
Subject: Re: 2D FFT Slow. Any ideas? fft2()
Posted by [h_chapman](#) on Sun, 14 Dec 2003 05:58:14 GMT
[View Forum Message](#) <> [Reply to Message](#)

"Richard G. French" <rfrench@wellesley.edu> wrote in message news:

> I've used FFTW from within IDL (using a DLM) in the past and it was a lot
> faster than the IDL FFT routine. I used a DLM wrapper set up by our old
> friend, S.V.H. Haugan. However, my current installation does not work since
> I have not updated it for IDL6.0. I'll see if I can get it working, and I'll
> report back on my findings.
>
> Dick French

I've been using FFTW for all my transform needs for quite some time, using a DLM based on probably the same post by S.V.H. Haugan. I updated it for fftw3 and works quite well under IDL 6.0 on Mac OS X. I include here the files `fftw.c`, `Makefile_fftw`, and `fftw.dlm`. Instructions for installing `fftw3` are in the `.c` file. It's been awhile since I worked on this, but there is still room for improvement - I can't remember how well the `/real` keyword works. But it does have an number of the nice new features from `fftw3`, such as the `patient` and `exhaustive` options (here the `/patient` and `/exhaustive` keywords) that you would run once in your life for a particular array size if you plan to do many FFTs of that size. I had an earlier version working on linux, I don't think there was much difference to the mac os x version.

A quick benchmark (on a dual 2GHz G5 under moderate load):

```
IDL> d=dcomplex(dist(1024))
IDL> fd=fftw(d,-1,/patient,nthreads=2)
% Loaded DLM: FFTW.
% FFTW: Imported wisdom from file.
IDL> t0=systime(1) & fd=fftw(d,-1,nthreads=2) & t1=systime(1)
IDL> print, t1-t0
    0.14081097
```

```
IDL> ;compare with single thread
IDL> t0=systime(1) & fd=fftw(d,-1) & t1=systime(1)
IDL> print, t1-t0
    0.18699801
```

```
IDL> ;compare with built-in FFT
IDL> t0=systime(1) & ft=fft(d) & t1=systime(1)
IDL> print, t1-t0
    1.0776391
```

Henry.

---- start of fftw.c ----

/* fftw.c

**

** Performs multi-dimensional complex FFT using the FFTW v3 libraries
** (Fastest Fourier Transform in the West) (double-precision version)
** to be linked in to IDL as a DLM. It is almost a direct replacement
** for FFT, and has the same syntax (except for /overwrite).
** The difference is the forward
** transform is not normalised and requires dividing by the number of
** elements.

**

** Written by Henry Chapman

**

** modified from an original source by S.V.H.Haugan, posted to
** comp.lang.idl-pvwave (for rfftw (v2) routines)

**

** To use this, first build the FFTW3 libraries:

** configure --enable-threads -enable-float -enable-altivec

** make

** make install

** configure -enable-float -enable-altivec

** make

** make install

** configure -enable-threads

** make

** make install

** configure

** make

** make install

**

** Next build the systemwide wisdom (see man fftw-wisdom)

** fftw-wisdom -v -c -o wisdom

** mv wisdom /etc/fftw/wisdom

** This may take a day to run!

**

** Then, build this routine with make -f Makefile_fftw

** and finally put fftw.so and fftw.dlm in a place in !DLM_PATH

**

**

** IDL Routine usage:

** result = fftw(f, dir, [/patient, /exhaustive, /overwrite, /real, \$

** nthreads=n)

**

** /patient keyword to use IDL_FFTW_PATIENT

```

** /exhaustive keyword to use IDL_FFTW_EXHAUSTIVE (has precedence over
patient)
** /overwrite keyword to do an in-place transform (complex only)
** /real keyword to do real-to-complex transform (the default is to
convert to
**   complex first). It is not a good idea to use both /real and
/overwrite
** nthread keyword to set number of threads
**
** When making a plan, the input array sometimes gets overwritten with
** zeroes.
** With /real, the transform of real data is returned back as an array
of
** (just over) half the size.
*/

```

```

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "idl_export.h"
#include <fftw3.h>

