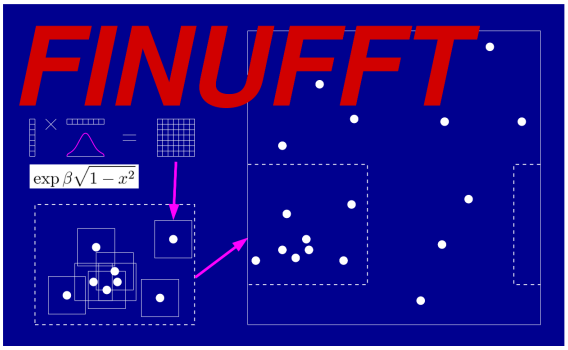
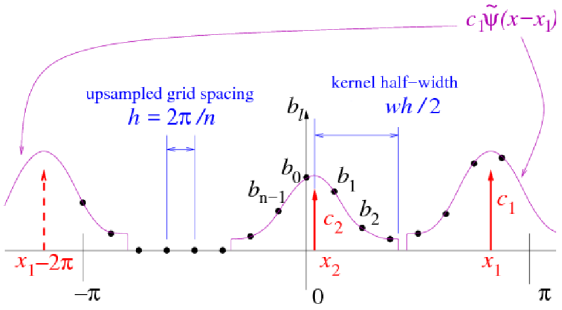
finufft Documentation

Release 1.2

Alex Barnett and Jeremy Magland

July 24, 2020





FINUFFT is a multi-threaded library to compute efficiently the three most common types of nonuniform fast Fourier transform (NUFFT) to a specified precision, in one, two, or three dimensions, on a multi-core shared-memory machine. It is extremely fast (typically achieving 10^6 to 10^8 points per second), has very simple interfaces to most major numerical languages (C/C++, Fortran, MATLAB, octave, python, and julia), but also has more advanced (vectorized and "guru") interfaces that allow multiple strength vectors and the reuse of FFT plans. It is written in C++ (with limited use of ++ features), OpenMP, and uses FFTW. It has been developed at the Center for Computational Mathematics at the Flatiron Institute, by *Alex Barnett and others*, and is released under an Apache v2 license.

CHAPTER

ONE

WHAT DOES FINUFFT DO?

As an example, given M arbitrary real numbers x_j and complex numbers c_j, with $j = 1, \dots, M$, and a requested integer number of modes N, FINUFFT computes the 1D type 1 (aka "adjoint") transform, which means it evaluates the N numbers

$$f_k = \sum_{j=1}^{M} c_j e^{ikx_j} \ , \qquad \text{for } \ k \in \mathbb{Z}, \quad -N/2 \le k \le N/2 - 1 \ . \tag{1.1}$$

As with other "fast" algorithms, FINUFFT does not evaluate this sum directly—which would take $O(NM)$ effort—but rather uses a sequence of steps (in this case, optimally chosen spreading, FFT, and deconvolution) to approximate the vector of answers (1.1) to within the user's desired relative tolerance with only $O(N \log N + M)$ effort, ie, quasi-linear. Thus the speed-up is similar to that of the FFT. You may want to jump to *quickstart*, or see the *definitions* of the type 2 and 3 transforms, and 2D and 3D cases.

One interpretation of (1.1) is: the returned values f_k are the *Fourier series coefficients* of the 2π-periodic distribution $f(x) := \sum_{j=1}^{M} c_j \delta(x - x_j)$, a sum of point-masses with arbitrary locations x_j and strengths c_j. Such exponential sums are needed in many applications in science and engineering, including signal processing (scattered data interpolation, applying convolutional transforms, fast summation), imaging (cryo-EM, CT, MRI gridding, coherent diffraction, VLBI astronomy), and numerical analysis (computing Fourier *transforms* of functions, moving between non-conforming quadrature grids, solving partial differential equations). See our *tutorials and demos* pages and the *related works* for examples of how to use the NUFFT in applications. In fact, there are several application areas where it has been overlooked that the needed computation is simply a NUFFT (eg, particle-mesh Ewald in molecular dynamics).

CHAPTER

TWO

WHY FINUFFT? FEATURES AND COMPARISON AGAINST OTHER NUFFT LIBRARIES

The basic scheme used by FINUFFT is not new, but there are many mathematical and software engineering improvements over other libraries. As is common in NUFFT algorithms, under the hood is an FFT on a regular "fine" (upsampled) grid—the user has no need to access this directly. Nonuniform points are either spread to, or interpolated from, this fine grid, using a specially designed kernel (see right figure above). Our main features are:

- **High speed**. For instance, at similar accuracy, FINUFFT is up to 10x faster than the multi-threaded Chemnitz NFFT3 library, and (in single-thread mode) up to 50x faster than the CMCL NUFFT library. This is achieved via:
  1. a simple new "exponential of semicircle" kernel that is provably close to optimal
  2. quadrature approximation for this kernel's Fourier transform
  3. load-balanced multithreaded spreading/interpolation (see left figure above)
  4. bin-sorting of points to improve cache reuse
  5. a low upsampling option for smaller FFTs, especially in type 3 transforms
  6. piecewise polynomial kernel evaluation (additions and multiplications only) that SIMD-vectorizes reliably on open-source compilers
- **Less RAM**. Our kernel is so fast that there is no point in precomputation; it is always evaluated on the fly. Thus our memory footprint is often an order of magnitude less than the fastest (precomputed) modes of competitors such as NFFT3 and MIRT, especially at high accuracy.
- **Automated kernel parameters**. Unlike many competitors, we do not force the user to worry about kernel choice or parameters. The user simply requests a desired relative accuracy, then FINUFFT chooses parameters that achieve this accuracy as fast as possible.
- **Simplicity**. We provide interfaces that perform a NUFFT with a single command—just like an FFT—from seven common languages/environments. For advanced users we also have "many vector" interfaces that can be much faster than repeated calls to the simple interface with the same points. Finally (like NFFT3) we have a "guru" interface for maximum flexibility, in all of these languages.

For technical details on much of the above see our *papers*. Note that there are other tasks (eg, transforms on spheres, inverse NUFFTs) provided by other libraries, such as NFFT3, that FINUFFT does not provide.

CHAPTER

THREE

DO I EVEN NEED A NUFFT?

A user's need for a nonuniform fast Fourier transform is often obscured by the lack of mathematical description in science application areas. Therefore, read our *tutorials and demos* to try and match to your task. Write the task in terms of one of the *three transform types*. If both M and N are larger than of order 10^2, FINUFFT should be the ticket. However, if M and/or N is small (of order 10 or less), there is no need for a "fast" algorithm: simply evaluate the sums directly.

If you need to fit off-grid data to a Fourier representation (eg, if you have off-grid $\mathbf{k}$-space samples of an unknown image) but you do not have *quadrature weights* for the off-grid points, you may need to *invert* the NUFFT, which actually means solving a large linear system; see the *tutorials and demos* and *references* [GLI] [KKP]. Poor coverage of the nonuniform point set leads to ill-conditioning and a heavy reliance on regularization.

Another scenario is that you wish to evaluate a forward transform such as (1.1) repeatedly with the same set of nonuniform points x_j, but *fresh* strength vectors $\{c_j\}_{j=1}^M$, as in the "many vectors" interface mentioned above. For small such problems it may be even faster to fill an N-by-M matrix A with entries $a_{kj} = e^{ikx_j}$, then use BLAS3 (eg `ZGEMM`) to compute $F = AC$, where each column of F and C is a new instance of (1.1). If you have very many columns this can be competitive with a NUFFT even for M and N up to 10^4, because BLAS3 is so fast.

CHAPTER

FOUR

DOCUMENTATION CONTENTS

Installation

Quick linux install instructions

In brief, go to the github page https://github.com/flatironinstitute/finufft and follow instructions to download the source (eg see the green button). Make sure you have packages `fftw3` and `fftw3-devel` installed. Then `cd` into your FINUFFT directory and `make test`. This should compile the static library in `lib-static/`, some C++ test drivers in `test/`, then run them, printing some terminal output ending in:

```
0 crashes out of 5 tests done
```

If this fails see the more detailed instructions below. If it succeeds, run `make lib` and proceed to link to the library. Alternatively, try one of our precompiled linux and OSX binaries. Type `make` to see a list of other aspects to build (language interfaces, etc). Consider installing `numdiff` as below to allow `make test` to perform a better accuracy check. Please read Usage and look in `examples/` and `test/` for other usage examples.

Dependencies

This library is fully supported for unix/linux and almost fully on Mac OSX. We have also heard that it can be compiled under Windows using MinGW; we also suggest trying within the Windows Subsystem for Linux (WSL).

For the basic libraries you need

- C++ compiler, such as `g++` packaged with GCC, or `clang` with OSX
- FFTW3
- GNU make

Optional:

- `numdiff` (preferred but not essential; enables better pass-fail accuracy validation)
- for Fortran wrappers: compiler such as `gfortran`
- for matlab/octave wrappers: MATLAB, or octave and its development libraries
- for the python wrappers you will need `python` and `pip` (it is assumed you have python v3; v2 is unsupported). You will also need `pybind11`
- for rebuilding new matlab/octave wrappers (experts only): `mwrap`

Tips for installing dependencies on linux

On a Fedora/CentOS linux system, dependencies can be installed as follows:

```
sudo yum install make gcc gcc-c++ gcc-gfortran fftw3 fftw3-devel libgomp octave octave-devel
```

Note: we are not exactly sure how to install python3 and pip3 using yum

Alternatively, on Ubuntu linux:

```
sudo apt-get install make build-essential libfftw3-dev gfortran numdiff python3 python3-pip octave li
```

For any linux flavor see below for the optional `numdiff` (and very optional `mwrap`). You should then compile via the various `make` tasks.

Note: GCC versions on linux. Rather than using the default GCC which may be as old as 4.8 or 5.4 on current linux systems, we **strongly** recommend you compile with a recent GCC version such as GCC 7.3 (which we used benchmarks in our SISC paper), or GCC 9.2.1. We do not recommend GCC versions prior to 7. We also **do not recommend GCC8** since its auto vectorization has worsened, and its kernel evaluation rate using the default looped piecewise-polynomial Horner code drops to less than 150 Meval/s/core on an i7. This contrasts 400-700 Meval/s/core achievable with GCC7 or GCC9 on i7. If you wish to test these raw kernel evaluation rates, do into `devel/`, compile `test_ker_ppval.cpp` and run `fig_speed_ker_ppval.m` in MATLAB. We are unsure if GCC8 is poor in Mac OSX (see below).

Tips for installing dependencies and compiling on Mac OSX

Note: Improved Mac OSX instructions, and possibly a brew package, will come shortly. Stay tuned. The below has been tested on 10.14 (Mojave) with both clang and gcc-8.

First you'll want to set up Homebrew, as follows. If you don't have Xcode, install Command Line Tools (this is only around 130 MB in contrast to the full 6 GB size of Xcode), by opening a terminal (from `/Applications/Utilities/`) and typing:

```
xcode-select --install
```

You will be asked for an administrator password. Then, also as an administrator, install Homebrew by pasting the installation command from https://brew.sh

Then do:

```
brew install libomp fftw
```

This happens to also install the latest GCC, which is 8.2.0 in our tests.

Note: There are two options for compilers: 1) the native `clang` which works with octave but will *not* so far allow you to link against fortran applications, or 2) GCC, which will allow fortran linking with `gfortran`, but currently fails with octave.

First the **clang route**, which is the default. Once you have downloaded FINUFFT, to set up for this, do:

```
cp make.inc.macosx_clang make.inc
```

This gives you compile flags that should work with `make test` and other tasks. Optionally, install `numdiff` as below. Then for python (note that pip is not installed with the default python v2):

```
brew install python3
pip3 install numpy pybind11
make python
```

This should generate the `finufftpy` module (and `finufftpy_cpp` which it depends on). However, we have found that it may fail with an error about `-lstdc++`, in which case you should try setting an environment variable:

```
export MACOSX_DEPLOYMENT_TARGET=10.14
```

We have also found that running:

```
pip3 install .
```

in the command line can work even when `make python` does not (probably to do with environment variables). Octave interfaces work out of the box:

```
brew install octave
make octave
```

Look in `make.inc.macosx_*`, and see below, for ideas for building MATLAB MEX interfaces.

Alternatively, here's the **GCC route**, which we have also tested on Movaje:

```
cp make.inc.macosx_gcc-8 make.inc
```

You must now by hand edit `python/setup.py`, changing `gcc` to `gcc-8` and `g++` to `g++-8`. Then proceed as above with python3. `make fortran` in addition to the above (apart from octave) should now work.

Note: Choosing GCC-8 in OSX there is a problem with octave MEX compilation. Please help if you can!

General notes about compilation and tests

We first describe compilation for default options (double precision, openmp) via GCC. If you have a nonstandard unix environment (eg a Mac) or want to change the compiler, then place your compiler and linking options in a new file `make.inc`. For example such files see `make.inc.*`. See the text of `makefile` for discussion of what can be overridden.

Compile and do a rapid (less than 1-second) test of FINUFFT via:

```
make test
```

This should compile the main libraries then run tests which should report zero crashes and zero fails. (If numdiff is absent, it instead produces output only about crashes; you will have to check by eye that accuracy is as expected.) One test is `test/basicpassfail` which does include a low-accuracy math test, producing the exit code 0 if success, nonzero if fail. You can check the exit code thus:

```
test/basicpassfail; echo $?
```

Use `make perftest` for larger spread/interpolation and NUFFT tests taking 10-20 seconds. This writes into `test/results/` where you will be able to compare to results from standard CPUs.

Run `make` without arguments for full list of possible make tasks.

`make examples` to compile and run the examples for calling from C++ and from C.

The `examples` and `test` directories are good places to see usage examples.

`make fortran` to compile and run the fortran wrappers and examples.

If there is an error in testing on a standard set-up, please file a bug report as a New Issue at https://github.com/flatironinstitute/finufft/issues

Custom library compilation options

*** UPDATE DUAL-PRECISION LIB

Single-threaded vs multithreaded are built with the same name, so you will have to move them to other locations, or build a 2nd copy of the repo, if you want to keep both versions.

Single-threaded: append `OMP=OFF` to the make task.

You *must* do at least `make objclean` before changing this option.

Building MATLAB/octave wrappers, including in Mac OSX

`make matlab` to build the MEX interface to matlab.

`make octave` to build the MEX-like interface to octave.

We have had success in Mac OSX Mojave compiling the octave wrapper out of the box. For MATLAB, the MEX settings may need to be overridden: edit the file `mex_C++_maci64.xml` in the MATLAB distro, to read, for instance:

```
CC="gcc-8"
CXX="g++-8"
CFLAGS="-ansi -D_GNU_SOURCE -fexceptions -fPIC -fno-omit-frame-pointer -pthread"
CXXFLAGS="-ansi -D_GNU_SOURCE -fPIC -fno-omit-frame-pointer -pthread"
```

These settings are copied from the `glnxa64` case. Here you will want to replace the compilers by whatever version of GCC you or the default gcc/g++ that are aliased to clang.

For pre-2016 MATLAB Mac OSX versions you'll instead want to edit the `maci64` section of `mexopts.sh`.

