Subject: n-point FFT
Posted by Jean Marc Delvit on Tue, 21 Nov 2000 08:00:00 GMT
View Forum Message <> Reply to Message

hi,

I wants to know how to do a n-point FFT with IDL
the same of FFT(X,n) in matlab (if the length of X is less than n, X is padded with trailing zeros to length n)

thanks

Jean-Marc

Subject: Re: n-point FFT
Posted by Richard G. French on Tue, 28 Nov 2000 02:09:24 GMT
View Forum Message <> Reply to Message

Has anyone tried to get an IDL interface working for the FFTW routine that is documented on http://www.fftw.org/? For those of you who don't know what it is, here is a brief description from their web page:

"FFTW is a C subroutine library for computing the Discrete Fourier Transform (DFT) in one or more dimensions, of both real and complex data, and of arbitrary input size. We believe that FFTW, which is free software, should become the FFT library of choice for most applications. Our benchmarks, performed on on a variety of platforms, show that FFTW's performance is typically superior to that of other publicly available FFT software. Moreover, FFTW's performance is portable: the program will perform well on most architectures without modification."

FFTW has lots of different methods of doing FFT's, and one of the novel features of the program is that it figures out which method is fastest for your particular problem by running a test case. Then, once it has selected the best method, it uses that for all of the runs you need to do. It is most useful when you have lots of transforms to do, and you want to have them all done as fast as possible.

I've seen an fftw.so version on a SUN Ultra-5 distribution, but I have not yet been able to get it working on my DEC (oops - Compaq) Alpha. Has anyone else tried to figure out how to get this working?

Thanks!
Dick French

Subject: Re: n-point FFT
Posted by Stein Vidar Hagfors H[1] on Tue, 28 Nov 2000 08:00:00 GMT
View Forum Message <> Reply to Message

"Richard G. French" <rfrench@wellesley.edu> writes:

> Has anyone tried to get an IDL interface working for the FFTW
> routine that is documented on http://www.fftw.org/? For those
> of you who don't know what it is, here is a brief description from their
> web page:

Yup, I even wrote a DLM wrapper for the real->complex->real transforms,
to be used like this (if I remember correctly):

IDL> n = (size(data))(1) ; To know the number of real pts
IDL> fdata = rfftwnd(data)
IDL> data1 = rfftwnd(fdata,n)

This was over a year ago, and I haven't used the routines since IDL
version ... something... so I cannot guarantee it'll work right away,
but it should be easy to fix any problems introduced in new IDL versions.
I've included the code below (and the .dlm file..)

When I tested it, it gave a speed increase of about 1 order of magnitude :-)
This was for the real transforms, though. An added bonus is that you're
cutting your memory usage by more than a factor of two.. :-)

One drawback is that you're restricted to float *or* double, unless you do
some fancy modifications involving library symbol hiding.. (never actually
tried this, but I think it's doable).

Stein Vidar Hagfors Haugan
ESA SOHO SOC/European Space Agency Science Operations Coordinator for SOHO

NASA Goddard Space Flight Center, Email: shaugan@esa.nascom.nasa.gov
Mail Code 682.3, Bld. 26, Room G-1, Tel.: 1-301-286-9028/240-354-6066
Greenbelt, Maryland 20771, USA. Fax: 1-301-286-0264

