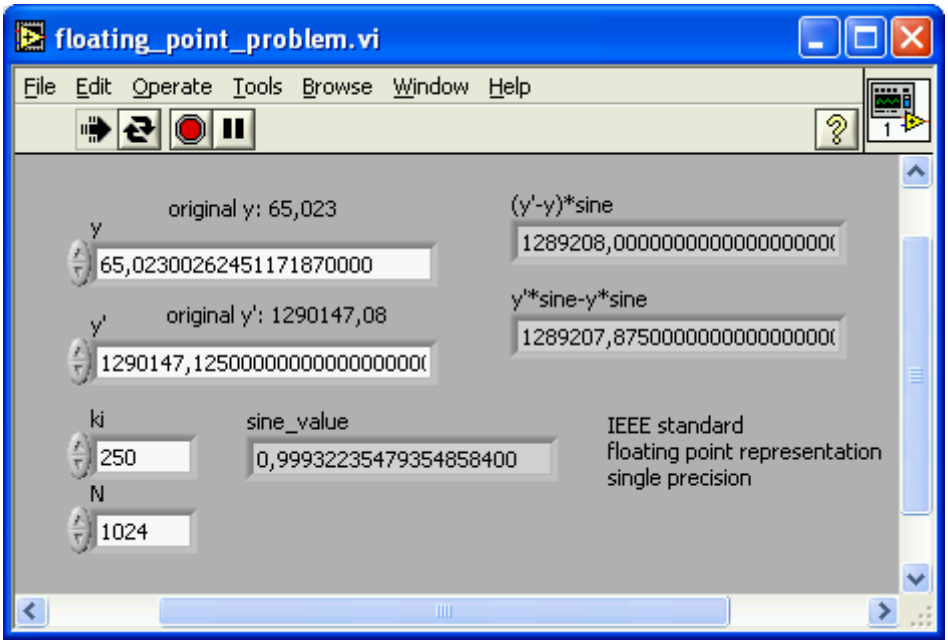
The RFT also has its trade-off. Because the processing must be executed within the small sampling period, the execution speed of one frequency bin update should be optimized according to the sampling period. On the other hand, it is also possible to reduce the number of coefficients in order to improve the computing performance per single sample. Many applications don’t even require the complete bin-set.

It is often underlined that recursive digital functions absolutely need theoretical “infinite” data precision. In fact iterative and even more recursive operations keep the inherent risk of error accumulation that can be the source of result instability. This is excessively hazardous when working with usual floating-point representations. The well-known problem is that in the floating-point representation numbers are not regularly spread over the data range. The precision of a value X is a direct function of $\log_2(X)$.[12] All this necessarily concerns a recursive DFT.

In order to guarantee result stability it must be made sure that the recursive operations don’t introduce fundamental errors that can accumulate. One error-source is the possibility that the following mathematically non-sense inequality might become true, due to floating-point computing errors:

$$b_k + (y'_i - y_i)LUT(j) \neq b_k + y'_i\, LUT(j) - y_i LUT(j) \quad ?$$

which could result in the consequence that the value $y_i LUT(j)$ having been added to the bin at t_i doesn’t equal the theoretical identical value $y_i LUT(j)$ that is subtracted at t_i+T just before writing the new data-point!! It can be retained that the permanent discontinuity, generated by the windowing problem, increases the probability that $\log_2(y'_i) \neq \log_2(y_i)$, which means that it is probable that two numbers of different precisions are subtracted, introducing a small error at each iteration. It cannot be predicted whether the errors compensate each other or are accumulated over time. While programming the RFT algorithm the programmer must be aware of this and search for remediation.



Picture 2: A simple illustration that visualizes that for certain values the distributive rule may not be true! This may corrupt computing results, if the generated errors are accumulated over time. The origin of the troubles is the fact that usual standard floating-point data types cannot represent all the values correctly and proceed to irregular rounding or truncating.

The safest method to avoid these problems is to change to a fixed-point representation. This perfectly makes sense, because, by contrast to the floating-point representation, the fixed-point values, being nothing else but scaled integers, guarantee absolute precision, since a regular repartition is realized over the data-range. If precautions have been taken of avoiding overflow or division by zero, there is no risk that the fundamental arithmetic operations will ever produce errors. Two major arguments support the switching to the scaled integer representation:

- Signals are digitalized within finite resolutions
- Only a selection of trigonometric function-values appears in the RFT.