Building the python wrappers

First make sure you have python3 and pip3 (or python and pip) installed and that you can already compile the C++ library (eg via `make lib`). Python links to this compiled library. You will get an error unless you first compile the static library. Next make sure you have NumPy and pybind11 installed:

```
pip install numpy pybind11
```

You may then do `make python` which calls `pip` for the install then runs some tests. An additional test you could do is:

```
python python/run_speed_tests.py
```

See also Dan Foreman-Mackey's earlier repo that also wraps finufft, and from which we have drawn code: python-finufft

A few words about python environments

There can be confusion and conflicts between various versions of python and installed packages. It is therefore a very good idea to use virtual environments. Here's a simple way to do it (after installing python-virtualenv):

```
Open a terminal
virtualenv -p /usr/bin/python3 env1
. env1/bin/activate
```

Now you are in a virtual environment that starts from scratch. All pip installed packages will go inside the env1 directory. (You can get out of the environment by typing `deactivate`). Also see documentation for `conda`. In both cases `python` will call the version of python you set up, which these days should be v3.

Tips for installing optional dependencies

Installing numdiff

numdiff by Ivano Primi extends `diff` to assess errors in floating-point outputs. It is an optional dependency that provides a better pass-fail test; in particular it allows the accuracy check message `0 fails out of 5 tests done` when `make test` is done for FINUFFT. To install `numdiff` on linux, download the latest version from http://gnu.mirrors.pair.com/savannah/savannah/numdiff/ un-tar the package, cd into it, then build via `./configure; make; sudo make install`.

This compilation fails on Mac OSX, for which we found the following was needed in Mojave. Assume you un-tarred into `/usr/local/numdiff-5.9.0`. Then:

```
brew install gettext
./configure 'CFLAGS=-I/usr/local/opt/gettext/include' 'LDFLAGS=-L/usr/local/opt/gettext/lib'
make
sudo ln /usr/local/numdiff-5.9.0/numdiff /usr/local/bin
```

You should now be able to run `make test` in FINUFFT and get the second message about zero fails.

Installing MWrap

This is not needed for most users. MWrap is a very useful MEX interface generator by Dave Bindel. Make sure you have `flex` and `bison` installed. Download version 0.33 or later from http://www.cs.cornell.edu/~bindel/sw/mwrap, un-tar the package, cd into it, then:

```
make
sudo cp mwrap /usr/local/bin/
```

Directories in this package

When you `git clone https://github.com/flatironinstitute/finufft`, or unpack a tar ball, you will get the following. (Please see *installation* instructions)

- `finufft-manual.pdf` : the manual (auto-generated by sphinx)
- `docs` : source files for documentation (.rst files are human-readable)
- `README.md` : github-facing (and human text-reader) doc info
- `LICENSE` : how you may use this software

- `CHANGELOG` : list of changes, release notes
- `TODO` : list of things needed to fix or extend (also see git Issues)
- `makefile` : the GNU makefile (there are no makefiles in subdirectories)
- `src` : main library C++ sources
- `include` : header files including those for users to compile against
- `lib` : dynamic (.so) library will be built here
- `lib-static` : static (.a) library will be built here
- `test` : C++/bash validation tests, including: - `test/basicpassfail{f}` simple smoke test with exit code - `test/check_finufft.sh` is the main pass-fail validation bash script - `test/results` : validation comparison outputs (*.refout; do not remove these), and local test outputs (*.out; you may remove these)
- `perftest` : C++/bash performance and developer tests, including: - `test/spreadtestnd.sh` spread/interp performance test bash script - `test/nufft testnd.sh` NUFFT performance test bash script
- `examples` : simple example codes for calling the library from C++ and C
- `fortran` : wrappers and example drivers for Fortran (see `fortran/README`)
- `matlab` : wrappers, tests, examples for MATLAB/octave wrappers
- `python` : python wrappers (the `finufftpy` package), examples, and tests
- `contrib` : any 3rd-party codes

Mathematical definitions of transforms

We use notation with a general space dimensionality d, which will be 1, 2, or 3, in our library. The arbitrary (ie nonuniform) points in space are denoted $\mathbf{x}_j \in \mathbb{R}^d$, $j = 1, \dots, M$. We will see that for type 1 and type 2, without loss of generality one could restrict to the periodic box $[-\pi, \pi)^d$. For type 1 and type 3, each such NU point carries a given associated strength $c_j \in \mathbb{C}$. Type 1 and type 2 involve the Fourier "modes" (Fourier series coefficients) with integer indices lying in the set

$$K = K_{N_1,\dots,N_d} := K_{N_1} K_{N_2} \dots K_{N_d} \ ,$$

where

$$K_{N_i} := \begin{cases} \{-N_i/2, \dots, N_i/2 - 1\}, & N_i \text{ even}, \\ \{-(N_i-1)/2, \dots, (N_i-1)/2\}, & N_i \text{ odd}. \end{cases}$$

For instance, $K_{10} = \{-5, -4, \dots, 4\}$, whereas $K_{11} = \{-5, -4, \dots, 5\}$. Thus, in the 1D case K is an interval containing N_1 integer indices, in 2D it is a rectangle of $N_1 N_2$ index pairs, and in 3D it is a cuboid of $N_1 N_2 N_3$ index triplets.

Then the type 1 (nonuniform to uniform, aka "adjoint") NUFFT evaluates

$$f_{\mathbf{k}} := \sum_{j=1}^{M} c_j e^{\pm i \mathbf{k} \cdot \mathbf{x}_j} \qquad \text{for } \mathbf{k} \in K \tag{4.1}$$

This can be viewed as evaluating a set of Fourier series coefficients due to sources with strengths c_j at the arbitrary locations $\mathbf{x}_j$. Either sign of the imaginary unit in the exponential can be chosen in the interface. Note that our normalization differs from that of references [DR,GL].

The type 2 (U to NU, aka "forward") NUFFT evaluates

$$c_j := \sum_{\mathbf{k}\in K} f_{\mathbf{k}} e^{\pm i \mathbf{k}\cdot\mathbf{x}_j} \qquad \text{for } j = 1,\dots,M \tag{4.2}$$

This is the adjoint of the type 1, ie the evaluation of a given Fourier series at a set of arbitrary points. Both type 1 and type 2 transforms are invariant under translations of the NU points by multiples of 2π, thus one could require that all NU points live in the origin-centered box $[-\pi,\pi)^d$. In fact, as a compromise between library speed, and flexibility for the user (for instance, to avoid boundary points being flagged as outside of this box due to round-off error), our library only requires that the NU points lie in the three-times-bigger box $\mathbf{x}_j \in [-3\pi, 3\pi]^d$. This allows the user to choose a convenient periodic domain that does not touch this three-times-bigger box. However, there may be a slight speed increase if most points fall in $[-\pi,\pi)^d$.

Finally, the type 3 (NU to NU) transform does not have restrictions on the NU points, and there is no periodicity. Let $\mathbf{x}_j \in \mathbb{R}^d$, $j = 1,\dots,M$, be NU locations, with strengths $c_j \in \mathbb{C}$, and let $\mathbf{s}_k$, $k = 1,\dots,N$ be NU frequencies. Then the type 3 transform evaluates:

$$f_{\mathbf{k}} := \sum_{j=1}^{M} c_j e^{\pm i \mathbf{s}_k\cdot\mathbf{x}_j} \qquad \text{for } k = 1,\dots,N \tag{4.3}$$

For all three transforms, the computational effort scales like the product of the space-bandwidth products (real-space width times frequency-space width) in each dimension. For type 1 and type 2 this means near-linear scaling in the total number of modes $N := N_1 \dots N_d$. However, be warned that for type 3 this means that, even if N and M are small, if the product of the tightest intervals enclosing the coordinates of $\mathbf{x}_j$ and $\mathbf{s}_k$ is large, the algorithm will be inefficient. For such NU points, a direct sum should be used instead.

We emphasise that the NUFFT tasks that this library performs should not be confused with either the discrete Fourier transform (DFT), the (continuous) Fourier transform (although it may be used to approximate this via a quadrature rule), or the inverse NUFFT (the iterative solution of the linear system arising from nonuniform Fourier sampling, as in, eg, MRI). It is also important to know that, for NU points, *the type 1 is not the inverse of the type 2*. See the references for clarification.

Usage from C++ and C

Quick-start example in C++

Here's how to perform a 1D type-1 transform in double precision from C++, using STL complex vectors. First include our header, and some others needed for the demo:

```
#include "finufft.h"
#include <vector>
#include <complex>
#include <stdlib.h>
```

We need nonuniform points `x` and complex strengths `c`. Let's create random ones for now:

```
int M = 1e7;                                    // number of nonuniform points
vector<double> x(M);
vector<complex<double> > c(M);
complex<double> I = complex<double>(0.0,1.0);   // the imaginary unit
for (int j=0; j<M; ++j) {
  x[j] = M_PI*(2*((double)rand()/RAND_MAX)-1); // uniform random in [-pi,pi)
  c[j] = 2*((double)rand()/RAND_MAX)-1 + I*(2*((double)rand()/RAND_MAX)-1);
}
```

With `N` as the desired number of Fourier mode coefficients, allocate their output array:

```
int N = 1e6;                                    // number of output modes
vector<complex<double> > F(N);
```

Now do the NUFFT (with default options, indicated by the `NULL` in the following call). Since the interface is C-compatible, we pass pointers to the start of the arrays (rather than C++-style vector objects), and also pass `N`:

```
int ier = finufft1d1(M,&x[0],&c[0],+1,1e-9,N,&F[0],NULL);
```

This fills `F` with the output modes, in increasing ordering from frequency index `-N/2` up to `N/2-1`. The transform (10^7 points to 10^6 modes) takes 0.4 seconds on a laptop. The indexing is offset by `(int)N/2`, so that frequency `k` is output in `F[(int)N/2 + k]`. Here `+1` sets the sign of i in the exponentials (see *definitions*), `1e-9` requests 9-digit relative tolerance, and `ier` is a status output which is zero if successful (otherwise see *error codes*).

Note: FINUFFT works with a periodicity of 2π for type 1 and 2 transforms; see *definitions*. For example, nonuniform points $x = \pm\pi$ are equivalent. Points must lie in the input domain $[-3\pi, 3\pi)$, which allows the user to assume a convenient periodic domain such as $[-\pi, \pi)$ or $[0, 2\pi)$. To handle points outside of $[-3\pi, 3\pi)$ the user must fold them back into this domain before passing to FINUFFT. FINUFFT does not handle this case, for speed reasons. To use a different periodicity, linearly rescale your coordinates.

If instead you want to change some options, first put default values in a `nufft_opts` struct, make your changes, then pass the pointer to FINUFFT:

```
nufft_opts* opts = new nufft_opts;
finufft_default_opts(opts);
opts->debug = 1;                                // prints timing/debug info
int ier = finufft1d1(M,&x[0],&c[0],+1,tol,N,&F[0],opts);
```

Warning:

- Without the `finufft_default_opts` call, options may take on arbitrary values which may cause a crash.
- This usage is new as of version 1.2: `opts` is passed as a pointer in both places.

See `examples/simple1d1.cpp` for a simple full working demo of the above, including a test of the math. If you instead use single-precision arrays, replace the tag `finufft` by `finufftf` in each command; see `examples/simple1d1f.cpp`.

Then to compile on a linux/GCC system, linking to the double-precision static library, use eg:

```
g++ simple1d1.cpp -o simple1d1 -I$FINUFFT/include $FINUFFT/lib-static/libfinufft.a -fopenmp -lfftw3_
```

where `$FINUFFT` denotes the absolute path of your FINUFFT installation. Better is instead link to dynamic shared (`.so`) libraries, via eg:

```
g++ simple1d1.cpp -o simple1d1 -I$FINUFFT/include -L$FINUFFT/lib -lfinufft -lm
```

The `examples` and `test` directories are good places to see further usage examples. The documentation for all 18 simple interfaces, and the more flexible guru interface, follows below.

Quick-start example in C

The FINUFFT C++ interface is intentionally also C-compatible, for simplity. Thus, to use from C, the above example only needs to replace the C++ `vector` with C-style array creation. Using C99 style, the above code, with options setting, becomes:

```
#include <finufft.h>
#include <stdlib.h>
#include <complex.h>

int M = 1e7;              // number of nonuniform points
double* x = (double *)malloc(sizeof(double)*M);
double complex* c = (double complex*)malloc(sizeof(double complex)*M);
for (int j=0; j<M; ++j) {
  x[j] = M_PI*(2*((double)rand()/RAND_MAX)-1);  // uniform random in [-pi,pi)
  c[j] = 2*((double)rand()/RAND_MAX)-1 + I*(2*((double)rand()/RAND_MAX)-1);
}
int N = 1e6;              // number of modes
double complex* F = (double complex*)malloc(sizeof(double complex)*N);
nufft_opts opts;                        // make an opts struct
finufft_default_opts(&opts);            // set default opts (must do this)
opts.debug = 2;                         // more debug/timing to stdout
int ier = finufft1d1(M,x,c,+1,1e-9,N,F,&opts);

// (now do something with F here!...)

free(x); free(c); free(F);
```

See `examples/simple1d1c.c` and `examples/simple1d1cf.c` for double- and single-precision C examples, including the math check to insure the correct indexing of output modes.

2D example in C++

We assume Fortran-style contiguous multidimensional arrays, as opposed to C-style arrays of pointers; this allows the widest compatibility with other languages. Assuming the same headers as above, we first create points (x_j, y_j) in the square $[-\pi, \pi)^2$, and strengths as before:

```
int M = 1e7;                                    // number of nonuniform points
vector<double> x(M), y(M);
vector<complex<double> > c(M);
for (int j=0; j<M; ++j) {
  x[j] = M_PI*(2*((double)rand()/RAND_MAX)-1);
  y[j] = M_PI*(2*((double)rand()/RAND_MAX)-1);
  c[j] = 2*((double)rand()/RAND_MAX)-1 + I*(2*((double)rand()/RAND_MAX)-1);
}
```

Let's say we want `N1=1000` by `N2=2000` 2D Fourier coefficients. We allocate and do the (default options) transform thus:

```
int N1=1000, N2=2000;
vector<complex<double> > F(N1*N2);
int ier = finufft2d1(M,&x[0],&y[0], &c[0], +1, 1e-6, N1, N2, &F[0], NULL);
```

This transform takes 0.6 seconds on a laptop. The modes have increasing ordering from frequency index `-N1/2` to `N1/2-1` in the fast (`x`) dimension, then ordering `-N2/2` up to `N2/2-1` in the slow (`y`) dimension. So, the output frequency `(k1,k2)` is found in `F[(int)N1/2 + k1 + ((int)N2/2 + k2)*N1]`.

See `opts.modeord` in Options to instead use FFT-style mode ordering, which simply differs by an `fftshift` (as it is commonly called).

See `examples/simple2d1.cpp` for an example with a math check, to insure the modes are correctly indexed.

Vectorized interface example

A common use case is to perform a stack of identical transforms with the same size and nonuniform points, but for new strength vectors. (Applications include interpolating vector-valued data, or processing MRI images collected with a fixed set of k-space sample points.) Because it amortizes sorting, FFTW planning, and FFTW plan lookup, it can be faster to use a "vectorized" interface (which does the entire stack in one call) than to repeatedly call the above "simple" interfaces. This is especially true for many small problems. Here we show how to do a stack of `ntrans=10` 1D type 1 NUFFT transforms, in C++, assuming the same headers as in the first example above. The strength data vectors are taken to be contiguous (the whole first vector, followed by the second, etc, rather than interleaved.) Ie, viewed as a matrix in Fortran storage, each column is a strength vector.

```
int ntrans = 10;                                // how many transforms
int M = 1e7;                                    // number of nonuniform points
vector<double> x(M);
vector<complex<double> > c(M*ntrans);           // ntrans strength vectors
complex<double> I = complex<double>(0.0,1.0);   // the imaginary unit
for (int j=0; j<M; ++j)
  x[j] = M_PI*(2*((double)rand()/RAND_MAX)-1);
for (int j=0; j<M*ntrans; ++j)                  // fill all ntrans vectors...
  c[j] = 2*((double)rand()/RAND_MAX)-1 + I*(2*((double)rand()/RAND_MAX)-1);
int N = 1e6;                                    // number of output modes
vector<complex<double> > F(N*trans);            // ntrans output vectors
int ier = finufft1d1(M,&x[0],&c[0],+1,1e-9,N,&F[0],NULL);     // default opts
```

This takes 2.6 seconds on a laptop, around 1.4x faster than making 10 separate "simple" calls. The frequency index `k` in transform number `t` (zero-indexing the transforms) is in `F[k + (int)N/2 + N*t]`.