$Id: rfftwnd.dlm,v 1.1 1999/08/16 10:59:04 steinhh Exp steinhh $
MODULE RFFTWND
DESCRIPTION N-dimensional Real fftw
VERSION $Revision: 1.1 $

```
BUILD_DATE $Date: 1999/08/16 10:59:04 $
SOURCE S.V.H.HAUGAN
FUNCTION RFFTWND    1 2
 ------------------------------------------------------------ ---------
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "export.h"
#include "srfftw.h"

#define NULL_VPTR ((IDL_VPTR) NULL)

#define GETVARADDR(v) ((v->flags&IDL_V_ARR) ? (void*)v->value.arr->data \
    : (void*) &v->value.c)

#define GETVARDATA(v,n) ((v->flags&IDL_V_ARR) \
    ? (*(n) = v->value.arr->n_elts, v->value.arr->data) \
    : (*(n) = 1, & v->value.c ) )


FILE *rfftw_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_rfftw_wisdom.",1024-strlen(name));
    gethostname(name+strlen(name),1024-strlen(name));
  }
  f=fopen(name,mode);
  free(name);
  return f;
}

static char *wisdom=(char*)NULL;

void rfftw_init(void)
{
  FILE *f;

  static int done=0;
```

```
  if (done) return;

  done=1;
  f = rfftw_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)!=FFTW_SUCCESS) {
    IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Bad wisdom data?");
  }
  fclose(f);

  wisdom=fftw_export_wisdom_to_string();
  IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Imported wisdom from file.");
}


void rfftw_finish(void)
{
  FILE *f;
  static char *new_wisdom;

  new_wisdom=fftw_export_wisdom_to_string();
  if (new_wisdom==(char*)NULL) return;

  if (wisdom!=(char*)NULL && !strcmp(wisdom,new_wisdom)) {
    fftw_free(new_wisdom);
    return;
  }

  if (wisdom!=(char*)NULL) fftw_free(wisdom);

  wisdom=new_wisdom;

  f = rfftw_wisdom_file("w");
  if (f==(FILE*)NULL) {
    IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Couldn't write wisdom file");
    return;
  }
  fftw_export_wisdom_to_file(f);
  fclose(f);
  IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Wrote wisdom to file");
}

/* Forward is REAL -> COMPLEX, use input dimensions for plan */
void rfftwnd_forward(IDL_VPTR in, IDL_VPTR out)
```

```
{
  fftw_real *src;
  fftw_complex *Fsrc;
  rfftwnd_plan plan;

  rfftw_init();

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

  plan=rfftwnd_create_plan(in->value.arr->n_dim,
     (const int *)in->value.arr->dim,
     FFTW_REAL_TO_COMPLEX,FFTW_MEASURE|FFTW_USE_WISDOM);

  rfftwnd_one_real_to_complex(plan,src,Fsrc);
  rfftwnd_destroy_plan(plan);
  rfftw_finish();
}


/* Backward is COMPLEX -> REAL, use OUTPUT dimensions for plan! */
void rfftwnd_backward(IDL_VPTR in, IDL_VPTR out)
{
  fftw_complex *src;
  fftw_real *Fsrc;
  rfftwnd_plan plan;

  rfftw_init();

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

  plan=rfftwnd_create_plan(out->value.arr->n_dim,
     (const int*)out->value.arr->dim,
     FFTW_COMPLEX_TO_REAL,FFTW_MEASURE|FFTW_USE_WISDOM);

  rfftwnd_one_complex_to_real(plan,src,Fsrc);
  rfftwnd_destroy_plan(plan);
  rfftw_finish();
}


IDL_VPTR RFFTWND(int argc, IDL_VPTR argv[])
{
  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) */
```

```
IDL_VPTR a=argv[i++];    /* Input array */