Since the sine and cosine values evolve in the interval [-1,1], they easily can be represented as scaled integers in the LUT covering the range [-c, c] and respecting the guideline that any consecutive scaled integer numbers must be unequal (bijection condition). This condition is given if:

$$\text{int}\left(c\cdot\sin\left(\frac{\pi}{2}+\frac{2\pi}{N}\right)\right)\neq\text{int}\left(c\cdot\sin\left(\frac{\pi}{2}+\frac{4\pi}{N}\right)\right),$$

where c is a constant that should be fixed at program start.

The LUT-values then are defined as:

$$\forall i\in\{0,1,2,3...N-1\},$$

$$LUT(i)=\text{int}\left(c\cdot\sin\left(i\cdot\frac{2\pi}{N}\right)\right)$$

The Fourier coefficients must be computed differently, where only integer sums are used recursively:

$$\forall N\in\{4,8,12,16.....\}$$

$$\begin{cases}S_{a'_k}=S_{a_k}+(y'_i-y_i)LUT(q)\\S_{b'_k}=S_{b'_k}+(y'_i-y_i)LUT(j)\end{cases}$$

$$q=[ki+(N\ div\ 4)]\bmod N$$

$$j=ki\bmod N$$

$$\begin{cases}a'_k=\dfrac{S_{a'_k}}{c\cdot N}\\b'_k=\dfrac{S_{b'_k}}{c\cdot N}\end{cases}$$

A positive side effect of this approach is that the processing speed is drastically increased, as floating-point normalizing and aligning procedures aren't necessary anymore. The precision doesn't suffer at all, because the divisions don't intervene in the recursions and, supposed that the original signal has an m bit resolution, we have the following set of valid bijection rules:

$let \quad range = [-c \cdot 2^{m-1}, c \cdot 2^{m-1}]$

$\forall d, d_1, d_2, d_4 \in [0,1,2,3,....2^{m-1}],$

$\forall l, l_1, l_2, l_3, l_4 \in LUT \subset [-c, c],$

1. $l + 0 = l$
2. $l - l = 0$
3. $l_1 + l_2 = l_2 + l_1 \in range$
4. $l_1 + (l_2 + l_3) = (l_1 + l_2) + l_3$
5. $d = 0 \Rightarrow d \cdot l = 0$
6. $l = 0 \Rightarrow d \cdot l = 0$
7. $d = 1 \Rightarrow d \cdot l = l$
8. $l = 1 \Rightarrow d \cdot l = d$
9. $d \cdot l = l \cdot d \in range$
10. $d \cdot (l_1 + l_2) = d \cdot l_1 + d \cdot l_2$
11. $\forall d \neq 0, \quad l_1 < l_2 \quad \Rightarrow \quad d \cdot l_1 < d \cdot l_2$
12. $\forall l > 0, \quad d_1 < d_2 \Rightarrow d_1 \cdot l < d_2 \cdot l$
13. $\forall l < 0, \quad d_1 < d_2 \Rightarrow d_1 \cdot l > d_2 \cdot l$
13. $(d_1 \cdot l_1 < d_2 \cdot l_2) \wedge (d_3 \cdot l_3 < d_4 \cdot l_4) \Rightarrow d_1 \cdot l_1 + d_3 \cdot d_4 < d_2 \cdot l_2 + d_4 \cdot l_4$

Note: Often the absolute magnitudes of the harmonics are not in the center of interest, but rather the relative values. In these cases the time-costly divisions may be omitted.