See `examples/many1d1.cpp` and `test/finufft?dmany_test.cpp` for more examples.

Guru interface example

If you want more flexibility than the above, use the "guru" interface: this is similar to that of FFTW3, and to the main interface of NFFT3. It lets you change the nonuniform points while keeping the same pointer to an FFTW plan for a particular number of stacked transforms with a certain number of modes. This avoids the overhead (typically 0.1 ms per thread) of FFTW checking for previous wisdom which would be significant when doing many small transforms. You may also send in a new set of stacked strength data (for type 1 and 3, or coefficients for type 2), reusing the existing FFTW plan and sorted points. Now we redo the above 2D type 1 C++ example with the guru interface.

One first makes a plan giving transform parameters, but no data:

```
// (assume x, y, c are filled, and F allocated, as in the 2D code above...)
int type=1, dim=2, ntrans=1;
int64_t Ns[] = {1000,2000};                     // N1,N2 as 64-bit int array
// step 1: make a plan...
finufft_plan plan;
int ier = finufft_makeplan(type, dim, Ns, +1, ntrans, 1e-6, &plan, NULL);
// step 2: send in M nonuniform points (just x, y in this case)...
finufft_setpts(plan, M, &x[0], &y[0], NULL, 0, NULL, NULL, NULL);
// step 3: do the planned transform to the c strength data, output to F...
finufft_execute(plan, &c[0], &F[0]);
// ... you could now send in new points, and/or do transforms with new c data
// ...
// step 4: when done, free the memory used by the plan...
finufft_destroy(plan);
```

This writes the Fourier coefficients to `F` just as in the earlier 2D example. One difference from the above simple and vectorized interfaces is that the `int64_t` type (aka `long long int`) is needed since the Fourier coefficient

dimensions are passed as an array.

> You must destroy a plan before making a new plan using the same plan object, otherwise a memory leak results.

The complete code with a math test is in `examples/guru2d1.cpp`, and for more examples see `examples/guru1d1*.c*`

Documentation of all C++ functions

All functions have double- and single-precision versions. We group the simple and vectorized interfaces together, by each of the nine transform types (dimensions 1,2,3, and types 1,2,3). The guru interface functions are defined at the end. You will also want to refer to the options and *error codes* which apply to all 46 routines.

A reminder on Fourier mode ordering. For example, if `N1=10` in a 1D type 1 or type 2 transform:

- if `opts.modeord=0`: frequency indices are ordered `-5,-4,-3,-2,-1,0,1,2,3,4` (CMCL ordering)
- if `opts.modeord=1`: frequency indices are ordered `0,1,2,3,4,-5,-4,-3,-2,-1` (FFT ordering)

The two orderings are related by a `fftshift`. This holds for each dimension. Multidimensional arrays are passed by a pointer to a contiguous Fortran-style array, with the "fastest" dimension x, then y (if present), then z (if present), then transform number (if `ntrans>1`). We do not use C or C++ style multidimensional arrays; this gives the most flexibility from several languages without loss of speed or memory due to unnecessary array copying.

Simple and vectorized interfaces

1D transforms

```
finufft1d1(int64_t M, double* x, complex<double>* c, int iflag, double eps, int64_t N1, complex<doubl
finufftf1d1(int64_t M, single* x, complex<single>* c, int iflag, single eps, int64_t N1, complex<sing

finufft1d1many(int ntr, int64_t M, double* x, complex<double>* c, int iflag, double eps, int64_t N1,
finufftf1d1many(int ntr, int64_t M, single* x, complex<single>* c, int iflag, single eps, int64_t N1,

 1D complex nonuniform FFT of type 1 (nonuniform to uniform).

 Computes via a fast algorithm, to precision eps, one or more transforms:

             M-1
    f[k1] =  SUM c[j] exp(+/-i k1 x(j))  for -N1/2 <= k1 <= (N1-1)/2
             j=0

 Inputs:
   ntr    how many transforms (vectorized "many" functions only, else ntr=1)
   M      number of nonuniform point sources
   x      nonuniform points (length M real array)
   c      source strengths (size M*ntr complex array)
   iflag  if >=0, uses +i in complex exponential, otherwise -i
   eps    desired relative precision; smaller is slower. This can be chosen
          from 1e-1 down to ~ 1e-14 (in double precision) or 1e-6 (in single)
   N1     number of output Fourier modes to be computed
   opts   pointer to options struct (see opts.rst), or NULL for defaults

 Outputs:
   f      Fourier mode coefficients (size N1*ntr complex array)
```

```
   return value  0: success, 1: success but warning, >1: error (see error.rst)

 Notes:
   * complex arrays interleave Re, Im values, and their size is stated with
     dimensions ordered fastest to slowest.
   * Fourier frequency indices in each dimension i are the integers lying
     in [-Ni/2, (Ni-1)/2]. See modeord in opts.rst for their ordering.
   * all nonuniform point coordinates must lie in [-3pi,3pi).
```

```
finufft1d2(int64_t M, double* x, complex<double>* c, int iflag, double eps, int64_t N1, complex<doubl
finufftf1d2(int64_t M, single* x, complex<single>* c, int iflag, single eps, int64_t N1, complex<sing

finufft1d2many(int ntr, int64_t M, double* x, complex<double>* c, int iflag, double eps, int64_t N1,
finufftf1d2many(int ntr, int64_t M, single* x, complex<single>* c, int iflag, single eps, int64_t N1,

 1D complex nonuniform FFT of type 2 (uniform to nonuniform).

 Computes via a fast algorithm, to precision eps, one or more transforms:

    c[j] = SUM   f[k1] exp(+/-i k1 x[j])      for j = 0,...,M-1
           k1
    where the sum is over integers -N1/2 <= k1 <= (N1-1)/2.

 Inputs:
   ntr    how many transforms (vectorized "many" functions only, else ntr=1)
   M      number of nonuniform point targets
   x      nonuniform points (length M real array)
   iflag  if >=0, uses +i in complex exponential, otherwise -i
   eps    desired relative precision; smaller is slower. This can be chosen
          from 1e-1 down to ~ 1e-14 (in double precision) or 1e-6 (in single)
   N1     number of input Fourier modes
   f      Fourier mode coefficients (size N1*ntr complex array)
   opts   pointer to options struct (see opts.rst), or NULL for defaults

 Outputs:
   c      values at nonuniform point targets (size M*ntr complex array)
   return value  0: success, 1: success but warning, >1: error (see error.rst)

 Notes:
   * complex arrays interleave Re, Im values, and their size is stated with
     dimensions ordered fastest to slowest.
   * Fourier frequency indices in each dimension i are the integers lying
     in [-Ni/2, (Ni-1)/2]. See modeord in opts.rst for their ordering.
   * all nonuniform point coordinates must lie in [-3pi,3pi).
```

```
finufft1d3(int64_t M, double* x, complex<double>* c, int iflag, double eps, int64_t N, double* s, cor
finufftf1d3(int64_t M, single* x, complex<single>* c, int iflag, single eps, int64_t N, single* s, co

finufft1d3many(int ntr, int64_t M, double* x, complex<double>* c, int iflag, double eps, int64_t N, d
finufftf1d3many(int ntr, int64_t M, single* x, complex<single>* c, int iflag, single eps, int64_t N,

 1D complex nonuniform FFT of type 3 (nonuniform to nonuniform).

 Computes via a fast algorithm, to precision eps, one or more transforms:

             M-1
    f[k]  =  SUM   c[j] exp(+-i s[k] x[j]),      for k = 0,...,N-1
             j=0
```

```
Inputs:
  ntr    how many transforms (vectorized "many" functions only, else ntr=1)
  M      number of nonuniform point sources
  x      nonuniform points (length M real array)
  c      source strengths (size M*ntr complex array)
  iflag  if >=0, uses +i in complex exponential, otherwise -i
  eps    desired relative precision; smaller is slower. This can be chosen
         from 1e-1 down to ~ 1e-14 (in double precision) or 1e-6 (in single)
  N      number of nonuniform frequency targets
  s      nonuniform frequency targets in R (length N real array)
  opts   pointer to options struct (see opts.rst), or NULL for defaults

Outputs:
  f     Fourier transform values at targets (size N*ntr complex array)
  return value  0: success, 1: success but warning, >1: error (see error.rst)

Notes:
  * complex arrays interleave Re, Im values, and their size is stated with
    dimensions ordered fastest to slowest.
```

2D transforms

```
finufft2d1(int64_t M, double* x, double* y, complex<double>* c, int iflag, double eps, int64_t N1, in
finufftf2d1(int64_t M, single* x, single* y, complex<single>* c, int iflag, single eps, int64_t N1, i

finufft2d1many(int ntr, int64_t M, double* x, double* y, complex<double>* c, int iflag, double eps, i
finufftf2d1many(int ntr, int64_t M, single* x, single* y, complex<single>* c, int iflag, single eps,

 2D complex nonuniform FFT of type 1 (nonuniform to uniform).

 Computes via a fast algorithm, to precision eps, one or more transforms:

                M-1
    f[k1,k2] =  SUM c[j] exp(+/-i (k1 x[j] + k2 y[j]))
                j=0

             for -N1/2 <= k1 <= (N1-1)/2,      -N2/2 <= k2 <= (N2-1)/2.

 Inputs:
  ntr    how many transforms (vectorized "many" functions only, else ntr=1)
  M      number of nonuniform point sources
  x,y    nonuniform point coordinates (length M real arrays)
  c      source strengths (size M*ntr complex array)
  iflag  if >=0, uses +i in complex exponential, otherwise -i
  eps    desired relative precision; smaller is slower. This can be chosen
         from 1e-1 down to ~ 1e-14 (in double precision) or 1e-6 (in single)
  N1     number of output Fourier modes to be computed (x direction)
  N2     number of output Fourier modes to be computed (y direction)
  opts   pointer to options struct (see opts.rst), or NULL for defaults

 Outputs:
  f      Fourier mode coefficients (size N1*N2*ntr complex array)
  return value  0: success, 1: success but warning, >1: error (see error.rst)

 Notes:
  * complex arrays interleave Re, Im values, and their size is stated with
```

```
     dimensions ordered fastest to slowest.
   * Fourier frequency indices in each dimension i are the integers lying
     in [-Ni/2, (Ni-1)/2]. See modeord in opts.rst for their ordering.
   * all nonuniform point coordinates must lie in [-3pi,3pi).
```

```
finufft2d2(int64_t M, double* x, double* y, complex<double>* c, int iflag, double eps, int64_t N1, in
finufftf2d2(int64_t M, single* x, single* y, complex<single>* c, int iflag, single eps, int64_t N1, i

finufft2d2many(int ntr, int64_t M, double* x, double* y, complex<double>* c, int iflag, double eps, i
finufftf2d2many(int ntr, int64_t M, single* x, single* y, complex<single>* c, int iflag, single eps,

 2D complex nonuniform FFT of type 2 (uniform to nonuniform).

 Computes via a fast algorithm, to precision eps, one or more transforms:

    c[j] =   SUM   f[k1,k2] exp(+/-i (k1 x[j] + k2 y[j]))    for j = 0,...,M-1
             k1,k2
    where the sum is over integers -N1/2 <= k1 <= (N1-1)/2,
                                   -N2/2 <= k2 <= (N2-1)/2.

 Inputs:
   ntr    how many transforms (vectorized "many" functions only, else ntr=1)
   M      number of nonuniform point targets
   x,y    nonuniform point coordinates (length M real arrays)
   iflag  if >=0, uses +i in complex exponential, otherwise -i
   eps    desired relative precision; smaller is slower. This can be chosen
          from 1e-1 down to ~ 1e-14 (in double precision) or 1e-6 (in single)
   N1     number of input Fourier modes (x direction)
   N2     number of input Fourier modes (y direction)
   f      Fourier mode coefficients (size N1*N2*ntr complex array)
   opts   pointer to options struct (see opts.rst), or NULL for defaults

 Outputs:
   c      values at nonuniform point targets (size M*ntr complex array)
   return value  0: success, 1: success but warning, >1: error (see error.rst)

 Notes:
   * complex arrays interleave Re, Im values, and their size is stated with
     dimensions ordered fastest to slowest.
   * Fourier frequency indices in each dimension i are the integers lying
     in [-Ni/2, (Ni-1)/2]. See modeord in opts.rst for their ordering.
   * all nonuniform point coordinates must lie in [-3pi,3pi).
```

```
finufft2d3(int64_t M, double* x, double* y, complex<double>* c, int iflag, double eps, int64_t N, dou
finufftf2d3(int64_t M, single* x, single* y, complex<single>* c, int iflag, single eps, int64_t N, si

finufft2d3many(int ntr, int64_t M, double* x, double* y, complex<double>* c, int iflag, double eps, i
finufftf2d3many(int ntr, int64_t M, single* x, single* y, complex<single>* c, int iflag, single eps,

 2D complex nonuniform FFT of type 3 (nonuniform to nonuniform).

 Computes via a fast algorithm, to precision eps, one or more transforms:

              M-1
    f[k]  =   SUM  c[j] exp(+-i (s[k] x[j] + t[k] y[j])),   for k = 0,...,N-1
              j=0

 Inputs:
```

```
   ntr    how many transforms (vectorized "many" functions only, else ntr=1)
   M      number of nonuniform point sources
   x,y    nonuniform point coordinates (length M real arrays)
   c      source strengths (size M*ntr complex array)
   iflag  if >=0, uses +i in complex exponential, otherwise -i
   eps    desired relative precision; smaller is slower. This can be chosen
          from 1e-1 down to ~ 1e-14 (in double precision) or 1e-6 (in single)
   N      number of nonuniform frequency targets
   s,t    nonuniform frequency target coordinates in R^2 (length N real arrays)
   opts   pointer to options struct (see opts.rst), or NULL for defaults

 Outputs:
   f     Fourier transform values at targets (size N*ntr complex array)
   return value  0: success, 1: success but warning, >1: error (see error.rst)

 Notes:
   * complex arrays interleave Re, Im values, and their size is stated with
     dimensions ordered fastest to slowest.
```