IDL_VPTR odim=argv[i++], /* Output leading dim. for backward transform */

  call[1], /* For use when calling conversion routines */
  tmp;

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

char odimreq[]=
  "2nd arg (leading output dimension) required for backward transform";
char odimwrong[]=
  "2nd arg (leading output dimension) has an inappropriate value";

/* TYPE CHECKING / ALLOCATION SECTION */

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

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

/* If we're doing a forward transform, convert to Float,              */
/* if baclwards, convert to Complex, and make sure we have the leading */
/* dimension size, and convert it to long                             */

call[0] = a;
if (a->type!=IDL_TYP_COMPLEX && a->type!=IDL_TYP_DCOMPLEX) {

  dir = -1; /* We're doing a forward transform */
  a = IDL_CvtFlt(1,call); /* May cause a to be tmp */

} else {

  /* Require 2 arguments */
  if (argc!=2) IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_LONGJMP,odimreq);

  dir = 1;    /* Backward transform */
  a = IDL_CvtComplex(1,call); /* May cause a to be tmp */

  IDL_EXCLUDE_UNDEF(odim); /* Output leading dimension */
  IDL_ENSURE_SIMPLE(odim);
  IDL_EXCLUDE_STRING(odim);
  IDL_ENSURE_SCALAR(odim);

  call[0] = odim;
  odim = IDL_CvtLng(1,call);
```

```
  /* Check validity of the output leading dimension */
  tmpi = 2*a->value.arr->dim[0] - odim->value.l;
  if (tmpi != 1 && tmpi != 2)
    IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_LONGJMP,odimwrong);

  /* Give warning about overwritten input arg. */
  if (ndim>1 && !(a->flags&IDL_V_TEMP))
    IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Input overwritten!");
}

/* 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<a->value.arr->n_dim; i++) dim[i] = a->value.arr->dim[i];

/* Correct dimensions - for allocation */
if (dir==-1) dim[ndim-1] = 2*(dim[ndim-1]/2+1);
else         dim[ndim-1] = 2*dim[ndim-1];  /* Complex takes 2x FLOAT */

/* Storage for result */
IDL_MakeTempArray(IDL_TYP_FLOAT,a->value.arr->n_dim,dim,
    IDL_ARR_INI_INDEX,&tmp);

/* Correct output leading dimension - to make correct plan */
if (dir==1) {
  tmp->value.arr->n_elts /= tmp->value.arr->dim[ndim-1];
  tmp->value.arr->dim[ndim-1] = odim->value.l;
  tmp->value.arr->n_elts *= tmp->value.arr->dim[ndim-1];
}

if (dir==FFTW_REAL_TO_COMPLEX) rfftwnd_forward(a,tmp);
else                           rfftwnd_backward(a,tmp);

/* Swap dimensions back! */
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;
  tmpi=tmp->value.arr->dim[i];
  tmp->value.arr->dim[i] = tmp->value.arr->dim[ndim-i-1];
  tmp->value.arr->dim[ndim-i-1] = tmpi;
}
```

```
  i=0;

  if (a!=argv[i++]) IDL_DELTMP(a);
  if (odim!=argv[i++]) IDL_DELTMP(odim);

  if (dir==-1) {
    tmp->type = IDL_TYP_COMPLEX;  /* Change type, halve the size */
    tmp->value.arr->dim[0] /= 2;
    tmp->value.arr->n_elts /= 2;
  } else {
  }
  return tmp;
}

int IDL_Load(void)
{
  static IDL_SYSFUN_DEF2 func_def[] = {
    {RFFTWND,"RFFTWND",1,2, 0, 0}
    };
  return IDL_SysRtnAdd(func_def,TRUE,1);
}
```

