Taking the IFFT/FFT Operation in C++ Using the FFTW Library

Document: 200
www.signal-processing.net
andreas_dsp@hotmail.com
Author: Andreas Schwarzinger
Date: September 16, 2020

Introduction

Languages like MatLab and Python have easily accessible libraries that allow you to execute optimized versions of the discrete Fourier transform with a few simple statements. A popular library that can be integrated into your C++ project is called FFTW, but its use requires far more effort on the programmer's part than the simple use cases in MatLab or Python. This document explains the most important parts of the FFTW library, and shows how to build your own I/DFT class with it in Visual Studio. Certainly, as a programmer that understands the DFT well, there is the temptation to write your own DFT routine. The advantage of the FFTW library is that it looks at the available parallel execution facilities in the processor and choses the optimal method to execute a DFT. Thus advanced parallel execution instruction sets such as SSE, SSE2, AVX, AVX2, ACX512 and several others are supported. Your code can thus run optimally on newer and older hardware without any additional programming effort on your part. The FFTW library splits the execution of the DFT into two phases.

The first step is the creation of a plan. This plan is an object that indicates the algorithm that will be used for the DFT calculation, the DFT length and also contains pointers to the input and output DFT buffers. During the creation of the plan, we may request that the library first find the optimal DFT algorithm given the underlying SIMD instructions supported by the hardware. This process yields something called wisdom, which will then be used to create the plan. Calculating this wisdom comes with a slight time penalty during the initial plan creation, but results in optimal speed for all subsequent DFT calculations using that plan. We can also forgo the creation of wisdom during the plan creation. This will result in a faster plan creation but in suboptimal performance of actual DFT calculation. However, for applications that only take a single or very few DFTs, this approach may be preferable. Luckily, wisdom can be saved to a file and simply read in when your application starts. This way, the wisdom truly only needs to be created once. It's the best of both worlds.

The second step is the execution of the plan, which actually computes the DFT. Simply load the DFT input buffer, execute the plan and fetch the result from the output buffer.

References

→ www.fftw.org

→ http://www.fftw.org/faq/

→ http://www.fftw.org/fftw3.pdf

Needed Files

You can run the IDFT/DFT operations in single and double precision floating point formats. The single precision operations are likely faster as they can be better parallelized. The required files are the following.

Precision	.lib (the import library)	.dll (dynamic link library)	Header files
float	libfftw3f-3.lib	libfftw3f-3.dll	fftw3.h
double	Libfftw3-3.lib	libfftw3-3.dll	fftw3.h

Types used in FFTW

The following discussion uses the double precision FFTW functions. The code example at the end of this documents uses the float versions.

→ `fftw_complex` is of type double[2]. Your plan requires I/FFT input and output buffers which will feature entries of this type. FFTW provides a function to dynamically allocate and deallocate this type of variable. Ensure that you dynamically allocate the input and output buffers before creating the plan and deallocate them after deallocating the plan. The number *N* must be equal to or larger than the DFT size.

```
in  = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * N);
out = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * N);


for (dword dwIndex = 0; dwIndex < wFftSize; dwIndex++, itStartEntryInput++)
{
   in[dwIndex][0] = ComplexDataArray.real();
   in[dwIndex][1] = ComplexDataArray.imag();
}

fftw_free(in);
fftw_free(out);
```

→ `fftw_plan` is a pointer type to a plan object, which will also sit on the heap. The pointer is returned during the plan creation and must later be deallocated as follows.

```
fftw_plan p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
// Load the input buffer
fftw_execute(p);
// Fetch result from output buffer
fftw_destroy_plan (p);
```

Of course, there is no need to destroy the plan after you take the DFT via fftw_execute(plan). Ideally, you would execute the plan many times, always refreshing the input buffer and extracting the output for every DFT calculation. The PDF documentation provides the following simple example for a 1 dimensional complex DFT calculation.

```
#include <fftw3.h>
...
{
     fftw_complex *in, *out;  // declare input and output buffer
     fftw_plan p;             // declare pointer fftw plan object.
     ...
     in  = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N); // Allocate buffer
     out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N); // Allocate buffer
     p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE); // Create plan
     ...                                      // Load the input buffer with complex data
     fftw_execute(p); /* repeat as needed */  // Compute DFT
     ...                                      // Unload the output buffer
     fftw_destroy_plan(p);                    // Deallocate the plan
     fftw_free(in);                           // Deallocate the input buffer
     fftw_free(out);                          // Deallocate the output buffer
}
```