3D transforms

```
finufft3d1(int64_t M, double* x, double* y, double* z, complex<double>* c, int iflag, double eps, int
finufftf3d1(int64_t M, single* x, single* y, single* z, complex<single>* c, int iflag, single eps, in

finufft3d1many(int ntr, int64_t M, double* x, double* y, double* z, complex<double>* c, int iflag, do
finufftf3d1many(int ntr, int64_t M, single* x, single* y, single* z, complex<single>* c, int iflag, s

 3D complex nonuniform FFT of type 1 (nonuniform to uniform).

 Computes via a fast algorithm, to precision eps, one or more transforms:

                M-1
    f[k1,k2] =  SUM c[j] exp(+/-i (k1 x[j] + k2 y[j] + k3 z[j]))
                j=0

for -N1/2 <= k1 <= (N1-1)/2,  -N2/2 <= k2 <= (N2-1)/2,  -N3/2 <= k3 <= (N3-1)/2

 Inputs:
   ntr    how many transforms (vectorized "many" functions only, else ntr=1)
   M      number of nonuniform point sources
   x,y,z  nonuniform point coordinates (length M real arrays)
   c      source strengths (size M*ntr complex array)
   iflag  if >=0, uses +i in complex exponential, otherwise -i
   eps    desired relative precision; smaller is slower. This can be chosen
          from 1e-1 down to ~ 1e-14 (in double precision) or 1e-6 (in single)
   N1     number of output Fourier modes to be computed (x direction)
   N2     number of output Fourier modes to be computed (y direction)
   N3     number of output Fourier modes to be computed (z direction)
   opts   pointer to options struct (see opts.rst), or NULL for defaults

 Outputs:
   f      Fourier mode coefficients (size N1*N2*N3*ntr complex array)
   return value  0: success, 1: success but warning, >1: error (see error.rst)

 Notes:
   * complex arrays interleave Re, Im values, and their size is stated with
     dimensions ordered fastest to slowest.
```

```
   * Fourier frequency indices in each dimension i are the integers lying
     in [-Ni/2, (Ni-1)/2]. See modeord in opts.rst for their ordering.
   * all nonuniform point coordinates must lie in [-3pi,3pi).
```

```
finufft3d2(int64_t M, double* x, double* y, double* z, complex<double>* c, int iflag, double eps, int
finufftf3d2(int64_t M, single* x, single* y, single* z, complex<single>* c, int iflag, single eps, in

finufft3d2many(int ntr, int64_t M, double* x, double* y, double* z, complex<double>* c, int iflag, do
finufftf3d2many(int ntr, int64_t M, single* x, single* y, single* z, complex<single>* c, int iflag, s

 3D complex nonuniform FFT of type 2 (uniform to nonuniform).

 Computes via a fast algorithm, to precision eps, one or more transforms:

    c[j] =   SUM    f[k1,k2,k3] exp(+/-i (k1 x[j] + k2 y[j] + k3 z[j]))
           k1,k2,k3

    for j = 0,...,M-1,
    where the sum is over integers -N1/2 <= k1 <= (N1-1)/2,
                                   -N2/2 <= k2 <= (N2-1)/2,
                                   -N3/2 <= k3 <= (N3-1)/2.

 Inputs:
   ntr    how many transforms (vectorized "many" functions only, else ntr=1)
   M      number of nonuniform point targets
   x,y,z  nonuniform point coordinates (length M real arrays)
   iflag  if >=0, uses +i in complex exponential, otherwise -i
   eps    desired relative precision; smaller is slower. This can be chosen
          from 1e-1 down to ~ 1e-14 (in double precision) or 1e-6 (in single)
   N1     number of input Fourier modes (x direction)
   N2     number of input Fourier modes (y direction)
   N3     number of input Fourier modes (z direction)
   f      Fourier mode coefficients (size N1*N2*N3*ntr complex array)
   opts   pointer to options struct (see opts.rst), or NULL for defaults

 Outputs:
   c      values at nonuniform point targets (size M*ntr complex array)
   return value  0: success, 1: success but warning, >1: error (see error.rst)

 Notes:
   * complex arrays interleave Re, Im values, and their size is stated with
     dimensions ordered fastest to slowest.
   * Fourier frequency indices in each dimension i are the integers lying
     in [-Ni/2, (Ni-1)/2]. See modeord in opts.rst for their ordering.
   * all nonuniform point coordinates must lie in [-3pi,3pi).
```

```
finufft3d3(int64_t M, double* x, double* y, double* z, complex<double>* c, int iflag, double eps, int
finufftf3d3(int64_t M, single* x, single* y, single* z, complex<single>* c, int iflag, single eps, in

finufft3d3many(int ntr, int64_t M, double* x, double* y, double* z, complex<double>* c, int iflag, do
finufftf3d3many(int ntr, int64_t M, single* x, single* y, single* z, complex<single>* c, int iflag, s

 3D complex nonuniform FFT of type 3 (nonuniform to nonuniform).

 Computes via a fast algorithm, to precision eps, one or more transforms:

             M-1
    f[k]  =  SUM  c[j] exp(+-i (s[k] x[j] + t[k] y[j] + u[k] z[j])),
```

```
            j=0
                        for k = 0,...,N-1.

Inputs:
  ntr    how many transforms (vectorized "many" functions only, else ntr=1)
  M      number of nonuniform point sources
  x,y,z  nonuniform point coordinates (length M real arrays)
  c      source strengths (size M*ntr complex array)
  iflag  if >=0, uses +i in complex exponential, otherwise -i
  eps    desired relative precision; smaller is slower. This can be chosen
         from 1e-1 down to ~ 1e-14 (in double precision) or 1e-6 (in single)
  N      number of nonuniform frequency targets
  s,t,u  nonuniform frequency target coordinates in R^3 (length N real arrays)
  opts   pointer to options struct (see opts.rst), or NULL for defaults

Outputs:
  f     Fourier transform values at targets (size N*ntr complex array)
  return value  0: success, 1: success but warning, >1: error (see error.rst)

Notes:
  * complex arrays interleave Re, Im values, and their size is stated with
    dimensions ordered fastest to slowest.
```

Guru plan interface

Options parameters

Aside from the mandatory inputs (dimension, type, nonuniform points, strengths or coefficients, and, in C++/C/Fortran/MATLAB, sign of the imaginary unit and tolerance) FINUFFT has optional parameters. These adjust the workings of the algorithm, change the output format, or provide debug/timing text to stdout. Sensible default options are chosen, so that the new user need not worry about changing them. However, users wanting to try to increase speed or see more timing breakdowns will want to change options from their defaults. See each language doc page for how this is done, but is generally by creating an options structure, changing fields from their defaults, then passing this (or a pointer to it) to the simple, vectorized, or guru makeplan routines. Recall how to do this from C++:

```
.. code-block:: C++
```

    // (... set up M,x,c,tol,N, and allocate F here...) nufft_opts* opts; finufft_default_opts(opts); opts->debug = 1; int ier = finufft1d1(M,x,c,+1,tol,N,F,opts);

This setting produces more timing output to `stdout`.

In C/C++, not forget to call the command which sets default options (`finufft_default_opts`) before you start changing them or passing them to FINUFFT.

Summary of options and quick advice

Here is the 1-line summary of each option, with the full specifications below (see the header `include/nufft_opts.h`):

```
int debug;          // 0: silent, 1: text basic timing output
int spread_debug;   // passed to spread_opts, 0 (no text) 1 (some) or 2 (lots)
int spread_sort;    // passed to spread_opts, 0 (don't sort) 1 (do) or 2 (heuristic)
```

```
int spread_kerevalmeth; // "     spread_opts, 0: exp(sqrt()), 1: Horner ppval (faster)
int spread_kerpad;  // passed to spread_opts, 0: don't pad to mult of 4, 1: do
int chkbnds;        // 0: don't check if input NU pts in [-3pi,3pi], 1: do
int fftw;           // 0:FFTW_ESTIMATE, or 1:FFTW_MEASURE (slow plan, faster run)
int modeord;        // 0: increasing freqs (neg to pos), or 1: FFT-style order
double upsampfac;   // upsampling ratio sigma, either 2.0, or 1.25 (small FFTs)
int spread_thread;  // for ntrans>1 only. 0: auto, 1: sequential multithreaded, 2: parallel singleth
int maxbatchsize;   // for ntrans>1 only. max chunk size of data vectors. 0: auto
int showwarn;       // 0: don't print warnings to stderr; 1: do
int nthreads;       // number of threads to use, or 0: use all available
```

These options are of course listed in the code, in `include/nufft_opts.h`. Here are their default settings (defined in `src/finufft.cpp:finufft_default_opts`):

```
debug = 0;
spread_debug = 0;
spread_sort = 2;
spread_kerevalmeth = 1;
spread_kerpad = 1;
chkbnds = 1;
fftw = FFTW_ESTIMATE;
modeord = 0;
upsampfac = 2.0;
spread_thread = 0;
maxbatchsize = 0;
showwarn = 1;
nthreads = 0;
```

The main options you'll want to play with are `fftw` to try slower plan modes which give faster transforms (look at the FFTW3 docs), `debug` to look at timing output (to determine if your problem is spread/interpolation dominated or FFT dominated), `modeord` to flip the Fourier mode ordering, and `nthreads` if you want to try a different number of threads than the current maximum available through OpenMP. See *Troubleshooting* for good advice on trying options.

Some of the options are experts-only, and will result in slow or incorrect results. Please test them in a small known test case so you understand the effect.

To get the fastest run-time, we recommend that you experiment firstly with:

`fftw`, `upsampfac`, and `spread_sort`, detailed below. If you are having crashes, set `chkbnds=1` to see if illegal `x` non-uniform point coordinates are being input.

Notes on various options:

`spread_sort`: the default setting is `spread_sort=2` which applies the following heuristic rule: in 2D or 3D always sort, but in 1D, only sort if N (number of modes) > M/10 (where M is number of nonuniform pts).

`fftw`: The default FFTW plan is `FFTW_ESTIMATE`; however if you will be making multiple calls, consider `fftw=FFTW_MEASURE`, which could spend many seconds planning, but will give a faster run-time when called again. Note that FFTW plans are saved (by FFTW's library) automatically from call to call in the same executable (incidentally, also in the same MATLAB/octave or python session).

`upsampfac`: This is the internal factor by which the FFT is larger than the number of requested modes in each dimension. We have built efficient kernels for only two settings: `upsampfac=2.0` (standard), and `upsampfac=1.25` (lower RAM, smaller FFTs, but wider spreading kernel). The latter can be much faster when the number of nonuniform points is similar or smaller to the number of modes, and/or if low accuracy is required. It is especially much faster for type 3 transforms. However, the kernel widths w are about 50% larger in each dimension, which can lead to slower spreading (it can also be faster due to the smaller size of the fine grid). Thus only 9-digit accuracy can currently be reached when using `upsampfac=1.25`.

The remaining options only are relevant for multiple-vector calls, that is, using the simple interfaces containing the word "many", or the guru interface with `ntrans` > 1:

`spread_thread`: control how multithreading is used to spread/interpolate each batch of data.

- 0: makes an automatic choice.
- 1: acts on each vector in the batch in sequence, using multithreaded spread/interpolate. It can be slightly better than 2 for large problems.
- 2: acts on all vectors in batch simultaneously, assigning each a thread which performs single-threaded spread/interpolate. (This was used by Melody Shih for the original "2dmany" interface in 2018.) It is much better than 1 for all but large problems.
- 3: like 2 except allowing nested OMP parallelism, so multi-threaded spread-interpolate is used. (This was used by Andrea Malleo in 2019.) I have not yet found a case where this beats both 1 and 2.

`maxbatchsize`: set the largest batch size of data vectors. 0 makes an automatic choice. If you are unhappy with this, then for small problems it should equal the number of threads, while for large problems it appears that 1 is better (since otherwise too much simultaneous RAM movement occurs).

*** REWRITE AND SPLIT UP:

Usage and design notes

- We strongly recommend you use `upsampfac=1.25` for type-3; it reduces its run-time from around 8 times the types 1 or 2, to around 3-4 times. It is often also faster for type-1 and type-2, at low precisions.
- Sizes >=2^31 have been tested for C++ drivers (`test/finufft?d_test.cpp`), and work fine, if you have enough RAM. In fortran the interface is still 32-bit integers, limiting to array sizes <2^31. The fortran interface needs to be improved.
- C++ is used for all main libraries, almost entirely avoiding object-oriented code. C++ `std::complex<double>` (macroed to `CPX` and sometimes `dcomplex`) and FFTW complex types are mixed within the library, since to some extent our library is a glorified driver for FFTW. FFTW was considered universal and essential enough to be a dependency for the whole package.
- There is a hard-defined limit of `1e11` for the size of internal FFT arrays, set in `defs.h` as `MAX_NF`: if your machine has RAM of order 1TB, and you need it, set this larger and recompile. The point of this is to catch ridiculous-sized mallocs and exit gracefully. Note that mallocs smaller than this, but which still exceed available RAM, cause segfaults as usual. For simplicity of code, we do not do error checking on every malloc.
- As a spreading kernel function, we use a new faster simplification of the Kaiser–Bessel kernel, and eventually settled on piecewise polynomial approximation of this kernel. At high requested precisions, like the Kaiser–Bessel, this achieves roughly half the kernel width achievable by a truncated Gaussian. Our kernel is exp(-beta.sqrt(1-(2x/W)^2)), where W = nspread is the full kernel width in grid units. This (and Kaiser–Bessel) are good approximations to the prolate spheroidal wavefunction of order zero (PSWF), being the functions of given support [-W/2,W/2] whose Fourier transform has minimal L2 norm outside of a symmetric interval. The PSWF frequency parameter (see [ORZ]) is c = pi.(1-1/2sigma).W where sigma is the upsampling parameter. See our paper in the references.

Error (status) codes

In all FINUFFT interfaces, the returned value `ier` is a status indicator. It is 0 if successful, otherwise the error code has the following meanings (see `include/defs.h`):

```
1  requested tolerance epsilon too small to achieve
2  attemped to allocate internal array larger than MAX_NF (defined in defs.h)
3  spreader: fine grid too small compared to spread (kernel) width
4  spreader: if chkbnds=1, a nonuniform point coordinate is out of input range [-3pi,3pi]^d
5  spreader: array allocation error
6  spreader: illegal direction (should be 1 or 2)
7  upsampfac too small (should be >1.0)
8  upsampfac not a value with known Horner poly eval rule (currently 2.0 or 1.25 only)
9  ntrans not valid in "many" (vectorized) or guru interface (should be >= 1)
10 transform type invalid
11 general allocation failure
12 dimension invalid
13 spread_thread option invalid
```

When `ier=1` the transform(s) is/are still completed, at the smallest epsilon achievable, so the answer should be usable. For any other nonzero values of `ier` the transform may not have been performed and the output should not be trusted. However, we hope that the value of `ier` will help narrow down the problem.

FINUFFT sometimes also sends error reports to `stderr` if it detects faulty input parameters.

If you are getting error codes, please reread the documentation for your language, then see our *troubleshooting advice*.

Troubleshooting

If you are having issues (segfaults, slowness, "wrong" answers, etc), there is a high probability it is something we already know about, so please first read all of the advice below in the section relevant to your problem: math, speed, or segfaults.

Mathematical "issues" and advice

- When requested tolerance is around 10^{-14} or less in double-precision, or 10^{-6} or less in single-precision, it will most likely be impossible for FINUFFT (or any other NUFFT library) to achieve this, due to inevitable round-off error. Here, "error" is to be understood relative to the norm of the returned vector of values. This is especially true when there is a large number of modes in any single dimension (N_1, N_2 or N_3), since this empirically scales the round-off error (fortunately, round-off does not appear to scale with the total N or M). Such round-off error is analysed and measured in Section 4.2 of our SISC paper.
- If you request a tolerance that FINUFFT knows it cannot achieve, it will return `ier=1` after performing transforms as accurately as it can. However, the status `ier=0` does not imply that the requested accuracy *was* achieved, merely that parameters were chosen to give this estimated accuracy, if possible. As our SISC paper shows, for typical situations, relative ℓ_2 errors match the requested tolerances over a wide range. Users should always check *convergence* (by, for instance, varying `tol` and measuring any changes in their results); this is generally true in scientific computing.
- The type 1 and type 2 transforms are adjoints but **not inverses of each other** (unlike in the plain FFT case, where, up to a constant N, the adjoint is the inverse). Therefore, if you are not getting the expected answers, please check that you have not made this assumption. In the *tutorials* we will add examples showing how to invert the NUFFT; also see NFFT3 inverse transforms.