Subject: Re: n-point FFT
Posted by Richard G. French on Wed, 06 Dec 2000 04:20:44 GMT
View Forum Message <> Reply to Message

Stein Vidar Hagfors Haugan wrote:
>
> "Richard G. French" <rfrench@wellesley.edu> writes:
>
>> Has anyone tried to get an IDL interface working for the FFTW
>> routine that is documented on http://www.fftw.org/?

> Yup, I even wrote a DLM wrapper for the real->complex->real transforms,
> to be used like this (if I remember correctly):

I'd like to thank Stein for his heroic help in getting
his DLM wrapper for multidimensional real->complex->real transforms
up and running under IDL5.4. Some redefinitions of calling sequences
for system routines (not all of them properly updated in the 5.4
External Development Guide) caused some problems that I could
not solve, but Stein tracked them down, and the revised version
of rfftwnd.c works like a charm. As he points out, it could be
optimized a bit further by keeping track of the fastest algorithm
to use for a particular transform (this is called the'wisdom' file
in FFTW parlance), but I have found it perfect for my applications.

I've done some crude speed tests of forward/inverse transform pairs
for RFFTWND() vs IDL's FFT() for
1-D and 2-D reals, both 2^N and 2^N+1 (to see how they handle arrays
that are not a power of 2).

For 1-D arrays that are 2^N long, RFFTWND() is faster than
IDL for arrays bigger than 2^15, and it flattens out at twice as fast
for arrays 2^17 and larger.

For 2-D arrays, I find a speed increase of a factor of between 2-10
(wow!)
using RFFTWND() compared to FFT()
for 2^8 x 2^8 (256 x 256) to 2^11 x 2^11 (2048x2048)arrays.

For non-powers of two, times for transforms vary depending on
the prime factors of the number of points, and comparisons between
RFFTWND() and FFT() are less monotonic, but in general are 2-3 times
faster for RFFTWND() than FFT().

For my particular application, involving lots of 2048x2048 FFT's
and their inverses, the execution time was 5 sec compared to 60 sec
which is a factor of >10 in speed - well worth the effort of
installing the DLM! I think some of the speed increase is due to
using less memory in the RFFTWND() array, since for the big arrays
I think my UNIX box may have been doing some swapping.

Here is Stein's revised code, tested on IDL5.3 and IDL5.4:
 ---------- ----------

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "export.h"
#include "obsolete.h"
#include "srfftw.h"

FILE *rfftw_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_rfftw_wisdom.",1024-strlen(name));
    gethostname(name+strlen(name),1024-strlen(name));
  }
  f=fopen(name,mode);
  free(name);
  return f;
}

static char *wisdom=(char*)NULL;

void rfftw_init(void)
{
  FILE *f;

  static int done=0;

  if (done) return;

  done=1;
  f = rfftw_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)!=FFTW_SUCCESS) {
    IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Bad wisdom data?");
  }
  fclose(f);

  wisdom=fftw_export_wisdom_to_string();
  IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Imported wisdom from
file.");
}


void rfftw_finish(void)
{
  FILE *f;
  static char *new_wisdom;

  new_wisdom=fftw_export_wisdom_to_string();
  if (new_wisdom==(char*)NULL) return;

  if (wisdom!=(char*)NULL && !strcmp(wisdom,new_wisdom)) {
    fftw_free(new_wisdom);
    return;
```

```
  }

  if (wisdom!=(char*)NULL) fftw_free(wisdom);

  wisdom=new_wisdom;

  f = rfftw_wisdom_file("w");
  if (f==(FILE*)NULL) {
    IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Couldn't write wisdom
file");
    return;
  }
  fftw_export_wisdom_to_file(f);
  fclose(f);
  IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Wrote wisdom to file");
}


/* Forward is REAL -> COMPLEX, use input dimensions for plan */
void rfftwnd_forward(IDL_VPTR in, IDL_VPTR out)
{
  fftw_real *src;
  fftw_complex *Fsrc;
  rfftwnd_plan plan;

  /* 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];

  rfftw_init();

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

  plan=rfftwnd_create_plan(in->value.arr->n_dim,(const int *)dim,
      FFTW_REAL_TO_COMPLEX,FFTW_MEASURE|FFTW_USE_WISDOM);

  rfftwnd_one_real_to_complex(plan,src,Fsrc);
  rfftwnd_destroy_plan(plan);
  rfftw_finish();
}


/* Backward is COMPLEX -> REAL, use OUTPUT dimensions for plan! */
void rfftwnd_backward(IDL_VPTR in, IDL_VPTR out)
{
  fftw_complex *src;
  fftw_real *Fsrc;
```

```
  rfftwnd_plan plan;

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

  rfftw_init();

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

  plan=rfftwnd_create_plan(out->value.arr->n_dim,(const int*)dim,
     FFTW_COMPLEX_TO_REAL,FFTW_MEASURE|FFTW_USE_WISDOM);

  rfftwnd_one_complex_to_real(plan,src,Fsrc);
  rfftwnd_destroy_plan(plan);
  rfftw_finish();
}