The Plan Creation

The plan creation function `fftw_plan_dft_1d()` requires a couple of input arguments that must be discussed. The first input argument, *N*, indicates the size of the DFT. The second and third input arguments are the pointers to the input and output buffers of type `fftw_complex`. The fourth argument indicates the direction of the transform as `FFTW_FORWARD` (= -1) and `FFTW_BACKWARD` (= +1). The last argument is usually `FFTW_ESTIMATE` or `FFTW_MEASURE`. If no wisdom exists, then the `FFTW_MEASURE` flag will cause the time consuming step of creating wisdom. If no wisdom exists, then the flag `FFTW_ESTIMATE` will avoid creating wisdom and create a suboptimal plan. If wisdom exists, then plan creation with either flag will use the wisdom to accelerate the creation of a plan that will execute (take the I/DFT) very quickly. The PDF tutorial has more information on this topic in section 2.1.

Wisdom

Internal to the FFTW library, there exists a planner which accumulates wisdom as plans are created with the `FFTW_MEASURE` flag. You can also create a plan using the `FFTW_PATIENT` and `FFTW_EXHAUSTIVE` flags, both of which take even more time to create a plan than the `FFTW_MEASURE` flag. The accumulated wisdom can be saved to disk and later recalled when your program has to run again. The recalled wisdom will be applied to any plan creation with a flag that is less demanding than the flag for which the wisdom was created. Thus wisdom created with the `FFTW_MEASURE` flag, will accelerate the creation of a plan with the `FFTW_ESTIME` flag and `FFTW_MEASURE` flag. On my PC, I created 12 plans (with no prior wisdom available) for FFT sizes from 128 to 1536 in about 400 milliseconds. The second time around (when wisdom is available), creating those same plans takes only 600 microseconds for both the `FFTW_MEASURE` flag and `FFTW_ESTIME` flag. Wisdom is stored as a character string inside the FFTW library. Below is an example.

```
(fftw-3.1.2 fftwf_wisdom
  (fftwf_codelet_t2bv_64 0 #x11bdd #x11bdd #x0 #x072e38d7 #x6687741c #x4236c0ca #x50b2b9e9)
  (fftwf_codelet_t2fv_4 0 #x11bdd #x11bdd #x0 #xf656452b #x1e0a4e1a #xc3201626 #xb430ea0d)
  (fftwf_codelet_n2bv_16 0 #x11bdd #x11bdd #x0 #x182c16f4 #x80e80192 #x06497413 #xe8f9accb)
  (fftwf_codelet_n2bv_12 0 #x11bdd #x11bdd #x0 #x92be533b #x779112b3 #xb0d18403 #x1b9944cf)
  (fftwf_codelet_n2fv_32 0 #x11bdd #x11bdd #x0 #x6bde14e1 #x51d50f72 #xdf6182e6 #x6fffed34)
  (fftwf_codelet_n2fv_32 0 #x11bdd #x11bdd #x0 #x2bf92fc2 #xed5c4aea #xa89a0f28 #x8584f03b)
  (fftwf_codelet_t2fv_32 0 #x11bdd #x11bdd #x0 #xaa11d6a4 #x7c52e57c #x2fc13425 #x60095fd3)
  (fftwf_codelet_n2bv_64 0 #x11bdd #x11bdd #x0 #xd690ab21 #x20023238 #xc4632431 #x63d1973c)
  (fftwf_codelet_n2bv_32 0 #x11bdd #x11bdd #x0 #x4f06074e #x5e8c024b #x24a07c44 #x9228d94b)
  (fftwf_codelet_t2bv_4 0 #x11bdd #x11bdd #x0 #x8a2f4f6f #xf1259a8c #x4f6913c8 #xd0b630ce)
  (fftwf_codelet_n2fv_32 0 #x11bdd #x11bdd #x0 #x70125cce #xee8dcce8 #x19b847d1 #x52d52df6)
  (fftwf_codelet_n2fv_64 0 #x11bdd #x11bdd #x0 #xe0c026da #x7fd480d2 #x033a79db #xf81f3304)
  (fftwf_codelet_t2bv_4 0 #x11bdd #x11bdd #x0 #xe7288594 #xa53b1379 #xc1ea7b45 #xe2150d3e)
  (fftwf_codelet_t2bv_64 0 #x11bdd #x11bdd #x0 #xb8964bc9 #x77cb40ab #xa22d9de7 #x24274f6d)
  (fftwf_codelet_t2bv_16 0 #x11bdd #x11bdd #x0 #xac79e5c1 #xf99ecd88 #xcff2c8e0 #x6bcf98ed)
  (fftwf_codelet_t2fv_16 0 #x11bdd #x11bdd #x0 #x3fc2bea4 #x586431be #x98565f1d #x4fb459f6)
  (fftwf_codelet_t2fv_4 0 #x11bdd #x11bdd #x0 #xcf8bb9e3 #xbf1d7cdc #xe43bcc43 #xfa9b719f)
  (fftwf_codelet_n2bv_32 0 #x11bdd #x11bdd #x0 #x6a4dfdef #x59ef7143 #xac30ce37 #x394d99bb)
  (fftwf_codelet_t2fv_64 0 #x11bdd #x11bdd #x0 #xc5a4c21b #xfe9cbd08 #xc19bdb36 #x8e83aba3)
  (fftwf_codelet_n2fv_12 0 #x11bdd #x11bdd #x0 #xcbeec26d #xd23db9a6 #xb284c093 #xcbe58a02)
)
```