Speed issues and advice

If FINUFFT is slow (eg, less than 10^6 nonuniform points per second), here is some advice:

- Try printing debug output to see step-by-step progress by FINUFFT. Set `opts.debug` to 1 or 2 and look at the timing information.
- Try reducing the number of threads (recall as of v1.2 this is controlled externally to FINUFFT), perhaps down to 1 thread, to make sure you are not having collisions between threads. The latter is possible if large problems are run with a large number of (say more than 30) threads. We added the constant `MAX_USEFUL_NTHREADS` in `include/defs.h` to catch this case. Another corner case causing slowness is very many repetitions of small problems; see `test/manysmallprobs` which exceeds 10^7 points/sec with one thread via the guru interface, but can get ridiculously slower with many threads; see https://github.com/flatironinstitute/finufft/issues/86
- Try setting a crude tolerance, eg `tol=1e-3`. How many digits do you actually need? This has a big effect in higher dimensions, since the number of flops scales like $(\log 1/\epsilon)^d$, but not quite as big an effect as this scaling would suggest, because in higher dimensions the flops/RAM ratio is higher.
- If type 3, make sure your choice of points does not have a massive *space-bandwidth product* (ie, product of the volumes of the smallest d-dimension axes-aligned cuboids enclosing the nonuniform source and the target points); see Remark 5 of our SISC paper. In short, if the spreads of $\mathbf{x}_j$ and of $\mathbf{s}_k$ are both big, you may be in trouble. This can lead to enormous fine grids and hence slow FFTs. Set `opts.debug=1` to examine the `nf1`, etc, fine grid sizes being chosen, and the array allocation sizes. If they are huge, consider direct summation, as discussed *here*.
- The timing of the first FFTW call is complicated, depending on the FFTW flags (plan mode) used. This is really an FFTW planner flag usage question. Such issues are known, and modes benchmarked in other documentation, eg for 2D in poppy. In short, using more expensive FFTW planning modes like `FFTW_MEASURE` can give better performance for repeated FFTW calls, but be **much** more expensive in the first (planning) call. This is why we choose `FFTW_ESTIMATE` as our default `opts.fftw` option.
- Make sure you did not override `opts.spread_sort`, which if set to zero does no sorting, which can give very slow RAM access if the nonuniform points are ordered poorly (eg randomly) in larger 2D or 3D problems.
- Are you calling the simple interface a huge number of times for small problems, but these tasks have something in common (number of modes, or locations of nonuniform points)? If so, try the "many vector" or guru interface, which removes overheads in repeated FFTW plan look-up, and in bin-sorting. They can be 10-100x faster.

Crash (segfault) issues and advice

- The most common problem is passing in pointers to the wrong size of object,

eg, single vs double precision, or int32 vs int64. The library includes both precisions, so make sure you are calling the correct one (commands begin `finufft` for double, `finufftf` for single).

- If you use C++/C/Fortran and tried to change options, did you forget to call `finufft_default_opts` first?
- Maybe you have switched off nonuniform point bounds checking (`opts.chkbnds=0`) for a little extra speed? Try switching it on again to catch illegal coordinates.
- To isolate where a crash is occurring, set `opts.debug` to 1 or 2, and check the text output of the various stages. With a debug setting of 2 or above, when `ntrans>1` a large amount of text can be generated.
- To diagnose problems with the spread/interpolation stage, similarly setting `opts.spread_debug` to 1 or 2 will print even more output. Here the setting 2 generates a large amount of output even for a single transform.

Other known issues with library and interfaces

The master list is the github issues for the project page, https://github.com/flatironinstitute/finufft/issues

A secondary and more speculative list is in the `TODO` text file.

Please look through those issue topics, since sometimes workarounds are discussed before the problem is fixed in a release.

Bug reports

If you think you have found a new bug, and have read the above, please file a new issue on the github project page, https://github.com/flatironinstitute/finufft/issues . Include a minimal code which reproduces the bug, along with details about your machine, operating system, compiler, version of FINUFFT, and output with `opts.debug` at least 1. If you have a known bug and have ideas, please add to the comments for that issue.

You may also contact Alex Barnett (`abarnett` at-sign `flatironinstitute.org`) with FINUFFT in the subject line.

Tutorials and application demos

The following are instructive demos of using FINUFFT for a variety of spectrally-related tasks in numerical computing and signal/image processing. We will slowly grow the list (contact us to add one). For conciseness of code, and ease of writing, they are currently in MATLAB (tested on R2017a or later).

Fast evaluation of Fourier series at arbitrary points

This is a simple demo of using type 2 NUFFTs to evaluate a given 1D and then 2D Fourier series rapidly (close to optimal scaling) at arbitrary points. For conciseness of code, we use the MATLAB interface. The series we use are vaguely boring random ones relating to Gaussian random fields—please insert Fourier series coefficient vectors you care about.

1D Fourier series

Let our periodic domain be $[0, L)$, so that we get to see how to rescale from the fixed period of 2π in FINUFFT. We set up a random Fourier series with Gaussian decaying coefficients (this in fact is a sample from a stationary *Gaussian random field*, or *Gaussian process* with covariance kernel itself a periodized Gaussian):

```
L = 10;               % period
kmax = 500;           % bandlimit
k = -kmax:kmax-1;     % freq indices (negative up through positive mode ordering)
N = 2*kmax;           % # modes
rng(0);               % make some convenient Fourier coefficients...
fk = randn(N,1)+1i*randn(N,1);       % iid random complex data, column vec
k0 = 100;             % a freq scale parameter
fk = fk .* exp(-(k/k0).^2).';        % scale the amplitudes, kills high freqs
```

Now we use a 1D type 2 to evaluate this series at a large number of points very quickly:

```
M = 1e6; x = L*rand(1,M);         % make random target points in [0,L)
tol = 1e-12;
x_scaled = x * (2*pi/L);          % don't forget to scale to 2pi-periodic!
tic; c = finufft1d2(x_scaled,+1,tol,fk); toc    % evaluate Fourier series at x
```

```
Elapsed time is 0.026038 seconds.
```

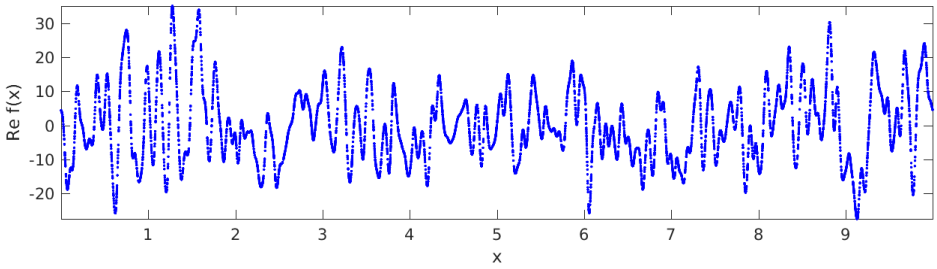
Compare this to a naive calculation (which serves to remind us exactly what sum FINUFFT approximates):

```
tic; cn = 0*c; for m=k, cn = cn + fk(m+N/2+1)*exp(1i*m*x_scaled.'); end, toc
norm(c-cn,inf)
```

```
Elapsed time is 11.679265 seconds.
ans =
      1.76508266507874e-11
```

Thus, with only 10^3 modes, FINUFFT is 500 times faster than naive multithreaded summation. (Naive summation with reversed loop order is even worse, taking 29 seconds.) We plot 1% of the resulting values and get the smooth but randomly-sampled graph below:

```
Mp = 1e4;                % how many pts to plot
jplot = 1:Mp;            % indices to plot
plot(x(jplot),real(c(jplot)),'b.'); axis tight; xlabel('x'); ylabel('Re f(x)');
```



See the full code matlab/examples/serieseval1d.m which also shows how to evaluate the same series on a uniform grid via the plain FFT.

2D Fourier series

Since we already know how to rescale to periodicity L, let's revert to the natural period and work in the square $[0, 2\pi)^2$. We create a random 2D Fourier series, which happens to be for a Gaussian random field with (doubly periodized) isotropic Matérn kernel of arbitrary power:

```
kmax = 500;                    % bandlimit per dim
k = -kmax:kmax-1;              % freq indices in each dim
N1 = 2*kmax; N2 = N1;          % # modes in each dim
[k1 k2] = ndgrid(k,k);         % grid of freq indices
rng(0); fk =  randn(N1,N2)+1i*randn(N1,N2);          % iid random complex modes
k0 = 30;                       % freq scale parameter
alpha = 3.7;                   % power; alpha>2 to converge in L^2
fk = fk .* ((k1.^2+k2.^2)/k0^2 + 1).^(-alpha/2);     % sqrt of spectral density
```

We then simply call a 2D type 2 to evaluate this double series at whatever target points you like:

```
M = 1e6; x = 2*pi*rand(1,M); y = 2*pi*rand(1,M);    % random targets in square
tol = 1e-9;
tic; c = finufft2d2(x,y,+1,tol,fk); toc    % evaluate Fourier series at (x,y)'s
```

```
Elapsed time is 0.092743 seconds.
```

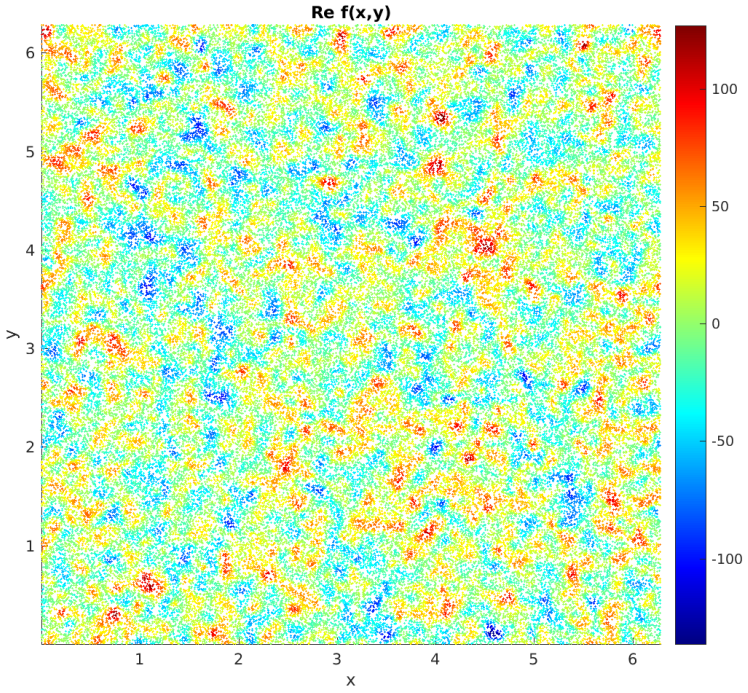
1 million modes to 1 million points in 92 milliseconds on a laptop is decent. We check the math (using a relative error measure) at just one (generic) point:

```
j = 1;                         % do math check on 1st target...
c1 = sum(sum(fk.*exp(1i*(k1*x(j)+k2*y(j)))));
abs(c1-c(j)) / norm(c,inf)
```

```
ans =
      2.30520830208365e-10
```

Finally we use a colored scatter plot to show the first 10% of the points in the square, and see samples of the underlying random field (reminiscent of WMAP microwave background data):

```
jplot = 1:1e5;            % indices to plot
scatter(x(jplot),y(jplot),1.0,real(c(jplot)),'filled'); axis tight equal
xlabel('x'); ylabel('y'); colorbar; title('Re f(x,y)');
```



See the full code matlab/examples/serieseval2d.m.

For background on Gaussian random fields, aka, Gaussian processes, see, eg, C. E. Rasmussen & C. K. I. Williams, *Gaussian Processes for Machine Learning*, the MIT Press, 2006. http://www.GaussianProcess.org/gpml

Numerical computation of continuous Fourier transforms of functions

It is common to assume that the FFT is the right tool to compute *continuous Fourier transforms*, but this is not so, unless you are content with very poor accuracy. The reason is that the FFT applies to equispaced data samples, that is, a quadrature scheme with only equispaced nodes.

Periodic Poisson solve on non-Cartesian quadrature grid

It is standard to use the FFT as a fast solver for the Poisson equation on a periodic domain, say $[0,2\pi)^d$. Namely, given f, find u satisfying

$$-\Delta u = f\ , \qquad \text{where} \int_{[0,2\pi)^d} f\,dx = 0\ ,$$

which has a unique solution up to constants. When f and u live on a regular Cartesian mesh, three steps are needed. The first takes an FFT to approximate the Fourier series coefficient array of f, the second divides by $\|k\|^2$, and the third uses another FFT to evaluate the Fourier series for u back on the original grid. Here is a MATLAB demo in $d = 2$ dimensions. Firstly we set up a smooth function, periodic up to machine precision:

```
w0 = 0.1;  % width of bumps
src = @(x,y) exp(-0.5*((x-1).^2+(y-2).^2)/w0^2)-exp(-0.5*((x-3).^2+(y-5).^2)/w0^2);
```

Now we do the FFT solve, using a loop to check convergence with respect to `n` the number of grid points in each dimension:

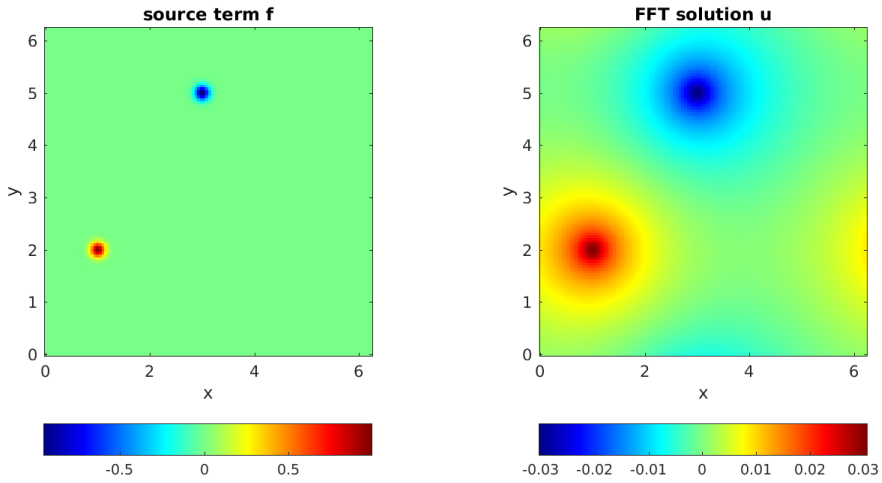
```
ns = 40:20:120;               % convergence study of grid points per side
for i=1:numel(ns), n = ns(i);
  x = 2*pi*(0:n-1)/n;         % grid
  [xx yy] = ndgrid(x,x);      % ordering: x fast, y slow
  f = src(xx,yy);             % eval source on grid
  fhat = ifft2(f);            % step 1: Fourier coeffs by Euler-F projection
  k = [0:n/2-1 -n/2:-1];      % Fourier mode grid
  [kx ky] = ndgrid(k,k);
  kfilter = 1./(kx.^2+ky.^2);     % -(Laplacian)^{-1} in Fourier space
  kfilter(1,1) = 0;           % kill the zero mode (even if inconsistent)
  kfilter(n/2+1,:) = 0; kfilter(:,n/2+1) = 0;  % kill n/2 modes since non-symm
  u = fft2(kfilter.*fhat);    % steps 2 and 3
  u = real(u);
  fprintf('n=%d:\t\tu(0,0) = %.15e\n',n,u(1,1))    % check conv at a point
end
```