```

```

FILE *fftw_wisdom_file(char *mode)
{
    char *name;
    FILE *f;

    name=getenv("IDL_FFTW_WISDOM");

    if (name == (char*)NULL) {
        name = calloc(1024,sizeof(char));
        if (name == (char*)NULL) return (FILE*) NULL;

        strncpy(name,getenv("HOME"),1023);
        strncat(name,"/.idl_fftw3_wisdom.",1024-strlen(name));
        gethostname(name+strlen(name),1024-strlen(name));
    }
    /* IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO, name); */
    f = fopen(name, mode);
    free(name);
    return f;
}

```

```

FILE *fftwf_wisdom_file(char *mode)
{
    char *name;
    FILE *f;

```

```

name=getenv("IDL_FFTWF_WISDOM");

if (name == (char*)NULL) {
    name = calloc(1024,sizeof(char));
    if (name == (char*)NULL) return (FILE*) NULL;

    strncpy(name,getenv("HOME"),1023);
    strncat(name, ".idl_fftw3f_wisdom.",1024-strlen(name));
    gethostname(name+strlen(name),1024-strlen(name));
}
/* IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO, name); */
f = fopen(name, mode);
free(name);
return f;
}

static char *wisdom=(char*)NULL;
static char *wisdomf=(char*)NULL;

void fftw_init(void)
{
    FILE *f;

    static int done=0;

    /* We only call once per IDL session */
    if (done) {
        if (fftw_import_wisdom_from_string(wisdom) != 1){
            IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Bad wisdom
data?");
        }
        return;
    }
    done = 1;

    /* Initialise threads */
    if (fftw_init_threads() == 0) {
        IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Can't initialise
threads.");
        return;
    }

    /* Load the wisdom */

    f = fftw_wisdom_file("r");
    if (f==(FILE*)NULL) {
        IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Can't read wisdom

```

```

file.");
    return;
}
if (fftw_import_wisdom_from_file(f) != 1) {
    IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Bad wisdom data?");
}
fclose(f);
IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Imported wisdom from
file.");

if (fftw_import_system_wisdom() == 1 ) {
    IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Imported system
wisdom.");
}

wisdom = fftw_export_wisdom_to_string();
}

void fftwf_init(void)
{
    FILE *f;

    static int done=0;

    /* We only call once per IDL session */
    if (done) {
        if (fftwf_import_wisdom_from_string(wisdomf) != 1){
            IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Bad wisdom
data?");
        }
        return;
    }
    done = 1;

    /* Initialise threads */
    if (fftw_init_threads() == 0) {
        IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Can't initialise
threads.");
        return;
    }

    /* Load the wisdom */

    f = fftwf_wisdom_file("r");
    if (f==(FILE*)NULL) {
        IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Can't read wisdom
file.");
        return;
    }

```

```

}
if (fftwf_import_wisdom_from_file(f) != 1) {
    IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Bad wisdom data?");
}
fclose(f);
IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Imported wisdom from
file.");

if (fftwf_import_system_wisdom()) {
    IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Imported system
wisdom.");
}

wisdomf = fftwf_export_wisdom_to_string();
}

void fftw_finish(void)
{
    FILE *f;

    f = fftw_wisdom_file("w");
    fftw_export_wisdom_to_file(f);
    fclose(f);

    wisdom = fftw_export_wisdom_to_string();

/*  fftw_cleanup_threads(); */

}

void fftwf_finish(void)
{
    FILE *f;

    f = fftwf_wisdom_file("w");
    fftwf_export_wisdom_to_file(f);
    fclose(f);

    wisdomf = fftwf_export_wisdom_to_string();

/*  fftw_cleanup_threads(); */

}

void fftw_transform(IDL_VPTR in, IDL_VPTR out, int dir,
    int nthreads, unsigned flags)
/* Double-precision complex transform */

```

```

{
    fftw_complex *src, *Fsrc;
    fftw_plan plan;
    FILE *f;

    /* Make sure the dimension array is correct type */
    int i, dim[IDL_MAX_ARRAY_DIM];
    int ndim = in->value.arr->n_dim;
    for (i=0; i < in->value.arr->n_dim; i++)
        dim[i]=in->value.arr->dim[i];

    src = (fftw_complex*) in->value.arr->data;
    Fsrc = (fftw_complex*) out->value.arr->data;

    fftw_init();
    fftw_plan_with_nthreads(nthreads);
    plan = fftw_plan_dft(in->value.arr->n_dim, (const int *)dim,
        src, Fsrc, dir, flags);

    fftw_execute(plan);

    fftw_finish();
    fftw_destroy_plan(plan);
}

