# NSFFT Reference Manual

## Introduction

This is a library for performing 1-dimensional discrete Fourier transforms. NSDFT is a simple, small and portable library, and it is efficient since it can utilize SIMD instruction sets in modern processors. It performs multiple transforms simultaneously, and thus it is especially suitable for digital signal processing. It does not need so much computation to make a good execution plan. This library is in public domain, so that you can incorporate this library into your product without any obligation.

## API Reference

In this section, the API functions are explained.

### Include files

You have to include two include files in dft directory.

```
#include <stdint.h>
#include "SIMDBase.h"
#include "DFT.h"
```

### Data types

First, you have to choose a data type to represent an element in the input and output sequence of numbers. You can choose from the following three types.

| Symbol | Data Type |
|---|---|
| SIMDBase_TYPE_FLOAT | float type in C language |
| SIMDBase_TYPE_DOUBLE | double type in C language |
| SIMDBase_TYPE_LONGDOUBLE | long double type in C language |

Table 1 Data types

### Computation modes

Next, a compuation mode have to be chosen. You can choose from the following modes.

| Symbol | Type | Vector Length | Computation Mode |
|---|---|---|---|
| SIMDBase_MODE_PUREC_FLOAT | float | 1 | Scalar float |
| SIMDBase_MODE_PUREC_DOUBLE | double | 1 | Scalar double |
| SIMDBase_MODE_PUREC_LONGDOUBLE | long double | 1 | Scalar long double |
| SIMDBase_MODE_SSE_FLOAT | float | 4 | x86 SSE |
| SIMDBase_MODE_SSE2_DOUBLE | double | 2 | x86 SSE2 |
| SIMDBase_MODE_NEON_FLOAT | float | 4 | ARM NEON |
| SIMDBase_MODE_AVX_FLOAT | float | 8 | x86 AVX (float) |
| SIMDBase_MODE_AVX_DOUBLE | double | 4 | x86 AVX (double) |
| SIMDBase_MODE_ALTIVEC_FLOAT | float | 4 | PowerPC Altivec |

Table 2 Computation modes

The following function automatically checks the availability of each instruction set on your computer, and chooses the best computation mode.

```
int32_t SIMDBase_chooseBestMode(int32_t type);
```

The return value is the best mode chosen by this routine. *type* is the data type you chose.

#### Retrieving parameters

You can make queries for any mode using the following function.

```
int32_t SIMDBase_getModeParamInt(int32_t paramId, int32_t mode);
```

*mode* is the computation mode you chose. *paramId* is one of the following.

| Symbol | Meaning |
|---|---|
| SIMDBase_PARAMID_SIZE_OF_REAL | Size of an element in a vector in byte |
| SIMDBase_PARAMID_SIZE_OF_VECT | Size of the vector in byte |
| SIMDBase_PARAMID_VECTOR_LEN | Number of elements in a vector |
| SIMDBase_PARAMID_MODE_AVAILABILITY | Whether the given mode is available or not |

Table 3 Querying parameter for computation mode

Here, a vector is a set of multiple primitive data element (single or double precision FP number) which can be stored in one SIMD register, and can be processed by one SIMD instruction at the same time.

You can get the mode name in string data type. In this case, *paramId* must be SIMDBase_PARAMID_MODE_NAME.

```
char *SIMDBase_getModeParamString(int32_t paramId, int32_t mode);
```

You should not modify the data returned by the above function.

#### Making and destroying execution plan

An execution plan can be made by the following function.

```
DFT *DFT_init(int32_t mode, int32_t n, int32_t flags);
```

The return value is a pointer to a newly made plan. *mode* is the mode you chose above. *n* is the length of a transform. You can specify a bitwise OR of the following symbols as *flags*. You should not specify more than one flags regarding to test run. You should not specify DFT_FLAG_FORCE_RECURSIVE and DFT_FLAG_FORCE_COBRA at the same time. If neither DFT_FLAG_REAL nor DFT_FLAG_ALT_REAL is specified, an execution plan for complex transforms are made.