We observe spectral convergence to 14 digits:

```
n=40:         u(0,0) = 1.551906153625019e-03
n=60:         u(0,0) = 1.549852227637310e-03
n=80:         u(0,0) = 1.549852190998224e-03
n=100:        u(0,0) = 1.549852191075839e-03
n=120:        u(0,0) = 1.549852191075828e-03
```

Here we plot the FFT solution:

```
figure; subplot(1,2,1); imagesc(x,x,f'); colorbar('southoutside');
axis xy equal tight; title('source term f'); xlabel('x'); ylabel('y');
subplot(1,2,2); imagesc(x,x,u'); colorbar('southoutside');
axis xy equal tight; title('FFT solution u'); xlabel('x'); ylabel('y');
```



Now let's say you wish to do a similar Poisson solve on a **non-Cartesian grid** covering the same domain. There are two cases: a) the grid is unstructured and you do not know the weights of a quadrature scheme, or b) you do know the weights of a quadrature scheme (which ususally implies that the grid is structured, such as arising from a different coordinate system or an adaptive subdivision). By *quadrature scheme* we mean nodes $x_j \in \mathbb{R}^d$, $j = 1, \dots, M$, and weights w_j such that, for all smooth functions f,

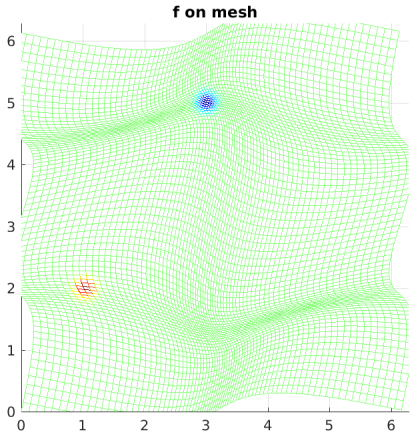
$$\int_{[0,2\pi)^d} f(x)dx \approx \sum_{j=1}^{M} f(x_j)w_j$$

holds to sufficient accuracy. We consider case b) only. For demo purposes, we use a simple smooth diffeomorphism from $[0, 2\pi)^2$ to itself to define a distorted mesh (the associated quadrature weights will come from the determinant of the Jacobian):

```
map = @(t,s) [t + 0.5*sin(t) + 0.2*sin(2*s); s + 0.3*sin(2*s) + 0.3*sin(s-t)];
mapJ = @(t,s) [1 + 0.5*cos(t), 0.4*cos(2*s); ...
               -0.3*cos(s-t),  1+0.6*cos(2*s)+0.3*cos(s-t)]; % its 2x2 Jacobian
```

For convenience of checking the solution against the above one, we chose the map to take the origin to itself. To visualize the grid, we plot f on it, noting that it covers the domain when periodically extended:

```
t = 2*pi*(0:n-1)/n;              % 1d unif grid
[tt ss] = ndgrid(t,t);
xxx = map(tt(:)',ss(:)');
xx = reshape(xxx(1,:),[n n]); yy = reshape(xxx(2,:),[n n]);  % 2D NU pts
f = src(xx,yy);
figure; mesh(xx,yy,f); view(2); axis equal; axis([0 2*pi 0 2*pi]); title('f on mesh');
```

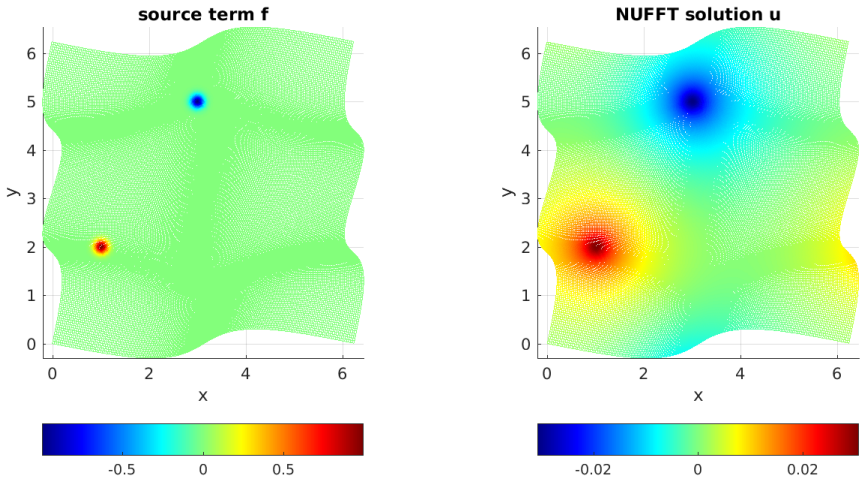


To solve on this grid, replace step 1 above by evaluating the Euler-Fourier formula using the quadrature scheme, which needs a type-1 NUFFT, and step 3 (evaluation on the nonuniform grid) by a type-2 NUFFT. Step 2 (the frequency filter) remains the same. Here is the demo code:

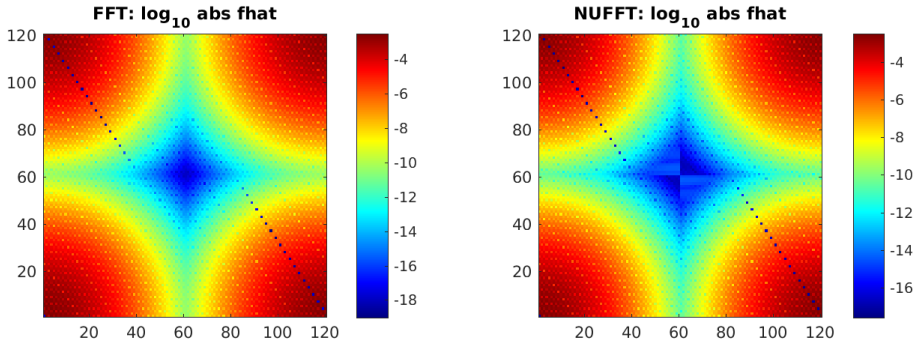
```
tol = 1e-12;             % NUFFT precision
ns = 80:40:240;          % convergence study of grid points per side
for i=1:numel(ns), n = ns(i);
  t = 2*pi*(0:n-1)/n;    % 1d unif grid
  [tt ss] = ndgrid(t,t);
  xxx = map(tt(:)',ss(:)');
  xx = reshape(xxx(1,:),[n n]); yy = reshape(xxx(2,:),[n n]);  % 2d NU pts
  J = mapJ(tt(:)',ss(:)');
  detJ = J(1,1:n^2).*J(2,n^2+1:end) - J(2,1:n^2).*J(1,n^2+1:end);
  ww = detJ / n^2;       % 2d quadr weights, including 1/(2pi)^2 in E-F integr
  f = src(xx,yy);
  Nk = 0.5*n; Nk = 2*ceil(Nk/2);  % modes to trust due to quadr err
  o.modeord = 1;         % use fft output mode ordering
  fhat = finufft2d1(xx(:),yy(:),f(:).*ww(:),1,tol,Nk,Nk,o);  % do E-F
  k = [0:Nk/2-1 -Nk/2:-1];   % Fourier mode grid
  [kx ky] = ndgrid(k,k);
  kfilter = 1./(kx.^2+ky.^2);  % inverse -Laplacian in k-space (as above)
  kfilter(1,1) = 0; kfilter(Nk/2+1,:) = 0; kfilter(:,Nk/2+1) = 0;
  u = finufft2d2(xx(:),yy(:),-1,tol,kfilter.*fhat,o);  % eval filt F series @ NU
  u = reshape(real(u),[n n]);
  fprintf('n=%d:\tNk=%d\tu(0,0) = %.15e\n',n,Nk,u(1,1))  % check conv at same pt
end
```

Here a convergence parameter (`Nk = 0.5*n`) had to be set to choose how many modes to trust with the quadrature. Thus `n` is about twice what it needed to be in the uniform case, accounting for the stretching of the grid. The convergence is again spectral, down to at least `tol`, and matches the FFT solution at the test point to 12 relative digits:

```
n=80:           Nk=40   u(0,0) = 1.549914931081811e-03
n=120:          Nk=60   u(0,0) = 1.549851996895389e-03
n=160:          Nk=80   u(0,0) = 1.549852191032026e-03
n=200:          Nk=100  u(0,0) = 1.549852191076891e-03
n=240:          Nk=120  u(0,0) = 1.549852191077001e-03
```



Finally, here is the decay of the modes $\hat{f}_k$ on a log plot, for the FFT and NUFFT versions. They are identical down to the level `tol`:



The full code is at matlab/examples/poisson2dnuquad.m.

Note: If the non-Cartesian grids were of *tensor product* form, one could instead exploit 1D NUFFTs for the above, and, most likely the use of BLAS3 (`ZGEMM` with an order-`n` dense NUDFT matrix) would be optimal.

Note: Using the NUFFT as above does *not* give an optimal scaling scheme in the case of a **fully adaptive grid**, because all frequencies must be handled up to the highest one needed. The latter is controlled by the smallest spatial scale, so that the number of modes needed, N, is no smaller than the number in a *uniform* spatial discretization of the original domain at resolution needed to capture the smallest features. In other words, the advantage of full adaptivity is lost when using the NUFFT, and one may as well have used the FFT with a uniform Cartesian grid. To remedy this and recover linear complexity in the fully adaptive case, an FMM could be used to convolve f with the (periodized) Laplace fundamental solution to obtain u, or a multigrid or direct solver used on the discretization of the Laplacian on

the adaptive grid.

For further applications, see *references*, and these PDF slides.

Usage from Fortran

We provide Fortran interfaces that are very similar to those in C/C++. We deliberately use "legacy" Fortran style (in the terminology of FFTW), enabling the widest applicability and avoiding the complexity of later Fortran features. Namely, we use f77, with two features from f90: dynamic allocation and derived types. The latter is only needed if options must be changed from default values.

Simple example

To perform a double-precision 1D type 1 transform from `M` nonuniform points `xj` with strengths `cj`, to `N` output modes whose coefficients will be written into the `fk` array, using 9-digit tolerance, the $+i$ imaginary sign, and default options, the declarations and call are

```
      integer ier,iflag
      integer*8 N,M
      real*8, allocatable :: xj(:)
      real*8 tol
      complex*16, allocatable :: cj(:),fk(:)
      integer*8, allocatable :: null

!     (...allocate xj, cj, and fk, and fill xj and cj here...)

      tol = 1.0D-9
      iflag = +1
      call finufft1d1(M,xj,cj,iflag,tol,N,fk,null,ier)
```

which writes the output to `fk`, and the status to the integer `ier`. A status 0 indicates success, otherwise error codes are in *here*. All available OMP threads are used, unless FINUFFT was built single-thread. (Note that here the unallocated `null` is simply a way to pass a NULL pointer to our C++ wrapper; another would be `%val(0_8)`.) For a minimally complete test code demonstrating the above see `fortran/examples/simple1d1.f`.

To compile (eg using GCC/linux) and link such a program against the FINUFFT dynamic (`.so`) library (which links all dependent libraries):

```
gfortran -I $(FINUFFT)/include simple1d1.f -o simple1d1 -L$(FINUFFT)/lib -lfinufft
```

where `$(FINUFFT)` indicates the top-level FINUFFT directory. Or, using the static library, one must list dependent libraries:

```
gfortran -I $(FINUFFT)/include simple1d1.f -o simple1d1 $(FINUFFT)/lib-static/libfinufft.a -lfftw3 -l
```

Alternatively you may want to compile with `g++` and use `-lgfortran` at the end of the compile statement instead of `-lstdc++`. In Mac OSX, replace `fftw3_omp` by `fftw3_threads`, and if you use clang, `-lgomp` by `-lomp`. See `makefile` and `make.inc.*`.

Note: Our simple interface is designed to be a near drop-in replacement for the native f90 CMCL libraries of Greengard-Lee. The differences are: i) we added a penultimate argument in the list which allows options to be changed, and ii) our normalization differs for type 1 transforms (one divides our output by M to match the CMCL output).

Changing options

To choose non-default options in the above example, create an options derived type, set it to default values, change whichever you wish, and pass it to FINUFFT, for instance

```
      include 'finufft.fh'
      type(nufft_opts) opts

!     (...declare, allocate, and fill stuff as above...)

      call finufft_default_opts(opts)
      opts%debug = 2
      opts%upsampfac = 1.25d0
      call finufft1d1(M,xj,cj,iflag,tol,N,fk,opts,ier)
```

See `fortran/examples/simple1d1.f` for the complete code, and below for the complete list of Fortran subroutines available, and more complicated examples.

Summary of Fortran interface

The naming of routines is as in C/C++. Eg, `finufft2d3` means double-precision 2D transform of type 3. `finufft2d3many` means applying double-precision 2D transforms of type 3 to a stack of many strength vectors (vectorized interface). `finufft2d3f` means single-precision 2D type 3. The guru interface has very similar arguments to its C/C++ version. Compared to C/C++, all argument lists have `ier` appended at the end, to which the status is written. These routines and arguments are, in double-precision:

```
      include 'finufft.fh'

      integer ier,iflag,ntrans,type,dim
      integer*8 M,N1,N2,N3,Nk
      integer*8 plan,n_modes(3)
      real*8, allocatable :: xj(:),yj(:),zj(:), sk(:),tk(:),uk(:)
      real*8 tol
      complex*16, allocatable :: cj(:), fk(:)
      type(nufft_opts) opts

!     simple interface
      call finufft1d1(M,xj,cj,iflag,tol,N1,fk,opts,ier)
      call finufft1d2(M,xj,cj,iflag,tol,N1,fk,opts,ier)
      call finufft1d3(M,xj,cj,iflag,tol,Nk,sk,fk,opts,ier)
      call finufft2d1(M,xj,yj,cj,iflag,tol,N1,N2,fk,opts,ier)
      call finufft2d2(M,xj,yj,cj,iflag,tol,N1,N2,fk,opts,ier)
      call finufft2d3(M,xj,yj,cj,iflag,tol,Nk,sk,tk,fk,opts,ier)
      call finufft3d1(M,xj,yj,zj,cj,iflag,tol,N1,N2,N3,fk,opts,ier)
      call finufft3d2(M,xj,yj,zj,cj,iflag,tol,N1,N2,N3,fk,opts,ier)
      call finufft3d3(M,xj,yj,zj,cj,iflag,tol,Nk,sk,tk,uk,fk,opts,ier)

!     vectorized interface
      call finufft1d1many(ntrans,M,xj,cj,iflag,tol,N1,fk,opts,ier)
      call finufft1d2many(ntrans,M,xj,cj,iflag,tol,N1,fk,opts,ier)
      call finufft1d3many(ntrans,M,xj,cj,iflag,tol,Nk,sk,fk,opts,ier)
      call finufft2d1many(ntrans,M,xj,yj,cj,iflag,tol,N1,N2,fk,opts,ier)
      call finufft2d2many(ntrans,M,xj,yj,cj,iflag,tol,N1,N2,fk,opts,ier)
      call finufft2d3many(ntrans,M,xj,yj,cj,iflag,tol,Nk,sk,tk,fk,opts,ier)
      call finufft3d1many(ntrans,M,xj,yj,zj,cj,iflag,tol,N1,N2,N3,fk,opts,ier)
      call finufft3d2many(ntrans,M,xj,yj,zj,cj,iflag,tol,N1,N2,N3,fk,opts,ier)
      call finufft3d3many(ntrans,M,xj,yj,zj,cj,iflag,tol,Nk,sk,tk,uk,fk,opts,ier)
```

```
!     guru interface
      call finufft_makeplan(type,dim,n_modes,iflag,ntrans,tol,plan,opts,ier)
      call finufft_setpts(plan,M,xj,yj,zj,Nk,sk,yk,uk,ier)
      call finufft_execute(plan,cj,fk,ier)
      call finufft_destroy(plan,ier)
```

The single-precision routines are identical except with the replacement of finufft with finufft. They are defined (from the C++ side) in `fortran/finufftfort.cpp`.