IDL_VPTR RFFTWND(int argc, IDL_VPTR argv[])
{
  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) */

  IDL_VPTR a=argv[i++];    /* Input array */
  IDL_VPTR odim=argv[i++], /* Output leading dim. for backward transform
*/

    call[1], /* For use when calling conversion routines */
    tmp;

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

  char odimreq[]=
    "2nd arg (leading output dimension) required for backward
transform";
  char odimwrong[]=
    "2nd arg (leading output dimension) has an inappropriate value";

  /* TYPE CHECKING / ALLOCATION SECTION */

  IDL_EXCLUDE_STRING(a);
  IDL_ENSURE_ARRAY(a);
```

```
  IDL_ENSURE_SIMPLE(a);

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

  /* If we're doing a forward transform, convert to Float,
*/
  /* if baclwards, convert to Complex, and make sure we have the leading
*/
  /* dimension size, and convert it to long
*/

  call[0] = a;
  if (a->type!=IDL_TYP_COMPLEX && a->type!=IDL_TYP_DCOMPLEX) {

    dir = -1; /* We're doing a forward transform */
    a = IDL_CvtFlt(1,call); /* May cause a to be tmp */

  } else {

    /* Require 2 arguments */
    if (argc!=2)
IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_LONGJMP,odimreq);

    dir = 1;    /* Backward transform */

#if IDL_VERSION_MAJOR >= 5 && IDL_VERSION_MINOR >= 4
    a = IDL_CvtComplex(1,call,(void*)NULL); /* May cause a to be tmp */
#else
    a = IDL_CvtComplex(1,call);
#endif

    IDL_EXCLUDE_UNDEF(odim); /* Output leading dimension */
    IDL_ENSURE_SIMPLE(odim);
    IDL_EXCLUDE_STRING(odim);
    IDL_ENSURE_SCALAR(odim);

    call[0] = odim;
    odim = IDL_CvtLng(1,call);

    /* Check validity of the output leading dimension */
    tmpi = 2*a->value.arr->dim[0] - odim->value.l;
    if (tmpi != 1 && tmpi != 2)
      IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_LONGJMP,odimwrong);

    /* Give warning about overwritten input arg. */
    if (ndim>1 && !(a->flags&IDL_V_TEMP))
      IDL_Message(IDL_M_NAMED_GENERIC,IDL_MSG_INFO,"Input
overwritten!");
```

```
  }

  /* 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<a->value.arr->n_dim; i++) dim[i] = a->value.arr->dim[i];

  /* Correct dimensions - for allocation */
  if (dir==-1) dim[ndim-1] = 2*(dim[ndim-1]/2+1);
  else         dim[ndim-1] = 2*dim[ndim-1];  /* Complex takes 2x FLOAT
*/

  /* Storage for result */
  IDL_MakeTempArray(IDL_TYP_FLOAT,a->value.arr->n_dim,dim,
      IDL_ARR_INI_INDEX,&tmp);

  /* Correct output leading dimension - to make correct plan */
  if (dir==1) {
    tmp->value.arr->n_elts /= tmp->value.arr->dim[ndim-1];
    tmp->value.arr->dim[ndim-1] = odim->value.l;
    tmp->value.arr->n_elts *= tmp->value.arr->dim[ndim-1];
  }

  if (dir==FFTW_REAL_TO_COMPLEX) rfftwnd_forward(a,tmp);
  else                           rfftwnd_backward(a,tmp);

  /* Swap dimensions back! */
  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;
    tmpi=tmp->value.arr->dim[i];
    tmp->value.arr->dim[i] = tmp->value.arr->dim[ndim-i-1];
    tmp->value.arr->dim[ndim-i-1] = tmpi;
  }

  i=0;

  if (a!=argv[i++]) IDL_DELTMP(a);
  if (odim!=argv[i++]) IDL_DELTMP(odim);

  if (dir==-1) {
    tmp->type = IDL_TYP_COMPLEX;  /* Change type, halve the size */
```