```

```

void fftwf_transform(IDL_VPTR in, IDL_VPTR out, int dir,
    int nthreads, unsigned flags)
    /* Single-precision complex transform */
{
    fftwf_complex *src, *Fsrc;
    fftwf_plan plan;
    FILE *f;

    /* Make sure the dimension array is correct type */
    int i, dim[IDL_MAX_ARRAY_DIM];
    for (i=0; i < in->value.arr->n_dim; i++)
        dim[i]=in->value.arr->dim[i];

    src = (fftwf_complex*) in->value.arr->data;
    Fsrc = (fftwf_complex*) out->value.arr->data;

    fftwf_init();
    fftwf_plan_with_nthreads(nthreads);
    plan = fftwf_plan_dft(in->value.arr->n_dim, (const int *)dim,
        src, Fsrc, dir, flags);

    fftwf_execute(plan);
}

```

```

    fftwf_finish();
    fftwf_destroy_plan(plan);
}

void fftwrf_transform(IDL_VPTR in, IDL_VPTR out,
    int nthreads, unsigned flags)
    /* Single-precision real transform (always forward) */
{
    float *src;
    fftwf_complex *Fsrc;
    fftwf_plan plan;
    FILE *f;

    /* Make sure the dimension array is correct type */
    int i, dim[IDL_MAX_ARRAY_DIM];
    for (i=0; i < in->value.arr->n_dim; i++)
        dim[i]=in->value.arr->dim[i];

    src = (float*) in->value.arr->data;
    Fsrc = (fftwf_complex*) out->value.arr->data;

    fftwf_init();
    fftw_plan_with_nthreads(nthreads);
    plan = fftwf_plan_dft_r2c(in->value.arr->n_dim, (const int *)dim,
        src, Fsrc, flags);

    fftwf_execute(plan);

    fftwf_finish();
    fftwf_destroy_plan(plan);
}

void fftwr_transform(IDL_VPTR in, IDL_VPTR out,
    int nthreads, unsigned flags)
    /* Double-precision real transform (always forward) */
{
    double *src;
    fftw_complex *Fsrc;
    fftw_plan plan;
    FILE *f;

    /* Make sure the dimension array is correct type */
    int i, dim[IDL_MAX_ARRAY_DIM];
    for (i=0; i < in->value.arr->n_dim; i++)
        dim[i]=in->value.arr->dim[i];

    src = (double*) in->value.arr->data;
    Fsrc = (fftw_complex*) out->value.arr->data;

```

```

fftw_init();
fftw_plan_with_nthreads(nthreads);
plan = fftw_plan_dft_r2c(in->value.arr->n_dim, (const int *)dim,
    src, Fsrc, flags);

fftw_execute(plan);

fftw_finish();
fftw_destroy_plan(plan);
}

IDL_VPTR FFTW(int argc, IDL_VPTR argv[], char argk[])
{
    typedef struct {
        IDL_KW_RESULT_FIRST_FIELD;
        IDL_LONG exhaustive;
        IDL_LONG overwrite;
        IDL_LONG patient;
        IDL_LONG real;
        IDL_LONG nthreads;
        int nthreads_there;
    } KW_RESULT;
    static IDL_KW_PAR kw_pars[] = {
        IDL_KW_FAST_SCAN,
        {"EXHAUSTIVE", IDL_TYP_LONG, 1, IDL_KW_ZERO|IDL_KW_VALUE|15, 0,
        IDL_KW_OFFSETOF(exhaustive)},
        {"NTHREADS", IDL_TYP_LONG, 1, 0, IDL_KW_OFFSETOF(nthreads_there),
        IDL_KW_OFFSETOF(nthreads) },
        {"OVERWRITE", IDL_TYP_LONG, 1, IDL_KW_ZERO|IDL_KW_VALUE|15, 0,
        IDL_KW_OFFSETOF(overwrite) },
        {"PATIENT", IDL_TYP_LONG, 1, IDL_KW_ZERO|IDL_KW_VALUE|15, 0,
        IDL_KW_OFFSETOF(patient) },
        {"REAL", IDL_TYP_LONG, 1, IDL_KW_ZERO|IDL_KW_VALUE|15, 0,
        IDL_KW_OFFSETOF(real) }, {NULL}
    };
    KW_RESULT kw;

    int i=0; /* Note it's use in indexing argv[i++] */
        /* This simplifies taking away extra arguments, */
        /* typically those specifying the number of elements */
        /* in input arrays (available as var->value.arr->n_elts) */
/* int i0=1; */
    unsigned fftw_flag=FFTW_MEASURE;

    (void) IDL_KWProcessByOffset(argc, argv, argk, kw_pars,
        (IDL_VPTR *) 0, 1, &kw);