Saving Wisdom

The FFTW library provides a host of methods that allow us to save off wisdom information to a file, a char string or some other output. At the heart of this methodology we find the following method, which can be reviewed in section 4.7.1 of the PDF.

```
void fftw_export_wisdom(void (*write_char)(char c, void*), void *data)
```

This rather sophisticated function allows you to pass to it a function pointer of a callback function that returns void and takes a character and void pointer as its inputs. Additionally, you may supply a void pointer to the fftw_export_wisdom function. Let's take a quick look at a callback function with the right function signature.

```
void _cdecl WriteChar(char c, void *In)
{
   // The void* in is a convenient feature that allows us to pass a pointer to the container
   // that will collect each character.
   std::string* pString = (std::string*)In;
   pString->append(&c);
}
```

The second argument, void *data, of the fftw_export_wisdom() function, will be passed on to the callback function and arrives there as the second argument, void *In. Take a look at the code below and assume that the wisdom is composed of 2000 characters. In the fftw_export_wisdom(), we pass the callback function as well as the address of a std::string object. Once fftw_export_wisdom() executes, the WriteChar() function is now called 2000 times, each time a new character is provided along with the pointer to the str:string object. The WriteChar() function receives each character and adds it to the str::string object until we are done.

```
std::string  strWisdom;
// Make some space in the string so we don't have to reallocate the string.
strWisdom.reserve(10000);

// This function must be provided with a callback function pointer. For convenience it allows
// us to pass a void*, which I use to pass the address to the std::string WisdomString, which
// will store each widsom string character. See WriteChar() at the start of this page. One we
// enter the fftw_export_wisdom(function), the WriteChar function will be called N times, where
// N is the length  of the wisdom string in character. Only then do we proceed.
fftw_export_wisdom(WriteChar, &strWisdom);

word wStrLength = (word)strWisdom.size();

err = fopen_s(&hFile, WISDOM_FILE, "wb");
bool        bFileOpen = err == 0;
if (bFileOpen)
{
   // Writing wisdom to a binary file
   fwrite(&wStrLength,   sizeof(word), 1, hFile);
   fwrite(strWisdom.c_str(), sizeof(char), wStrLength, hFile);
   fclose(hFile);
}
```

We now find the length of the str::string object giving us the length of the wisdom character string. We open a binary file, write the number of wisdom characters as a word, followed by all wisdom characters.

→ I mentioned earlier that the FFTW library has wrapper functions that make it easier to save wisdom character to a file. Why not use those? At the moment these simpler wrapper function somehow misbehave and I was forced to use this more complicated scheme.

→ Notice also the calling convention `_cdecl` preceeding the callback function declaration. There are two popular calling conventions: `_cdecl and _stdcall.` The `__stdcall` calling convention is mostly used for callback functions that are called by the windows operating system. Most normal C/C++ libraries use the `_cdecl` calling convention and we want to make sure here that WriteChar callback function receives its input parameters in the proper way.