```
    tmp->value.arr->dim[0] /= 2;
    tmp->value.arr->n_elts /= 2;
  } else {
  }
  return tmp;
}

int IDL_Load(void)
{
  static IDL_SYSFUN_DEF2 func_def[] = {
    {RFFTWND,"RFFTWND",1,2, 0, 0}
    };
  return IDL_SysRtnAdd(func_def,TRUE,1);
}
```

.... and here is the dlm file.....

```
# $Id: rfftwnd.dlm,v 1.1 1999/08/16 10:59:04 steinhh Exp steinhh $
MODULE RFFTWND
DESCRIPTION N-dimensional Real fftw
VERSION $Revision: 1.1 $
BUILD_DATE $Date: 1999/08/16 10:59:04 $
SOURCE S.V.H.HAUGAN
FUNCTION RFFTWND    1 2
```

.... and here is a simple speed test comparison:

```
; time tests of fft routines - 2 D version

ntimes_array=[10,10,10,10,10,5,1]
expos=[5,6,7,8,9,10,11]
npts_array=2L^expos
npts_vals=n_elements(npts_array)
dt_FFTW=fltarr(npts_vals)
dt_IDL=fltarr(npts_vals)
for nn=0,npts_vals-1 do begin
        ntimes=ntimes_array[nn]
        npts=npts_array[nn]
        data=findgen(npts)#findgen(npts)
        nsize=(size(data))[1]
        fdataFFTW=rfftwnd(data)
        t0=systime(1)
        for n=1,ntimes do begin
                fdataFFTW=rfftwnd(data)
                 dataFFTW=rfftwnd(fdataFFTW,nsize)
        endfor
        dt=(systime(1)-t0)/ntimes
        print,'Time for n x n FFTW transform:',dt,' n=',npts
```

```
        dt_FFTW[nn]=dt

        t0=systime(1)
        for n=1,ntimes do begin
                fdataIDL=fft(data,/OVERWRITE)
                dataIDL=fft(fdataIDL,-1,/OVERWRITE)
        endfor
        dt=(systime(1)-t0)/ntimes
        dt_IDL[nn]=dt
        print,'Time for n x n  IDL transform:',dt,' n=',npts
endfor
for nn=0,npts_vals-1 do begin
        N=npts_array[nn]^2
        Nlog2N=N*ALOG(N)/ALOG(2) * 1e-6
        print,'2^'+strtrim(string(expos[nn]),2),$
                N,dt_FFTW[nn],dt_IDL[nn],$
                dt_IDL[nn]/dt_FFTW[nn],$
                dt_FFTW[nn]/Nlog2N,$
                dt_IDL[nn]/Nlog2N,$
                format='(A5," square array ",I10,5F10.4)'
endfor
end
---------------------- and the results on a DEC Alpha 5000 ----------
                          sec per FFT&INVERSE
                      NxN    FFTW      IDL     IDL/FFTW FFTW/NlogN
IDL/NlogN
  2^5 square array     32x32    0.0051    0.0006    0.1154    0.4951
0.0571
  2^6 square array     64x64    0.0058    0.0029    0.5085    0.1170
0.0595
  2^7 square array    128x128    0.0096    0.0215    2.2416    0.0419
0.0939
  2^8 square array    256x256    0.0241    0.1465    6.0821    0.0230
0.1397
  2^9 square array    512x512    0.0913    0.7532    8.2503    0.0193
0.1596
 2^10 square array  1024x1024    0.5685    3.6035    6.3387    0.0271
0.1718
 2^11 square array  2048x2048    5.2725   74.2175   14.0763    0.0571
0.8043
                                      *********
               THIS IS THE IMPORTANT COLUMN - execution time ratio
(IDL/FFTW)
```