```

```

if (kw.patient) fftw_flag = FFTW_PATIENT;
if (kw.exhaustive) fftw_flag = FFTW_EXHAUSTIVE;
if (!kw.nthreads_there) kw.nthreads = 1;

IDL_VPTR a=argv[i++]; /* Input array */
IDL_VPTR in_dir=argv[i++], /* Direction of transform */
call[1], /* For use when calling conversion routines */
tmp;

IDL_MEMINT dim[IDL_MAX_ARRAY_DIM], tdim[IDL_MAX_ARRAY_DIM], tmpi;
int dir, ndim;

/* TYPE CHECKING / ALLOCATION SECTION */

IDL_EXCLUDE_STRING(a);
IDL_ENSURE_ARRAY(a);
IDL_ENSURE_SIMPLE(a);

ndim = a->value.arr->n_dim; /* Shorthand */

if (argc!=2) {

    /* Default to forward direction */
    dir = FFTW_FORWARD;

} else {

    /* Make sure in_dir is an integer scalar */
    dir = IDL_LongScalar(in_dir);

    if (dir >= 0) dir = FFTW_BACKWARD;
    else dir = FFTW_FORWARD;

}

/* Swap dimensions to row major */
for (i=0; i<ndim/2; i++) {
    tmpi=a->value.arr->dim[i];
    a->value.arr->dim[i] = a->value.arr->dim[ndim-i-1];
    a->value.arr->dim[ndim-i-1] = tmpi;
}

/* Copy dimensions */
for (i=0; i<ndim; i++) {
    dim[i] = a->value.arr->dim[i];
    tdim[i] = a->value.arr->dim[i];
}

```

```

call[0] = a;

/* Convert real to complex, unless the keyword /real is used */
if ((a->type == IDL_TYP_FLOAT) && !kw.real) {
    a = IDL_CvtComplex(1, call, (void*)NULL);
} else if ((a->type == IDL_TYP_DOUBLE) && !kw.real) {
    a = IDL_CvtDComplex(1, call);
}

if (a->type == IDL_TYP_DCOMPLEX) {

    /* double complex */
    if (kw.overwrite) {
        fftw_transform(a, a, dir, kw.nthreads, fftw_flag);
    } else {
        IDL_MakeTempArray(IDL_TYP_DCOMPLEX, a->value.arr->n_dim, dim,
IDL_ARR_INI_NOP, &tmp);
        fftw_transform(a, tmp, dir, kw.nthreads, fftw_flag);
    }

} else if (a->type == IDL_TYP_COMPLEX) {

    /* single complex */
    if (kw.overwrite) {
        fftwf_transform(a, a, dir, kw.nthreads, fftw_flag);
    } else {
        IDL_MakeTempArray(IDL_TYP_COMPLEX, a->value.arr->n_dim, dim,
IDL_ARR_INI_NOP, &tmp);
        fftwf_transform(a, tmp, dir, kw.nthreads, fftw_flag);
    }

} else if (a->type == IDL_TYP_DOUBLE) {

    /* double float input -> 1/2 array dcomplex output */
    if (kw.overwrite) {
        a = IDL_CvtDComplex(1, call);
        fftwr_transform(a, a, kw.nthreads, fftw_flag);
    } else {
        tdim[a->value.arr->n_dim-1] = tdim[a->value.arr->n_dim-1]/2+1;
        IDL_MakeTempArray(IDL_TYP_DCOMPLEX, a->value.arr->n_dim, tdim,
IDL_ARR_INI_NOP, &tmp);
        fftwr_transform(a, tmp, kw.nthreads, fftw_flag);
    }

} else {

    /* single float input -> 1/2 array complex output */
    a = IDL_CvtFlt(1, call);
    if (kw.overwrite) {
        a = IDL_CvtComplex(1, call, (void*)NULL);
    }
}