| Symbol | Meaning |
|---|---|
| DFT_FLAG_NO_TEST_RUN | Make execution plan without performing a test run |
| DFT_FLAG_LIGHT_TEST_RUN | Perform small amount of test run to make an execution plan |
| DFT_FLAG_HEAVY_TEST_RUN | Perform large amount of test run to make an execution plan |
| DFT_FLAG_EXHAUSTIVE_TEST_RUN | Perform exhaustive search of parameters and find the optimal execution plan |
| DFT_FLAG_REAL | Make an execution plan for a real transform |
| DFT_FLAG_ALT_REAL | Make an execution plan for an alternative real transform |
| DFT_FLAG_VERBOSE | Make some noise during making an execution plan |
| DFT_FLAG_NOBITREVERSAL | Does not perforam bitreversal operation during a transform |
| DFT_FLAG_FORCE_RECURSIVE | Force using the recursive bit-reveral routine. This routine is suited for small transforms. |

| | |
|---|---|
| DFT_FLAG_FORCE_COBRA | Force using the Cobra bit-reveral routine. This routine is suited for large transforms. |

Table 4 Options for making execution plan

You can destroy the plan you made by the following function.

```
void DFT_dispose(DFT *p, int32_t mode);
```

*p* is a pointer to the execution plan. *mode* is the corresponding execution mode.

You can retrieve parameters of a plan using the following function.

```
int32_t DFT_getPlanParamInt(int32_t paramId, void *p);
```

*p* is a pointer to an execution plan. *paramId* is one of the following.

| Symbol | Meaning |
|---|---|
| DFT_PARAMID_TYPE | Data type |
| DFT_PARAMID_MODE | Computation mode |
| DFT_PARAMID_FFT_LENGTH | Length of the transform |
| DFT_PARAMID_IS_REAL_TRANSFORM | Whether the plan is for real transforms |
| DFT_PARAMID_NO_BIT_REVERSAL | Whether the plan does not perform bit reversal operation |
| DFT_PARAMID_TEST_RUN | How much test run is performed when making this plan |

Table 5 Querying parameter for execution plan

#### Writing and reading execution plan to/from file

You can write or read an execution plan to/from a file using the following functions.

```
int32_t DFT_fwrite(DFT *p, FILE *fp);
DFT *DFT_fread(FILE *fp, int32_t *errcode);
```

*p* is a pointer to a plan. *fp* is a file pointer. DFT_fwrite returns 1 if the plan is successfully written, and 0 if an error occurs. DFT_fread returns the pointer to the read plan if the plan is successfully read, and NULL if an error occurs. If an error occurs, an error code is returned to a variable whose pointer is specified by *errcode*. The interpretation of error codes is given below.

| Symbol | Meaning |
|---|---|
| DFT_ERROR_NOERROR | No error |
| DFT_ERROR_FILE_VERSION | File format version mismatch |
| DFT_ERROR_FILE_IO | I/O error |
| DFT_ERROR_UNEXPECTED_EOF | Unexpected EOF |
| DFT_ERROR_MODE_NOT_COMPILED_IN | Tried to read a plan with mode that is not compiled in |
| DFT_ERROR_MODE_NOT_AVAILABLE | Tried to read a plan with mode that is not supported by hardware |
| DFT_ERROR_UNKNOWN_MODE | Tried to read a plan with mode that is unknown by library |

Table 6 Errors that may happen during file I/O

#### Allocating and freeing buffers for transforms

In order to allocate word-aligned buffers for storing data which is fed to the FFT routine, you have to use the following function.

```
void *DFT_alignedMalloc(uint64_t size);
```

This function allocates *size* bytes of word-aligned memory and returns the pointer. In order to free this memory, you have to use the following function.

```
void DFT_alignedFree(void *ptr);
```

*ptr* is the pointer returned from DFT_alignedMalloc function.

#### Executing transform

By the following function, the planned transform can be executed.

```
void DFT_execute(DFT *p, int32_t mode, void *s, int32_t dir);
```

*p* is a pointer to the plan. *mode* is the computation mode. *s* is the pointer to the buffer in which the sequence of input values is stored. This pointer must be a pointer returned from DFT_alignedMalloc function. *dir* specifies the direction of transform.