8. Optimized code implementation

```
|=======================|
|  constant definition:  |
|=======================|

integer:    N                       //number of samples; multiple of 4
            COS_OFFSET = N DIV 4    //shifts by PI/2
            K_MAX                   //number of harmonics <=N/2

|=======================|
|  variable definition:  |
|=======================|

integer:    i     //index (samples)
            k     //index (harmonics)
            j     //index (points to the sine-value in the LUT
            q     //index (points to the cosine-value in the LUT
            c     //LUT scale
            new_val, old_val, diff  //temporary variables



array 0..N-1 of integer:     f     //ring-buffer for signal data
array 0..N-1 of integer:     LUT   //look-up-table for c*sin(i*2*PI/N)
array 0..K_MAX of real:      A     //Re-coefficients
array 0..K_MAX of real:      B     //Im-coefficients; B[0] never used !

array 0..K_MAX of integer:   S_A   //Re-sums
array 0..K_MAX of integer:   S_B   //Im-sums; S_B[0] never used !
```

```
|========================|
|     initialization     |
|========================|

function: determine c   //required scaling factor; condition
                        //int(c*sin(PI/2+2*PI/N))<>int(c*sin(PI/2+4*PI/N))

inv_N:=1/N              //this allows replacing division by multiplication
inv_c_N:=1/c/N          //idem

for i=0 to N-1 do
{
      LUT[i]:=c*SIN(i*2*PI/N)       //initialize LUT
      f[i]:=0                       //zero data buffer
}

for k=0 to K_MAX do
{
      S_A[k]:=0                     //zero recursively used sums
      S_B[k]:=0
}

i=0                                 //start with 0th data-value

|========================|
|     RFT-algorithm      |
|========================|

while TRUE do
{
      wait for time-event           //timer ticks dt to produce regular
                                    //hard real-timing or timer interrupt
      new_val:=get sample from signal channel
      old_val:=f[i]                 //the value that is going to disappear
      f[i]:=new_val                 //store current sample
      diff:=new_val-old_val         //produce difference
      S_A[0]:=S_A[0]+diff           //integer only

      A[0]:=S_A[0]*inv_N            //get the correct k0 coefficient
      j:=i                          //point to first sine-value in the LUT

      For k=1 to K_MAX do           //do all the harmonic bins
      {
            q:=(j+COS_OFFSET) MOD N       //fix cosine-index
            S_A[k]:=S_A[k]+diff*LUT[q]    //only add difference*trig
            S_B[k]:=S_B[k]+diff*LUT[j]

            A[k]:=S_A[k]*inv_c_N    //re-scale
            B[k]:=S_B[k]*inv_c_N    //two floating-point mult. per harmonic
            j:=(j+i)MOD N           //produces j=k*i MOD N faster
      }
      i:=(i+1) MOD N                //next sample
}

|========================|
|     End of program     |
|========================|
```

Note: Because normally compilers calculate the integer modulo function as part of the quite costly integer division, a not insignificant gain of speed can be obtained by replacing the index-incrementing followed by the MOD function with the function INCR_MOD below.

```
Function INCR_MOD(x,y,N:integer):integer       //x<N and y<N   !!!!
```

```
      Temporary variables: tmp,delta:integer
{
      tmp:=y+x          //increment y by x
      delta:=tmp-N      //if tmp exceeds N, apply modulo
      If delta>=0 then result:=delta else result:=tmp
}
```

Code optimizing compilers also would prefer another instruction order, for the reason that accesses to arrays are more costly than to numeric variables:

```
S_A[k]:=S_A[k]+diff*LUT[q]
A[k]:=S_A[k]*inv_c_N

S_B[k]:=S_B[k]+diff*LUT[j]
B[k]:=S_B[k]*inv_c_N
```

8. Practical performance test

In order to get an idea about the performance of the algorithm in a real environment, we tested it with the LEGO RCX device. This module has been invented as a sophisticated toy, but, since its release in 1998 it has found many applications as an educational tool in schools, high schools and even universities. The heart of the RCX is a 16MHz clocked Hitachi H8/3292 micro-controller. This device combines many functions: 43 input/output ports (16 of which can drive LEDs), 8 input-only ports, 8 ADC with a 10-bit resolution of which only 4 channels are in use with the RCX (17µs acquisition time at 16MHz), serial communication interface, watchdog-timer, 16-bit free-running timer, two 8-bit timers, 16kB ROM, 512B on-chip RAM. The CPU has eight 16-bit registers r0..r7 or sixteen 8-bit registers r0H, r0L, … r7H, r7L (r7 is used as the stack-pointer); 16-bit program counter; 8-bit condition code register (flag register); register-register arithmetic and logic operations; 8 or 16-bit register-register add/subtract (125ns at 16MHz); 8·8-bit multiplying needs 875 ns; 16÷8-bit division with 875ns; concise instruction set (lengths 2 or 4 bytes); MOV instruction for data transfer between registers and memory (9 different addressing modes).