```

```

    fftwrf_transform(a, a, kw.nthreads, fftw_flag);
} else {
    tdim[a->value.arr->n_dim-1] = tdim[a->value.arr->n_dim-1]/2+1;
    IDL_MakeTempArray(IDL_TYP_COMPLEX, a->value.arr->n_dim,tdim,
IDL_ARR_INI_NOP, &tmp);
    fftwrf_transform(a, tmp, kw.nthreads, fftw_flag);
}

/* fill out the other half of the array (of last dimension) */
/* i0=1; */
/* for (i=0; i<ndim-1; i++) i0=i0*dim[i]; */
/* i0=i0-1; */
/* for (i=0; i<dim[ndim-1]/2; i++) { */
/*     tmp->value.arr->data[i0] */
/* } */
}

/* Swap dimensions back to column major */
if (!kw.overwrite) {
    for (i=0; i<ndim; i++) tmp->value.arr->dim[i]=tdim[ndim-i-1];
}
for (i=0; i<ndim; i++) {
    a->value.arr->dim[i]=dim[ndim-i-1];
}

/* i=0; */
/* if (a != argv[i++]) IDL_DELTMP(a); */
if (in_dir != argv[1]) IDL_DELTMP(in_dir);

IDL_KW_FREE;

if (kw.overwrite) {
    return a;
} else {
    if (a != argv[0]) IDL_DELTMP(a);
    return tmp;
}
}

int IDL_Load(void)
{
    static IDL_SYSFUN_DEF2 func_def[] = {
        {FFTW, "FFTW", 1, 2, IDL_SYSFUN_DEF_F_KEYWORDS, 0}
    };
    return IDL_SysRtnAdd(func_def, TRUE, 1);
}
---- end of fftw.c ----

```

```

---- start of Makefile_fftw ----
# makefile for FFTW3 DLM
#
IDL_DIR = /usr/local/rsi/idl
IDL_INC_DIR = $(IDL_DIR)/external/include
IDL_LIB_DIR = $(IDL_DIR)/bin/bin.darwin.ppc
IDL_LIBS = -lidl -llanginfo

FFT_INC_DIR = /usr/local/include
FFT_LIB_DIR = /usr/local/lib
FFTW_INCLUDELIST = fftw3.h
FFTW_LIBS = -lfftw3 -lc -lm
# FFTW_THREADS_INCLUDELIST = fftw3_threads.h
FFTW_THREADS_LIBS = -lfftw3_threads
FFTW_LIBS = -lfftw3 -lfftw3f -lc -lm
THREAD_LIBS = -lpthread

CC = cc
CFLAGS = -D_REENTRANT
C_FLAGS = -fPIC -no-cpp-precomp -dynamic -fPIC -fno-common
-I$(IDL_INC_DIR) -I$(FFT_INC_DIR) -c $(CFLAGS)
LD = cc
SHELL = /bin/sh
X_LD_FLAGS = -bundle -flat_namespace -undefined suppress
-L$(FFT_LIB_DIR)
X_LD_POST = $(FFTW_THREADS_LIBS) $(FFTW_LIBS) $(THREAD_LIBS)
SO_EXT = so

.c.o :
$(CC) $(C_FLAGS) $(X_CFLAGS) *.c -o *.o

fftw : fftw.$(SO_EXT)
@date

fftw.$(SO_EXT) : fftw.o
$(LD) $(X_LD_FLAGS) -o fftw.$(SO_EXT) fftw.o $(X_LD_POST)

clean :
rm -f fftw.o fftw.so fftw.sl fftw.a \
so_locations

---- end of Makefile_fftw ----

---- start of fftw.dlm ----

```

MODULE FFTW
DESCRIPTION N-dimensional DComplex fftw (v.3)
VERSION \$Revision: 1.0 \$
BUILD_DATE \$Date: Fri Mar 28 22:15:45 PST 2003\$
SOURCE H. N. Chapman
FUNCTION FFTW 1 2 KEYWORDS

---- end of fftw.dlm ----