Recalling Wisdom

To recall the wisdom, we open the wisdom file and if it exists, we start to read from it as follows. The goal is to read the wisdom characters into a character string and call fftw_import_wisdom_from_string() to convey the wisdom to the FFTW library. With this wisdom in hand very efficient plans are computed very fast.

```
word    wStrLength            = 0;
int     iWisdomInputSucceeded = 0;

FILE*       hFile;
errno_t     err = fopen_s(&hFile, WISDOM_FILE, "rb");
bool        bFileOpen = err == 0;
if (bFileOpen)
{
   // The file format for the WISDOM_FILE is as follows:
   // 1. A single word indicates the number of wisdom characters that follow.
   // 2. The remaining characters are the wisdom string.
```

```
    fread(&wStrLength, sizeof(wStrLength), 1, hFile); // Read Num of characters in wisdom string
    char* pWisdomString = new char[wStrLength];
    fread(pWisdomString, sizeof(char), wStrLength, hFile); // Read in the wisdom string
    fclose(hFile);
    std::cout << pWisdomString;
    // the fftw_import_wisdom_from_string() function returns non-zero upon success
    iWisdomInputSucceeded = fftw_import_wisdom_from_string(pWisdomString);
    delete[] pWisdomString;
}
```

Thread Safety

Most FFTW functions are not thread safe. Therefore, if you have FFTW functions simultaneously executing in separate threads, then corruption can occur. We need to protect the relevant FFTW function using a static mutex.