The forward and backward discrete Fourier transforms are defined by the following formula (1) and (2), respectively.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N}kn} \qquad k = 0, \cdots, N-1 \tag{1}$$

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N}kn} \qquad n = 0, \cdots, N-1 \tag{2}$$

The complex forward and backward transforms perform the transforms defined by the following formula (3) and (4), respectively. *V* is the vector length mentioned above. Again, calling DFT_execute once performs *V* forward or backward transforms at a time. Please note that (4) gives values multiplied by *N* compared to (2). Specifying -1 as the direction of transform performs the transform defined by (3). In this case, the input should be given as in (5) , and the output is given as in (6). Specifying 1 as the direction of transform performs the transform defined by (4), and in this case, the input should be given as in (6) , and the output is given as in (5).

$$X_{k,v} = \sum_{n=0}^{N-1} x_{n,v} e^{-\frac{2\pi i}{N}kn} \qquad k = 0, \cdots, N-1\ ,\ \ v = 0, \cdots, V-1 \tag{3}$$

$$x_{n,v} = \sum_{k=0}^{N-1} X_{k,v} e^{\frac{2\pi i}{N}kn} \qquad n = 0, \cdots, N-1\ ,\ \ v = 0, \cdots, V-1 \tag{4}$$

$$\begin{cases} s[(2n+0)V+v] = \mathrm{Re}(x_{n,v}) \\ s[(2n+1)V+v] = \mathrm{Im}(x_{n,v}) \end{cases} \qquad n = 0, \cdots, N-1\ ,\ \ v = 0, \cdots, V-1 \tag{5}$$

$$\begin{cases} s[(2k+0)V+v] = \mathrm{Re}(X_{k,v}) \\ s[(2k+1)V+v] = \mathrm{Im}(X_{k,v}) \end{cases} \qquad k = 0, \cdots, N-1\ ,\ \ v = 0, \cdots, V-1 \tag{6}$$

The real forward transform performs the transform defined by (3) when the condition (7) is satisfied. In this case, the output satisfies (8). You should specify -1 as the direction of transform, and the input should be given as in (9), and the output is given as in (10). The real backward transform is the opposite of the real forward transform. The input should satisfy (8) and the output satisfies (7). You should specify 1 as the direction of transform, and the input should be given as in (10), and the output is given as in (11).

$$\mathrm{Im}(x_{n,v}) = 0 \qquad n = 0, \cdots, N-1\ ,\ \ v = 0, \cdots, V-1 \tag{7}$$

$$\begin{cases} X_{k,v} = X^*_{N-k,v} & k = 1, \cdots, \frac{N}{2} - 1 \\ \mathrm{Im}(X_{k,v}) = 0 & k = 0, \frac{N}{2} \end{cases} \quad v = 0, \cdots, V-1 \tag{8}$$

$$s[nV+v] = \mathrm{Re}(x_{n,v}) \quad n = 0, \cdots, N-1\ , \ \ v = 0, \cdots, V-1 \tag{9}$$

$$\begin{cases} s[(2k+0)V+v] = \mathrm{Re}(X_{k,v}) & k = 0, \cdots, \frac{N}{2} - 1 \\ s[V+v] = \mathrm{Re}(X_{N/2,v}) & \\ s[(2k+1)V+v] = \mathrm{Im}(X_{k,v}) & k = 1, \cdots, \frac{N}{2} - 1 \end{cases} \quad v = 0, \cdots, V-1 \tag{10}$$

$$2\,s[nV+v] = \mathrm{Re}(x_{n,v}) \quad n = 0, \cdots, N-1\ , \ \ v = 0, \cdots, V-1 \tag{11}$$

The alternative real transforms are defined by (12) to (16), similarly to the real transforms. The alternative transforms are handy if you are migrating from the FFT library made by Prof. Takuya Ooura. You should specify 1 as the direction in order to perform a forward transform, and -1 when you perform a backward transform.