Code examples

The `fortran/examples` directory contains the following demos, in both precisions. Each has a math test to check the correctness of some or all outputs:

```
simple1d1.f         - 1D type 1, simple interface, default and various opts
guru1d1.f           - 1D type 1, guru interface, default and various opts
nufft1d_demo.f      - 1D types 1,2,3, minimally changed from CMCL demo codes
nufft2d_demo.f      - 2D "
nufft3d_demo.f      - 3D "
nufft2dmany_demo.f - 2D types 1,2,3, vectorized (many strengths) interface
```

These are the double-precision file names; the single precision have a trailing f before the .f. The last four here are modified from demos in the CMCL NUFFT libraries. The first three of these have been changed only to use FINUFFT, The final tolerance they request is `tol=1d-16`. For this case FINUFFT will report a warning that it cannot achieve it, and gets merely around 10^{-14}. The last four demos require direct summation (slow) reference implementations of the transforms in `fortran/directft`, modified from their CMCL counterparts only to remove the $1/M$ prefactor for type 1 transforms.

All demos have self-contained example GCC compilation/linking commands in their comment headers. For dynamic linking so that execution works from any directory, bake in an absolute path via the compile flag `-Wl,-rpath,$(FINUFFT)/lib`.

For authorship and licensing of the Fortran wrappers, see the directory README

MATLAB/octave interfaces

Quick-start examples

To demo a single 1D transform of type 1 (nonuniform points to uniform Fourier coefficients), we set up random data then do the transform as follows:

```
M = 1e5;                              % number of NU source points
x = 2*pi*rand(M,1);                   % points in a 2pi-periodic domain
c = randn(M,1)+1i*randn(M,1);         % iid random complex data (row or col vec)
N = 2e5;                              % how many desired Fourier modes?
f = finufft1d1(x,c,+1,1e-12,N);       % do it (takes around 0.02 sec)
```

The column vector output `f` should be interpreted as the Fourier coefficients with frequency indices `k = -N/2:N/2-1`. (This is because `N` is even; otherwise `k = -(N-1)/2:(N-1)/2`.) The values in `f` are accurate (relative to this vector's 2-norm) to roughly 12 digits, as requested by the tolerance argument `1e-12`. Choosing a larger (ie, worse) tolerance leads to faster transforms. The `+1` controls the sign in the exponential; recall equation (1) on the *front page*. All options maybe be changed from their defaults, for instance:

```
o.modeord = 1;                          % choose FFT-style output mode ordering
f = finufft1d1(x,c,+1,1e-12,N,o);       % do it
```

The above usage we call the "simple" interface. There is also a "vectorized" interface which does the transform for multiple stacked strength vectors, using the same nonuniform points each time. We demo this, reusing `x` and `N` from above:

```
ntr = 1e2;                              % number of vectors (transforms to do)
C = randn(M,ntr)+1i*randn(M,ntr);       % iid random complex data (matrix)
F = finufft1d1(x,C,+1,1e-12,N);         % do them (takes around 1.2 sec)
```

Here this is nearly twice as fast as doing 100 separate calls to the simple interface. For smaller transform sizes the acceleration factor of this vectorized call can be much higher.

If you want yet more control, consider using the "guru" interface. This can be faster than fresh calls to the simple or vectorized interfaces for the same number of transforms, for reasons such as this: the nonuniform points can be changed between transforms, without forcing FFTW to look up a previously stored plan. Usually, such an acceleration is only important when doing repeated small transforms, where "small" means each transform takes of order 0.01 sec or less. Here we use the guru interface to repeat the first demo above:

```
type = 1; ntr = 1; o.modeord = 1;       % transform type, #transforms, opts
N = 2e5;                                % how many desired Fourier modes?
plan = finufft_plan(1,N,+1,ntr,1e-12,o);      % plan for N output modes
M = 1e5;                                % number of NU source points
plan.setpts(2*pi*rand(M,1),[],[]);      % set some nonuniform points
c = randn(M,1)+1i*randn(M,1);           % iid random complex data (row or col vec)
f = plan.execute(c);                    % do the transform (0.008 sec)
% ...one could now change the points with setpts, and/or do new transforms
% with new c data...
delete(plan);                           % don't forget to clean up
```

Finally, we demo a 2D type 1 transform using the simple interface. Let's request a rectangular Fourier mode array of 1000 modes in the x direction but 500 in the y direction. The source points are in the square of side length 2π:

```
M = 1e6; x = 2*pi*rand(1,M); y = 2*pi*rand(1,M);     % points in [0,2pi]^2
c = randn(M,1)+1i*randn(M,1);           % iid random complex data (row or col vec)
N1 = 1000; N2 = 500;                    % desired Fourier mode array sizes
f = finufft2d1(x,y,c,+1,1e-9,N1,N2);          % do it (takes around 0.08 sec)
```

The resulting output `f` is indeed size 1000 by 500. The first dimension (number of rows) corresponds to the x input coordinate, and the second to y.

If you need to change the definition of the period from 2π, simply linearly rescale your points before sending them to FINUFFT.

Note: Under the hood FINUFFT has double- and single-precision libraries. The simple and vectorized MATLAB/octave interfaces infer which to call by checking the class of its input arrays, which must all match (ie, all must be `double` or all must be `single`). Since by default MATLAB arrays are double-precision, this is the precision that all of the above examples run in. To perform single-precision transforms, send in single-precision data. In contrast, precision in the guru interface is set with the `finufft_plan` option string `o.floatprec`, either `'double'` (the default), or `'single'`.

See tests and examples in the repo and *tutorials and demos* for plenty more MATLAB examples.

Full documentation

Here are the help documentation strings for all MATLAB/octave interfaces. They only abbreviate the options (for full documentation see opts). Informative warnings and errors are raised in MATLAB style with unique codes (see `../matlab/errhandler.m`, `../matlab/finufft.mw`, and `../valid_*.m`). The low-level error number codes are not used.

If you have added the `matlab` directory of FINUFFT correctly to your MATLAB path via something like `addpath FINUFFT/matlab`, then `help finufft/matlab` will give the summary of all commands:

```
% FINUFFT: Flatiron Institute Nonuniform Fast Fourier Transform
% Version 1.2.0 23-Jul-2020
%
% Basic and many-vector interfaces
%   finufft1d1 - 1D complex nonuniform FFT of type 1 (nonuniform to uniform).
%   finufft1d2 - 1D complex nonuniform FFT of type 2 (uniform to nonuniform).
%   finufft1d3 - 1D complex nonuniform FFT of type 3 (nonuniform to nonuniform).
%   finufft2d1 - 2D complex nonuniform FFT of type 1 (nonuniform to uniform).
%   finufft2d2 - 2D complex nonuniform FFT of type 2 (uniform to nonuniform).
%   finufft2d3 - 2D complex nonuniform FFT of type 3 (nonuniform to nonuniform).
%   finufft3d1 - 3D complex nonuniform FFT of type 1 (nonuniform to uniform).
%   finufft3d2 - 3D complex nonuniform FFT of type 2 (uniform to nonuniform).
%   finufft3d3 - 3D complex nonuniform FFT of type 3 (nonuniform to nonuniform).
%
% Guru interface
%   finufft_plan - create guru plan object for one/many general nonuniform FFTs.
%   setpts       - process nonuniform points for general FINUFFT transform(s).
%   execute      - do a single or many-vector FINUFFT transform in a plan.
```

The individual commands have the following help documentation:

Python interface

*** TO DO ***

*** here's the old stuff...

These python interfaces are by Daniel Foreman-Mackey, Jeremy Magland, and Alex Barnett, with help from David Stein. See the installation notes for how to install these interfaces; the main thing to remember is to compile the library before trying to *pip install*. Below is the documentation for the nine routines. The 2d1 and 2d2 "many vector" interfaces are now also included.

Notes:

1. The module has been designed not to recompile the C++ library; rather, it links to the existing static library. Therefore this library must have been compiled before building python interfaces.
2. In the below, "float" and "complex" refer to double-precision for the default library. One can compile the library for single-precision, but the python interfaces are untested in this case.
3. NumPy input and output arrays are generally passed directly without copying, which helps efficiency in large low-accuracy problems. In 2D and 3D, copying is avoided when arrays are Fortran-ordered; hence choose this ordering in your python code if you are able (see `python_tests/accuracy_speed_tests.py`).
4. Fortran-style writing of the output to a preallocated NumPy input array is used. That is, such an array is treated as a pointer into which the output is written. This avoids creation of new arrays. The python call return value is merely a status indicator.

finufftpy.**nufft1d1**(*x*, *c*, *ms=None*, *out=None*, *eps=None*, *isign=1*, ***kwargs*)
1D type-1 (aka adjoint) complex nonuniform fast Fourier transform

```
             nj-1
    f(k1) =  SUM c[j] exp(+/-i k1 x(j))  for -ms/2 <= k1 <= (ms-1)/2
             j=0
```

Parameters

- **x** (*float[nj]*) – nonuniform source points, valid only in [-3pi,3pi]
- **c** (*complex[nj] or complex[nj,ntransf]*) – source strengths
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)
- **ms** (*int*) – number of Fourier modes requested, may be even or odd; in either case the modes are integers lying in [-ms/2, (ms-1)/2]
- **out** (*complex[ms] or complex[ms,ntransf]*) – output Fourier mode values. Should be initialized as a numpy array of the correct size
- ****kwargs** (*nufft_opts + dtype, optional*) – nufft option fields and precision type as keyword arguments

Note: The output is written into the out array if supplied.

Returns ndarray of the result

Example

see python_tests/demo1d1.py

finufftpy.**nufft1d2**(*x*, *f*, *out=None*, *eps=None*, *isign=-1*, ***kwargs*)
1D type-2 (aka forward) complex nonuniform fast Fourier transform

```
    c[j] = SUM   f[k1] exp(+/-i k1 x[j])      for j = 0,...,nj-1
            k1

      where sum is over -ms/2 <= k1 <= (ms-1)/2.
```

Parameters

- **x** (*float[nj]*) – nonuniform target points, valid only in [-3pi,3pi]
- **f** (*complex[ms] or complex[ms,ntransf]*) – Fourier mode coefficients, where ms is even or odd In either case the mode indices are integers in [-ms/2, (ms-1)/2]
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)
- **out** (*complex[nj] or complex[nj,ntransf]*) – output values at targets. Should be initialized as a numpy array of the correct size
- ****kwargs** (*nufft_opts + dtype, optional*) – nufft option fields and precision type as keyword arguments

Note: The output is written into the out array if supplied.

Returns ndarray of the result

Example

see `python_tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft1d3`**(*x*, *c*, *s*, *out=None*, *eps=None*, *isign=1*, ***kwargs*)
1D type-3 (NU-to-NU) complex nonuniform fast Fourier transform

```
            nj-1
    f[k]  =  SUM   c[j] exp(+-i s[k] x[j]),      for k = 0, ..., nk-1
            j=0
```

Parameters

- **`x`** (*`float[nj]`*) – nonuniform source points, in R
- **`c`** (*`complex[nj] or complex[nj,ntransf]`*) – source strengths
- **`isign`** (*`int`*) – if >=0, uses + sign in exponential, otherwise - sign
- **`eps`** (*`float`*) – precision requested (>1e-16)
- **`s`** (*`float[nk]`*) – nonuniform target frequency points, in R
- **`out`** (*`complex[nk] or complex[nk,ntransf]`*) – output values at target frequencies. Should be initialized as a numpy array of the correct size
- **`**kwargs`** (*`nufft_opts + dtype, optional`*) – nufft option fields and precision type as keyword arguments

Note: The output is written into the out array if supplied.

Returns ndarray of the result

Example

see `python_tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft2d1`**(*x*, *y*, *c*, *ms=None*, *mt=None*, *out=None*, *eps=None*, *isign=1*, ***kwargs*)
2D type-1 (aka adjoint) complex nonuniform fast Fourier transform

```
                nj-1
    f(k1,k2) =  SUM c[j] exp(+/-i (k1 x(j) + k2 y[j])),
                j=0
                      for -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2
```

Parameters

- **`x`** (*`float[nj]`*) – nonuniform source x-coords, valid only in [-3pi,3pi]
- **`y`** (*`float[nj]`*) – nonuniform source y-coords, valid only in [-3pi,3pi]

- **c** (*complex[nj] or complex[nj,ntransf]*) – source strengths
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)
- **ms** (*int*) – number of Fourier modes in x-direction, may be even or odd; in either case the modes are integers lying in [-ms/2, (ms-1)/2]
- **mt** (*int*) – number of Fourier modes in y-direction, may be even or odd; in either case the modes are integers lying in [-mt/2, (mt-1)/2]
- **out** (*complex[ms,mt] or complex[ms,mt,ntransf]*) – output Fourier mode values. Should be initialized as a Fortran-ordered (ie ms fast, mt slow) numpy array of the correct size
- ****kwargs** (*nufft_opts + dtype, optional*) – nufft option fields and precision type as keyword arguments

Note: The output is written into the out array if supplied.

Returns ndarray of the result

Example

see `python/tests/accuracy_speed_tests.py`

finufftpy.**nufft2d2**(*x*, *y*, *f*, *out=None*, *eps=None*, *isign=-1*, ***kwargs*)

2D type-2 (aka forward) complex nonuniform fast Fourier transform

```
    c[j] =   SUM   f[k1,k2] exp(+/-i (k1 x[j] + k2 y[j])),  for j = 0,...,nj-1
          k1,k2

    where sum is over -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2
```

Parameters

- **x** (*float[nj]*) – nonuniform target x-coords, valid only in [-3pi,3pi]
- **y** (*float[nj]*) – nonuniform target y-coords, valid only in [-3pi,3pi]
- **f** (*complex[ms,mt] or complex[ms,mt,ntransf]*) – Fourier mode coefficients, where ms and mt are either even or odd; in either case their mode range is integers lying in [-m/2, (m-1)/2], with mode ordering in all dimensions given by modeord. Ordering is Fortran-style, ie ms fastest.
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)
- **out** (*complex[nj] or complex[nj,ntransf]*) – output values at targets. Should be initialized as a numpy array of the correct size
- ****kwargs** (*nufft_opts + dtype, optional*) – nufft option fields and precision type as keyword arguments

Note: The output is written into the out array if supplied.