The CFft Class Header File

The following CFft class uses the single precision function variants, which will have slightly different names than what we used above. The SIMD instructions usually execute more float instructions than double instructions, making the single precision I/DFT calculations faster.

```
#pragma once

#include "fftw3.h"
#include <vector>   // The data for which we wish to take the DFT will be
#include <complex>  // organized as std::vector<std::complex<double>>
#include <mutex>    // The mutex protects certain fftw functions from simultaneous access by
#include <array>    // multiple threads
#include <vector>
#include <map>

using word        = unsigned short;
using dword       = unsigned long;
using PlanMap     = std::map<int, fftwf_plan>;
using PlanMapIter = PlanMap::iterator;

// Use this import library. Make sure "fftw3.h", "libfftw3f-f.lib", and libfftw3f-f.dll" are in
// the root directory.
#pragma comment(lib, "libfftw3f-3.lib")

// We simple define the wisdom file name here
#define WISDOM_FILE "LteNbIot.wisdom"

// Enum for going forward DFT and backward IDFT
enum class eFft_Direction : int
{
   Forward  = FFTW_FORWARD,
   Backward = FFTW_BACKWARD
};


class CFft
{
public:
   CFft();
   ~CFft();

   const word      GetNumFftSizes() const { return (word)sm_aFftSizes.size(); }
                                          // (word) as .size() returns type size_t

   const word      GetMaxFftSize()  const { return *(sm_aFftSizes.end() - 1); }
            // Subtract a 1 to get an iterator to the last entry in the array. We presuppose
            // that the last size is the largest.
```

```
//! \brief      GetPlanFromMap fetches one of the plans that is stored in m_MapOfFftPlans
//!
//! \param      wFftSize        I  It can only be one of the sizes provided in sm_aFftSizes
//! \param      iFftDirection   I  Can only be FFTW_FORWARD = -1  or   FFTW_BACKWARD  = +1
fftwf_plan*                          GetPlanFromMap
(    word                              wFftSize
   , eFft_Direction                    eFftDirection
);

//! \brief      Execute I/FFT given a length and direction. The typename T (float or double)
//! \note       The 'typename' keyword is needed ahead of the some of the types. This is
//!             often the case when you have nested types that depend on a template
//!             parameter such as T. The template allows us to pass complex vectors of type
//!             double or float.
//!
//! \param      iFftDirection    I       The direction of the FFT -> FFTW_FORWARD = -1 or
//!                                      FFTW_BACKWARD  = +1
//! \param      wFftSize         I       Self Explanatory
//! \param      vComplexInput    I       A vector of complex input samples
//! \param      iterInputVector  I An iterator indicating the first sample to be transformed
//! \param      vComplexOutput   O       A vector of complex output samples
//! \param      iterOutputVector I      An iterator to the first sample where the function
//!                                     will place the output
template <typename T>
bool                                 RunFFTW
(    eFft_Direction                                eFftDirection
   , word                                          wFftSize
   , std::vector<std::complex<T>>&                 rvInput
   , typename std::vector<std::complex<T>>::iterator itStartEntryInput
   , std::vector<std::complex<T>>&                 rvOutput
   , typename std::vector<std::complex<T>>::iterator itStartEntryOutput
)
{
   // The following steps when running the FFT
   // 1. Grab a pointer to the plan that we need
   // 2. Ensure that we have enough input and output samples to work with
   // 3. Copying samples into input vector.
   // 4. Execute the plan
   // 5. Copy the FFT output samples into the output vector.

   // ------------------------------------------------------------------
   // 1. Fetch a pointer to the plan that we want and ensure that calls success
   fftwf_plan* pFftw_Plan = GetPlanFromMap(wFftSize
                                         , eFftDirection);

   // Check that we succeeded in retrieving the plan
   if (pFftw_Plan == nullptr) return false;

   // ---------------------------------------------------------------------------
   // 2. As we allow different sections of the input vector to be Fourier transformed, we
   //    must ensure that the number of entries after the iterator>= m_dwFftSize. The caller
   //    should zero pad the input buffer such that enough entries exist. The same goes for
   //    the output vector, which must feature a sufficient number of entries to absorb the
   //    FFT output values.
   dword dwAvailableInputValues  = rvInput.end()  - itStartEntryInput;
   dword dwAvailableOutputValues = rvOutput.end() - itStartEntryOutput;
   if (dwAvailableInputValues  < wFftSize) return false;
   if (dwAvailableOutputValues < wFftSize) return false;

   // ---------------------------------------------------------------------------------------
   // 3. Copying vector information into fftwf input buffer
   m_sMutex.lock();
   for (dword dwIndex = 0; dwIndex < wFftSize; dwIndex++, itStartEntryInput++)
   {
      m_cInputBuffer[dwIndex][0] = (float)(*itStartEntryInput).real();
      m_cInputBuffer[dwIndex][1] = (float)(*itStartEntryInput).imag();
   }
   m_sMutex.unlock();
```

```
        // ---------------------------------------------------------------------------------------
        // 4. Compute the fftw algorithm, whether forward or backward
        fftwf_execute(*pFftw_Plan);

        // ---------------------------------------------------------------------------------------
        // 5. Copying fftw output buffer into output vector.
        //    fftw_execute does not scale it's results. In this code, we will divide the FORWARD
        //    FFT results by m_dwFftSize. The results of the BACHWARD FFT remain unscaled
        double dScale = 1.0;
        if (eFftDirection == eFft_Direction::Forward) dScale = (double)wFftSize;

        m_sMutex.lock();
        for (dword dwIndex = 0; dwIndex < wFftSize; dwIndex++, itStartEntryOutput++)
        {
            *itStartEntryOutput =  std::complex<T>((T)(m_cOutputBuffer[dwIndex][0]/dScale),
                                                    (T) (m_cOutputBuffer[dwIndex][1] / dScale));
        }
        m_sMutex.unlock();
        return true;
    }


    // The constexpr indicates that the expression should be evaluated at compile time.
    // Static constexpr actually guarantees that it will be evaluated at compile time.
    // The following are the FFT sizes we support.
    static constexpr std::array<word, 5> sm_aFftSizes = { 128, 256, 512, 768, 1024 };

private:
    // We use a static mutex such that only one CFft instance from one thread can access the
    // FFTW features.
    static  std::mutex                  m_sMutex;

    // The following is a map containing plans. The key to this map will be the size of the
    // associated FFT multiplied by the FFT direction. FFTW_FORWARD = -1 and FFTW_BACKWARD = +1.
    PlanMap                             m_PlanMap;

    // The fftwf_complex type is a float[2]. Once you create an array of type fftwf_complex
    // you can access the real component via Buffer[i][1] and the imaginary component via
    // Buffer[i][2].
    fftwf_complex*                     m_cInputBuffer;
    fftwf_complex*                     m_cOutputBuffer;
};
```