$$\mathrm{Im}(X_{k,v}) = 0 \quad k = 0, \cdots, N-1\ , \ \ v = 0, \cdots, V-1 \tag{12}$$

$$\begin{cases} x_{n,v} = x^*_{N-n,v} & n = 1, \cdots, \frac{N}{2} - 1 \\ \mathrm{Im}(x_{n,v}) = 0 & n = 0, \frac{N}{2} \end{cases} \quad v = 0, \cdots, V-1 \tag{13}$$

$$s[nV+v] = \mathrm{Re}(X_{k,v}) \quad k = 0, \cdots, N-1\ , \ \ v = 0, \cdots, V-1 \tag{14}$$

$$\begin{cases} s[(2n+0)V+v] = \mathrm{Re}(x_{n,v}) & n = 0, \cdots, \frac{N}{2} - 1 \\ s[V+v] = \mathrm{Re}(x_{N/2,v}) & \\ s[(2n+1)V+v] = \mathrm{Im}(x_{n,v}) & n = 1, \cdots, \frac{N}{2} - 1 \end{cases} \quad v = 0, \cdots, V-1 \tag{15}$$

$$2\,s[nV+v] = \mathrm{Re}(X_{k,v}) \quad k = 0, \cdots, N-1\ , \ \ v = 0, \cdots, V-1 \tag{16}$$

## Examples

Below is an example code using nsfft library.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdint.h>
#include <complex.h>

#include "SIMDBase.h"
#include "DFT.h"

typedef float REAL;
#define TYPE SIMDBase_TYPE_FLOAT
```

```
#define THRES 1e-3

double complex omega(double n, double kn) {
  return cexp((-2 * M_PI * _Complex_I / n) * kn);
}

void forward(double complex *ts, double complex *fs, int len) {
  int k, n;

  for(k=0;k<len;k++) {
    fs[k] = 0;

    for(n=0;n<len;n++) {
      fs[k] += ts[n] * omega(len, n*k);
    }
  }
}

int main(int argc, char **argv) {
  const int n = 256;

  int mode = SIMDBase_chooseBestMode(TYPE);
  printf("mode : %d, %s\n", mode, SIMDBase_getModeParamString(SIMDBase_PARAMID_MODE_NAME, mode));

  int veclen = SIMDBase_getModeParamInt(SIMDBase_PARAMID_VECTOR_LEN, mode);
  int sizeOfVect = SIMDBase_getModeParamInt(SIMDBase_PARAMID_SIZE_OF_VECT, mode);

  //

  int i, j;

  DFT *p = DFT_init(mode, n, 0);
  REAL *sx = SIMDBase_alignedMalloc(sizeOfVect*n*2);

  //

  double complex ts[veclen][n], fs[veclen][n];

  for(j=0;j<veclen;j++) {
    for(i=0;i<n;i++) {
      ts[j][i] = (random() / (double)RAND_MAX) + (random() / (double)RAND_MAX) * _Complex_I;
      sx[(i*2+0)*veclen+j] = creal(ts[j][i]);
      sx[(i*2+1)*veclen+j] = cimag(ts[j][i]);
    }
  }

  //

  DFT_execute(p, mode, sx, -1);

  for(j=0;j<veclen;j++) {
    forward(ts[j], fs[j], n);
  }

  //

  int success = 1;

  for(j=0;j<veclen;j++) {
    for(i=0;i<n;i++) {
      if ((fabs(sx[(i*2+0)*veclen+j] - creal(fs[j][i])) > THRES) ||
          (fabs(sx[(i*2+1)*veclen+j] - cimag(fs[j][i])) > THRES)) {
        success = 0;
      }
    }
  }

  printf("%s\n", success ? "OK" : "NG");

  //

  SIMDBase_alignedFree(sx);
  DFT_dispose(p, mode);

  exit(0);
}
```

You should put this code under a directory in the root directory of the library, and then you can compile this code with the following command.

```
gcc -Wall -g -I ../simd -I ../dft -L../simd -L../dft -O DFTExample.c -lDFT -lSIMD -lm -o DFTExample
```

## Compilation

The nsfft source package include a few makefiles for various architectures. You should make symbolic links to makefiles suited for your computer under *dft* and *simd* directories.