Returns ndarray of the result

Example

see `python_tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft2d3`**(*x*, *y*, *c*, *s*, *t*, *out=None*, *eps=None*, *isign=1*, ***kwargs*)

2D type-3 (NU-to-NU) complex nonuniform fast Fourier transform

```
             nj-1
f[k]   =     SUM   c[j] exp(+-i s[k] x[j] + t[k] y[j]),  for k = 0,...,nk-1
             j=0
```

Parameters

- **`x`** (*`float[nj]`*) – nonuniform source point x-coords, in R
- **`y`** (*`float[nj]`*) – nonuniform source point y-coords, in R
- **`c`** (*`complex[nj] or complex[nj,ntransf]`*) – source strengths
- **`isign`** (*`int`*) – if >=0, uses + sign in exponential, otherwise - sign
- **`eps`** (*`float`*) – precision requested (>1e-16)
- **`s`** (*`float[nk]`*) – nonuniform target x-frequencies, in R
- **`t`** (*`float[nk]`*) – nonuniform target y-frequencies, in R
- **`out`** (*`complex[nk] or complex[nk,ntransf]`*) – output values at target frequencies. Should be initialized as a numpy array of the correct size
- **`**kwargs`** (*`nufft_opts + dtype, optional`*) – nufft option fields and precision type as keyword arguments

Note: The output is written into the out array if supplied.

Returns ndarray of the result

Example

see `python_tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft3d1`**(*x*, *y*, *z*, *c*, *ms=None*, *mt=None*, *mu=None*, *out=None*, *eps=None*, *isign=1*, ***kwargs*)

3D type-1 (aka adjoint) complex nonuniform fast Fourier transform

```
                nj-1
f(k1,k2,k3) =   SUM c[j] exp(+/-i (k1 x(j) + k2 y[j] + k3 z[j])),
                j=0
   for -ms/2 <= k1 <= (ms-1)/2,
     -mt/2 <= k2 <= (mt-1)/2,  -mu/2 <= k3 <= (mu-1)/2
```

Parameters

- **x** (*float[nj]*) – nonuniform source x-coords, valid only in [-3pi,3pi]
- **y** (*float[nj]*) – nonuniform source y-coords, valid only in [-3pi,3pi]
- **z** (*float[nj]*) – nonuniform source z-coords, valid only in [-3pi,3pi]
- **c** (*complex[nj] or complex[nj,ntransf]*) – source strengths
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)
- **ms** (*int*) – number of Fourier modes in x-direction, may be even or odd; in either case the modes are integers lying in [-ms/2, (ms-1)/2]
- **mt** (*int*) – number of Fourier modes in y-direction, may be even or odd; in either case the modes are integers lying in [-mt/2, (mt-1)/2]
- **mu** (*int*) – number of Fourier modes in z-direction, may be even or odd; in either case the modes are integers lying in [-mu/2, (mu-1)/2]
- **out** (*complex[ms,mt,mu] or complex[ms,mt,mu,ntransf]*) – output Fourier mode values. Should be initialized as a Fortran-ordered (ie ms fastest) numpy array of the correct size
- ****kwargs** (*nufft_opts + dtype, optional*) – nufft option fields and precision type as keyword arguments

Note: The output is written into the out array if supplied.

Returns ndarray of the result

Example

see `python_tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft3d2`**(*x*, *y*, *z*, *f*, *out=None*, *eps=None*, *isign=-1*, ***kwargs*)

3D type-2 (aka forward) complex nonuniform fast Fourier transform

```
    c[j] =   SUM   f[k1,k2,k3] exp(+/-i (k1 x[j] + k2 y[j] + k3 z[j])).
         k1,k2,k3
                   for j = 0,...,nj-1,  where sum is over
    -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2, -mu/2 <= k3 <= (mu-1)/2
```

Parameters

- **x** (*float[nj]*) – nonuniform target x-coords, valid only in [-3pi,3pi]
- **y** (*float[nj]*) – nonuniform target y-coords, valid only in [-3pi,3pi]
- **z** (*float[nj]*) – nonuniform target z-coords, valid only in [-3pi,3pi]
- **f** (*complex[ms,mt,mu] or complex[ms,mt,mu,ntransf]*) – Fourier mode coefficients, where ms, mt and mu are either even or odd; in either case their mode range is integers lying in [-m/2, (m-1)/2], with mode ordering in all dimensions given by modeord. Ordering is Fortran-style, ie ms fastest.
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign

- **eps** (*float*) – precision requested (>1e-16)
- **out** (*complex[nj] or complex[nj,ntransf]*) – output values at targets. Should be initialized as a numpy array of the correct size
- ****kwargs** (*nufft_opts + dtype, optional*) – nufft option fields and precision type as keyword arguments

Note: The output is written into the out array if supplied.

Returns ndarray of the result

Example

see `python_tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft3d3`**(*x*, *y*, *z*, *c*, *s*, *t*, *u*, *out=None*, *eps=None*, *isign=1*, ***kwargs*)

3D type-3 (NU-to-NU) complex nonuniform fast Fourier transform

```
             nj-1
    f[k]  =  SUM   c[j] exp(+-i s[k] x[j] + t[k] y[j] + u[k] z[j]),
             j=0
                                                  for k = 0,...,nk-1
```

Parameters

- **x** (*float[nj]*) – nonuniform source point x-coords, in R
- **y** (*float[nj]*) – nonuniform source point y-coords, in R
- **z** (*float[nj]*) – nonuniform source point z-coords, in R
- **c** (*complex[nj] or complex[nj,ntransf]*) – source strengths
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)
- **s** (*float[nk]*) – nonuniform target x-frequencies, in R
- **t** (*float[nk]*) – nonuniform target y-frequencies, in R
- **u** (*float[nk]*) – nonuniform target z-frequencies, in R
- **out** (*complex[nk] or complex[nk,ntransf]*) – output values at target frequencies. Should be initialized as a numpy array of the correct size
- ****kwargs** (*nufft_opts + dtype, optional*) – nufft option fields and precision type as keyword arguments

Note: The output is written into the out array if supplied.

Returns ndarray of the result

Example

see `python_tests/accuracy_speed_tests.py`

Julia interface

Ludvig af Klinteberg has built FINUFFT.jl, an interface from the Julia language. This package will automatically download and build FINUFFT at installation, as long as GCC is available. It has been tested on Linux and Mac OS X (the latter with GCC 8).

Related packages

Other recommended NUFFT libraries

- NFFT3: well-supported and multi-featured C++ library using FFTW. Has MATLAB MEX interface. However, significantly slower and/or more memory-intensive than FINUFFT (see reference [FIN]). Has many more general abilities, eg, inverse NUFFT. We are working on this too.
- CMCL NUFFT: NYU single-threaded Fortran library using self-contained FFT, fast Gaussian gridding kernel. Has MATLAB MEX interface. Much (up to 50x even for one thread) slower than FINUFFT, but very easy to compile.
- cuFINUFFT: Our GPU version of FINUFFT, for single precision in 2D and 3D, type 1 and 2. Still under development by Melody Shih (NYU) and others. Often achieves speeds around 10x the CPU version.
- MIRT Michigan Image Reconstruction Toolbox. Native MATLAB, single-threaded sparse mat-vec, prestores all kernel evaluations, thus is memory-intensive but surprisingly fast for a single-threaded implementation. However, slower than FINUFFT for all tolerances smaller than 0.1.
- PyNUFFT Python code supporting CPU and GPU operation. Have not compared against FINUFFT yet.

Also see the summary of library performances in our paper [FIN] in the *references*.

Dependent packages, users, and citations

Here we list packages that depend on FINUFFT, and papers or groups using it. Papers that merely cite our work are listed separately at the bottom. Please let us know (and use github's dependent package link) if you are a user or package maintainer but not listed.

Packages relying on FINUFFT

Here are some packages dependent on FINUFFT (please let us know of others, and also add them to github's Used By feature):

1. SMILI, very long baseline interferometry reconstruction code by Kazu Akiyama and others, uses FINUFFT (2d1, 2d2, Fortran interfaces) as a key library. Akiyama used SMILI to reconstruct the famous black hole image in 2019 from the Event Horizon Telescope.
2. ASPIRE: software for cryo-EM, based at Amit Singer's group at Princeton. github

3. sinctransform: C++ and MATLAB codes to evaluate sums of the sinc and sinc^2 kernels between arbitrary nonuniform points in 1,2, or 3 dimensions, by Hannah Lawrence (2017 summer intern at Flatiron).
4. fsinc: Gaute Hope's fast sinc transform and interpolation python package.
5. FTK: Factorization of the translation kernel for fast rigid image alignment, by Rangan, Spivak, Andén, and Barnett.
6. FINUFFT.jl: a julia language wrapper by Ludvig af Klinteberg (SFU), now using pure julia rather than python.
7. Vineet Bansal's pypi package https://pypi.org/project/finufftpy/. This will be updated soon.

Research output using FINUFFT

1. "Cryo-EM reconstruction of continuous heterogeneity by Laplacian spectral volumes", Amit Moscovich, Amit Halevi, Joakim Andén, and Amit Singer. To appear, Inv. Prob. (2020), https://arxiv.org/abs/1907.01898
2. "A Fast Integral Equation Method for the Two-Dimensional Navier-Stokes Equations", Ludvig af Klinteberg, Travis Askham, and Mary Catherine Kropinski (2019), use FINUFFT 2D type 2. https://arxiv.org/abs/1908.07392
3. "MR-MOTUS: model-based non-rigid motion estimation for MR-guided radiotherapy using a reference image and minimal k-space data", Niek R F Huttinga, Cornelis A T van den Berg, Peter R Luijten and Alessandro Sbrizzi, Phys. Med. Biol. 65(1), 015004. https://arxiv.org/abs/1902.05776
4. Koga, K. "Signal processing approach to mesh refinement in simulations of axisymmetric droplet dynamics", https://arxiv.org/abs/1909.09553 Koga uses 1D FINUFFT to generate a "guideline function" for reparameterizing 1D curves.
5. L. Wang and Z. Zhao, "Two-dimensional tomography from noisy projection tilt series taken at unknown view angles with non-uniform distribution", International Conference on Image Processing (ICIP), (2019).
6. "Factorization of the translation kernel for fast rigid image alignment," Aaditya Rangan, Marina Spivak, Joakim Andén, and Alex Barnett. Inverse Problems 36 (2), 024001 (2020). https://arxiv.org/abs/1905.12317
7. Aleks Donev's group at NYU; ongoing

Papers or codes using our new ES window (spreading) function but not the whole FINUFFT package:

1. Davood Shamshirgar and Anna-Karin Tornberg, "Fast Ewald summation for electrostatic potentials with arbitrary periodicity", exploit our "Barnett-Magland" (BM), aka exp-sqrt (ES) window function. https://arxiv.org/abs/1712.04732
2. Martin Reinecke, codes for radio astronomy reconstruction including https://gitlab.mpcdf.mpg.de/ift/nifty_gridder

Citations to FINUFFT that do not appear to be actual users

1. https://arxiv.org/abs/1903.08365
2. https://arxiv.org/abs/1908.00041
3. https://arxiv.org/abs/1908.00574
4. https://arxiv.org/abs/1912.09746

Acknowledgments

FINUFFT was initiated by Jeremy Magland and Alex Barnett at the Center for Computational Mathematics, Flatiron Institute in early 2017. The main developer and maintainer is:

- Alex Barnett

Major code contributions by:

- Jeremy Magland - early multithreaded spreader, benchmark vs other libraries
- Ludvig af Klinteberg - SIMD vectorization/acceleration of spreader, julia wrapper
- Yu-Hsuan ("Melody") Shih - 2d1many, 2d2many vectorized interface, GPU version
- Andrea Malleo - guru interface prototype and tests
- Libin Lu - guru Fortran, python, MATLAB/octave, julia interfaces

Other significant code contributions by:

- Joakim Andén - catching bugs, MATLAB/FFTW issues, performance tests, python
- Leslie Greengard and June-Yub Lee - CMCL Fortran test drivers
- Dan Foreman-Mackey - early python wrappers
- David Stein - python wrappers
- Vineet Bansal - pypy packaging

Testing, bug reports, helpful discussions:

- Hannah Lawrence - user testing and finding bugs
- Marina Spivak - Fortran testing
- Hugo Strand - python bugs
- Amit Moscovich - Mac OSX build
- Dylan Simon - sphinx help
- Zydrunas Gimbutas - explanation that NFFT uses Kaiser-Bessel backwards
- Charlie Epstein - help with analysis of kernel Fourier transform sums
- Christian Muller - optimization (CMA-ES) for early kernel design
- Andras Pataki - complex number speed in C++
- Timo Heister - pass/fail numdiff testing ideas
- Vladimir Rokhlin - piecewise polynomial approximation on complex boxes

We are also indebted to the authors of other NUFFT codes such as NFFT3, CMCL NUFFT, MIRT, BART, etc, upon whose interfaces, code, and algorithms we have built.

References

Please cite the following two papers if you use this software:

[FIN] A parallel non-uniform fast Fourier transform library based on an "exponential of semicircle" kernel. A. H. Barnett, J. F. Magland, and L. af Klinteberg. SIAM J. Sci. Comput. 41(5), C479-C504 (2019). arxiv version

[B20] Aliasing error of the $\exp(\beta\sqrt{1-z^2})$ kernel in the nonuniform fast Fourier transform. A. H. Barnett. submitted, Appl. Comput. Harmon. Anal. (2020). arxiv version

Background references

For the Kaiser–Bessel kernel and the related PSWF, see:

[KK] Chapter 7. System Analysis By Digital Computer. F. Kuo and J. F. Kaiser. Wiley (1967).

[FT] K. Fourmont. Schnelle Fourier-Transformation bei nichtäquidistanten Gittern und tomographische Anwendungen. PhD thesis, Univ. Münster, 1999.

[F] Non-equispaced fast Fourier transforms with applications to tomography. K. Fourmont. J. Fourier Anal. Appl. 9(5) 431-450 (2003).

[FS] Nonuniform fast Fourier transforms using min-max interpolation. J. A. Fessler and B. P. Sutton. IEEE Trans. Sig. Proc., 51(2):560-74, (Feb. 2003)

[ORZ] Prolate Spheroidal Wave Functions of Order Zero: Mathematical Tools for Bandlimited Approximation. A. Osipov, V. Rokhlin, and H. Xiao. Springer (2013).

[KKP] Using NFFT3—a software library for various nonequispaced fast Fourier transforms. J. Keiner, S. Kunis and D. Potts. Trans. Math. Software 36(4) (2009).

[DFT] How exponentially ill-conditioned are contiguous submatrices of the Fourier matrix? A. H. Barnett, submitted, SIAM Rev. (2020). arxiv version

The appendix of the last of the above contains the first known published proof of the Kaiser–Bessel Fourier transform pair.

FINUFFT builds upon the CMCL NUFFT, and the Fortran wrappers are very similar to its interfaces. For that, the following are references:

[GL] Accelerating the Nonuniform Fast Fourier Transform. L. Greengard and J.-Y. Lee. SIAM Review 46, 443 (2004).

[LG] The type 3 nonuniform FFT and its applications. J.-Y. Lee and L. Greengard. J. Comput. Phys. 206, 1 (2005).

Inversion of the NUFFT is covered in [KKP] above and in:

[GLI] The fast sinc transform and image reconstruction from nonuniform samples in $\mathbf{k}$-space. L. Greengard, J.-Y. Lee and S. Inati, Commun. Appl. Math. Comput. Sci (CAMCOS) 1(1) 121-131 (2006).

The original NUFFT analysis using truncated Gaussians is (the second improving upon the first):

[DR] Fast Fourier Transforms for Nonequispaced data. A. Dutt and V. Rokhlin. SIAM J. Sci. Comput. 14, 1368 (1993).

[S] A note on fast Fourier transforms for nonequispaced grids. G. Steidl, Adv. Comput. Math. 9, 337-352 (1998).

Talk slides

These PDF slides may be a useful introduction.

N