__The CFft Class Source File__

```
#include "Fft.h"

std::mutex                 CFft::m_sMutex;

// This is a callback function, that is called muliple times by fftw_export_wisdom(). The
// fftw_export_wisdom() provides one wisdom character per call to the WriteChar callback.
// The _cdecl calling convention is preferred over _stdcall (which is the calling convention
// of WINAPI calls). So if you are writing a callback function that is called by windows, then
// _stdcall is likely the way to go. Calls from C/C++ libraries, will likely use _cdecl.
void _cdecl WriteChar(char c, void *In)
{
    // The void* in is a convenient feature that allows us to pass a pointer to the container
    // that will collect each character.
    std::string* pString = (std::string*)In;
    pString->append(&c);
}
```

```
// ---------------------------------------------------------------------------
// The CFft CTOR
// ---------------------------------------------------------------------------
CFft::CFft()
{
   // ---------------------------------------------------------------
   // 1. Protect wisdom lookup and plan creation from interference due to other threads.
   m_sMutex.lock();

   // --------------------------------------------------------------
   // 2. Check to see whether a wisdom file exists
   word    wStrLength            = 0;
   int     iWisdomInputSucceeded = 0;

   FILE*       hFile;
   errno_t     err = fopen_s(&hFile, WISDOM_FILE, "rb");
   bool        bFileOpen = err == 0;
   if (bFileOpen)
   {
      // The file format for the WISDOM_FILE is as follows:
      // 1. A single word indicates the number of wisdom characters that follow.
      // 2. The remaining characters are the wisdom string.
      fread(&wStrLength, sizeof(wStrLength), 1, hFile);  // Read num char in the wisdom string
      char* pWisdomString = new char[wStrLength];
      fread(pWisdomString, sizeof(char), wStrLength, hFile); // Read in the wisdom string
      fclose(hFile);
      // std::cout << pWisdomString;
      // the fftw_import_wisdom_from_string() function returns non-zero upon success
      iWisdomInputSucceeded = fftwf_import_wisdom_from_string(pWisdomString);
      delete[] pWisdomString;
   }


   // ---------------------------------------------------------------------------
   // 3. The input and output buffers are simply set to the maximum available FFT size.
   //    These inputs will be used for all FFTs.
   m_cInputBuffer  = (fftwf_complex*)fftwf_malloc(sizeof(fftwf_complex) * GetMaxFftSize());
   m_cOutputBuffer = (fftwf_complex*)fftwf_malloc(sizeof(fftwf_complex) * GetMaxFftSize());

   // ---------------------------------------------------------------------------
   // 4. We will use the famous emplace function to insert pairs into the map of FFT plans.
   //    We will literally create the FFT plan during the emplace function. I believe the
   //    fft plan is a pointer. Notice that I only use FFTW_MEASURE. When the wisdom is
   //    available, the plan will use it to equally accelerate the plan recreation, whether
   //    we use FFTW_MEASURE or FFTW_ESTIMATE. On my machine, wisdom creation takes about
   //    400 milliseconds. Plan creation with available wisdom about 600 microseconds
   //    whether FFTW_MEASURE or FFTW_ESTIMATE is specified.
   for (word wFftSize : sm_aFftSizes)
   {
      // Build and insert the FFTW plan for the forward direction
      size_t                MapKey = (int)wFftSize * FFTW_FORWARD;
      m_PlanMap.emplace(MapKey, fftwf_plan_dft_1d(wFftSize
                                               , m_cInputBuffer
                                               , m_cOutputBuffer
                                               , FFTW_FORWARD
                                               , FFTW_MEASURE));

      // Build and insert the FFTW plan for the backward direction
      MapKey = (int)wFftSize * FFTW_BACKWARD;
      m_PlanMap.emplace(MapKey, fftwf_plan_dft_1d(wFftSize
                                               , m_cInputBuffer
                                               , m_cOutputBuffer
                                               , FFTW_BACKWARD
                                               , FFTW_MEASURE));
   }
```

```
   // ----------------------------------------------------------------------------
   // 5. In case we were unable to read the wisdom from the file, new wisdom was created
   //  during plan creation in the last loop. We will now write the wisdom string to the file.
   //  We use a rather low level fftw function called fftw_export_wisdom() to extract the
   //  wisdom string. The FFTW library provides some wrapper functions that use
   //  fftwf_export_wisdom() internally, but I have had nothing but problems using these
   //  wrapper functions. Thus, similarly to other implementations I will create a callback
   //  function to interact with fftwf_export_wisdom() and extract the wisdom string.
   if (iWisdomInputSucceeded == 0)
   {
      std::string  strWisdom;
      // Make some space in the string so we don't have to reallocate the string constantly.
      strWisdom.reserve(2500);

      // This function must be provided with a callback function pointer. For convenience it
      // allows us to pass a void*, which I use to pass the address to the std::string
      // WisdomString, which will store each widsom string character. See WriteChar() at the
      // start of this page. Once we enter the fftwf_export_wisdom(function), the WriteChar
      // function will be called N times, where N is the length of the wisdom string in
      // character. Only then do we proceed.
      fftwf_export_wisdom(WriteChar, &strWisdom);

      word wStrLength = (word)strWisdom.size();

      err = fopen_s(&hFile, WISDOM_FILE, "wb");
      bool        bFileOpen = err == 0;
      if (bFileOpen)
      {
         // Writing wisdom to a binary file
         fwrite(&wStrLength,   sizeof(word), 1, hFile);
         fwrite(strWisdom.c_str(), sizeof(char), wStrLength, hFile);
         fclose(hFile);
      }
   }

   // ----------------------------------------------------------------------------
   // 6. Unlock the mutex so that other threads can run this code.
   m_sMutex.unlock();

}


// -------------------------------------------------------------------------------
// The CFft DTOR
// -------------------------------------------------------------------------------
CFft::~CFft()
{
   // Protecting plan destruction from interference of other threads
   m_sMutex.lock();

   for (auto Pair : m_PlanMap)
   {
      // Destroy all plans, as they are sitting on the heap.
      fftwf_destroy_plan(Pair.second);
   }

   // Deallocate the input and output vectors
   fftwf_free(m_cInputBuffer);                          // Deallocate the input buffer
   fftwf_free(m_cOutputBuffer);                         // Deallocate the output buffer

   // Unlock the mutex
   m_sMutex.unlock();
}
```

```
// ------------------------------------------------------------------------------------------
//  CLocalFft::GetPlanFromMap
// ------------------------------------------------------------------------------------------

fftwf_plan*                              CFft::GetPlanFromMap
(    word                                  wFftSize
   , eFft_Direction                        eFftDirection
)
{
   // Here we create the access key to the map. The Map key is the size of the FFT multiplied
   // by the direction
   _ASSERT_EXPR(eFftDirection == eFft_Direction::Forward ||
   eFftDirection == eFft_Direction::Backward, L"The direction of FFT direction is incorrect.");

   size_t                 MapKey = (int)wFftSize * (int)eFftDirection;
   PlanMapIter            Iterator = m_PlanMap.find(MapKey);

   // Did we find the plan???
   if (Iterator == m_PlanMap.end())
   {
      _ASSERT_EXPR(false, L"No FFTW plan was retrieved.");
      return nullptr;
   }
   else
   {
      // First get the pair by dereferencing the iterator and then get the plan by requesting
      // member variable second.
      return &((*Iterator).second);
   }
}
```