Picture 3: The RCX H8 micro-controller in its FP-64 package

Together with 32k external RAM, LCD-diplay, buttons, infrared communication module, analog sensor ports and H-bridge output ports, the RCX is an ideal instrument for exploration of micro-controllers in educational contexts. There exist various programming environments, and also firmwares that unlock the device-features at different degrees. Our own development tool called *Ultimate ROBOLAB* that is based on the graphical programming language LabVIEW is a most powerful software-package that allows RCX programming at micro-controller level. The graphical code is directly converted into compact machine code; each program is an individual RCX firmware.

The optimized algorithm has been implemented with *Ultimate ROBOLAB* avoiding the final floating-point rescaling. The computing time for one single sample-iteration was 3.94ms with k_{max}=25 and 1.62ms with k_{max}=10. Replacing the modulo function with the INCR_MOD

function reduced the duration to 3.35ms with k_{max}=25, which represents a gain of 15.5%. Note that a minimal system kernel needed to be run in order to dispose of the data acquisition device. For various reasons the standard RCX sensor ports are sampled at about 300Hz. This rate has been maintained in the standard *Ultimate ROBOLAB* configuration, but may be increased to much higher values. Under standard conditions the RFT-update frequency is able to reach the sampling rate if few harmonics only are observed. This practical verification illustrates well, how efficiently the algorithm can be implemented in small micro-controller environments.

9. Conclusion

The presented sliding real-time Fourier transform algorithm (RFT) is an ideal solution for embedded applications, where limited resources in terms of processing speed, program memory size and costs in general play a role. Besides this, the advantages compared to commonly used FFT methods are that the Fourier coefficients are updated at a higher rate than the usual bin width. Depending on the computing speed per sample, the analysis can reach the maximum speed, which is the data sampling time itself. The simplicity of the algorithm predestinates it for a use with other transform methods like the cosine-transform.

---

[1] „Convict“ doesn't have the same meaning in French-spoken countries than in Anglo-Saxons. The term is derived from the Latin „convivere“ which roughly means „living together“.

[2] Since 1999, besides his administrative tasks, the author teaches advanced LEGO Mindstorms robotics in after-school classes (K6-K12) and maintains the related widely known website www.convict.lu/Jeunes/RoboticsIntro.htm. He participates at the evaluation of the LabVIEW-based ROBOLAB software that is originated from Tufts University Massachusetts in cooperation with the LEGO Company and National Instruments. For instance he has been in charge -in collaboration with Prof. Chris Rogers, Tufts University- of the realization of Ultimate ROBOLAB, a cross-compiler software that allows graphical programming of the Hitachi H8 micro-controller and a few models of the Microchip PIC family. He has been the assessor of various high school robot projects, author and co-author of several related articles and conferences. In 2004, he has been keynote-speaker at the First Annual ROBOLAB Conference in Austin/TX.

[3] KRÜGER K.E.: Transformationen, GE, 2002, pp.20

[4] COOLEY, J.W. & J.W. TUKEY: *An algorithm for the machine calculation of complex Fourier series. Math. Of Computations*, 19:297-301, 1965

[5] f_a is also equals the FFT *bin width* $f_b=f_s/N$.

[6] Only a few minor changes must be added to the algorithm in order to obtain a non-sliding version!

[7] an example : BEARD J.H., St.P.Given, B.J.Young: *A discrete Fourier transform based digital DTMF detection algorithm*, Dep. of Electrical and Computer Engineering, Mississippi State University, 1995

[8] DIRICHLET conditions: The function must have a finite number of discontinuities and a finite number of extrema.

[9] Often a_0 is expressed differently as $a_0 = \frac{1}{2\pi}\int_0^{2\pi} g(x)dx$. In this case the Fourier series is written as: $g(x) = a_0 + \sum_{k=1}^{\infty}\left[a_k \cos kx + b_k \sin kx\right]$. Also note that the observed interval may vary from author to author. Sometimes $[-\pi, \pi]$ is preferred to $[0,2\pi]$.

[10] Time intervals also differ from author to author. Sometimes [0,2T] is used, or [-T,T], or [-T/2,T/2]. Normally there are only practical reasons for the various representation habitudes.

[11] Also see CHICHARO Joe F., Mahdi T. KILANI: *A sliding Goertzel Algorithm*, p.283-297, Signal Processing, Vol52, 1996, Elsevier

[12] Excellent documentations to the problem may be found in: PLATO R.: *Numerische Mathematik*, 282-398, Wiesbaden-GE, 2004 or GOLDBERG, D.: *What ever computer scientists should know about floating-point arithmetic*. ACM Computer Surveys, 23:5-48, 1991