The Test Bench

Use the following code to exercise the class.

```
#include <iostream>
#include <complex>
#include <vector>
#include "Fft.h"

// Select either double or float. The I/DFT calculation itself will always use single precision
// but the input signal may be of either type.
using FloatType = double;
using Complex   = std::complex<FloatType>;

int main()
{
   // Construct a CFft object
   CFft myFft;
   // Get the first FFT size that is supported
   word wMinFftSize = CFft::sm_aFftSizes[0];

   // Instantiate the input vector holding the input signal. CFft will copy this
   // vector into its input buffer.
   std::vector<Complex>   vInput(wMinFftSize, Complex(0,0));
   // Instantiate the output vector. CFft will copy its output buffer into this vector.
   std::vector<Complex>   vOutput(wMinFftSize, Complex(0,0));

   // Place some non-zer content into the vector holding the intput data.
   vInput[0] = Complex(1,0);
   vInput[127] = Complex(0, -5);

   // Run the DFT
   myFft.RunFFTW(eFft_Direction::Forward
               , wMinFftSize
               , vInput
               , vInput.begin()
               , vOutput
               , vOutput.begin());

   // Run the IDFT of the DFT output to get back to the original vInput values.
   myFft.RunFFTW(eFft_Direction::Backward
               , wMinFftSize
               , vOutput
               , vOutput.begin()
               , vInput
               , vInput.begin());

   // Place breakpoint below. You will see the vInput is virtually identical to the
   // original vInput vector used at the beginning.
   int BreakHereAndCheckvOutput = 0;
 }
```