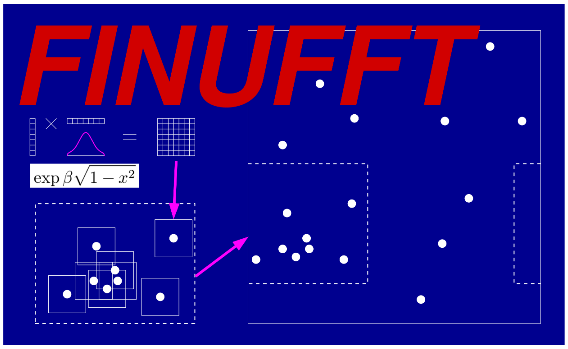
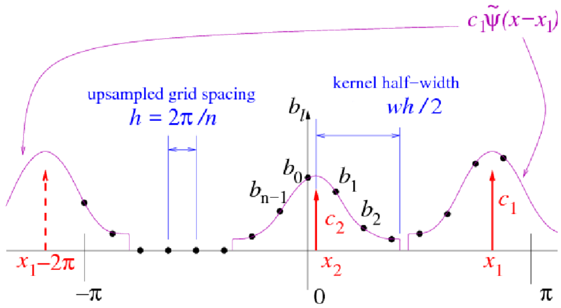
finufft Documentation

Release 1.1.2

Alex Barnett and Jeremy Magland

June 05, 2020





FINUFFT is a set of libraries to compute efficiently three types of nonuniform fast Fourier transform (NUFFT) to a specified precision, in one, two, or three dimensions, on a multi-core shared-memory machine. The library has a very simple interface, does not need any precomputation step, is written in C++ (using OpenMP and FFTW), and has wrappers to C, fortran, MATLAB, octave, and python. As an example, given M arbitrary real numbers x_j and complex numbers c_j, with $j = 1, \ldots, M$, and a requested integer number of modes N, the 1D type-1 (aka "adjoint") transform evaluates the N numbers

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

The x_j can be interpreted as nonuniform source locations, c_j as source strengths, and f_k then as the kth Fourier series coefficient of the distribution $f(x) = \sum_{j=1}^{M} c_j \delta(x-x_j)$. Such exponential sums are needed in many applications in science and engineering, including signal processing, imaging, diffraction, and numerical partial differential equations. The naive CPU effort to evaluate (1) is $O(NM)$. The library approximates (1) to a requested relative precision ϵ with nearly linear effort $O(M \log(1/\epsilon) + N \log N)$. Thus the speedup over the naive cost is similar to that achieved by the FFT. This is achieved by spreading onto a regular grid using a carefully chosen kernel, followed by an upsampled FFT, then a division (deconvolution) step. For the 2D and 3D definitions, and other types of transform, see below.

The FINUFFT library achieves its speed via several innovations including:

1. The use of a new spreading kernel that is provably close to optimal, yet faster to evaluate than the Kaiser-Bessel kernel
2. Quadrature approximation for the Fourier transform of the spreading kernel
3. Load-balanced multithreading of the type-1 spreading operation

For the same accuracy in 3D, the library is 3-50 times faster on a single core than the single-threaded fast Gaussian gridding CMCL libraries of Greengard-Lee, and in the multi-core setting for spreading-dominated problems is faster than the Chemnitz NFFT3 library even when the latter is allowed a RAM-intensive full precomputation of the kernel. This is especially true for highly non-uniform point distributions and/or high precision. Our library does not require precomputation, and uses minimal RAM.

For the case of small problems where repeated NUFFTs are needed with a fixed set of nonuniform points, we have started to build advanced interfaces for this case. These are a factor of 2 or more faster than repeated calls to the plain interface, since certain costs such as FFTW setup and sorting are performed only once.

Note: For very small repeated problems (less than 10000 input and output points), users should also consider a dense matrix-matrix multiplication against the NUDFT matrix using BLAS3 (eg ZGEMM). Since we did not want BLAS to be a dependency, we have not yet included this option.

CHAPTER

ONE

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` (if you are stuck on python v2), or `python3` and `pip3` (for the standard python v3). 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 (assuming python3 as opposed to python):

```
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 python3
```

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 python3` 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 `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.) Note that the very first test run is `test/finufft1d_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/finufft1d_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.

Note that the library includes fortran interfaces defined in `fortran/finufft_f.h`.

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

You may want to make the library for other data types. Currently library names are distinct for single precision (`libfinufftf`) vs double (`libfinufft`). However, 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.

You *must* do at least `make objclean` before changing precision or openmp options.

Single precision: append `PREC=SINGLE` to the make task. Single-precision saves half the RAM, and increases speed slightly (<20%). The C++, C, and fortran demos are all tested in single precision. However, it will break matlab, octave, python interfaces.

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

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:

```
pip3 install numpy pybind11
```

You may then do `make python3` which calls pip3 for the install then runs some tests. An additional test you could do is:

```
python3 python_tests/run_speed_tests.py
```

In all the above, the suffix "3" should be omitted if you either want to work with python v2, or you are using a virtual python environment (see below).

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`. You then should use `make python` instead of `make python3` in the above.

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/
```

CHAPTER

TWO

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 $\mathbf{x}_j \in \mathbb{R}^d$, $j = 1, \ldots, M$. 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

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

where

$$K_{N_i} := \left\{ \begin{array}{ll} \{-N_i/2, \ldots, N_i/2 - 1\}, & N_i \text{ even}, \\ \{-(N_i - 1)/2, \ldots, (N_i - 1)/2\}, & N_i \text{ odd}. \end{array} \right.$$

For instance, $K_{10} = \{-5, -4, \ldots, 4\}$, whereas $K_{11} = \{-5, -4, \ldots, 5\}$. 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

$$f_{\mathbf{k}} := \sum_{j=1}^{M} c_j e^{\pm i \mathbf{k} \cdot \mathbf{x}_j} \qquad \text{for } \mathbf{k} \in K \tag{2.1}$$

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

$$c_j := \sum_{\mathbf{k} \in K} f_{\mathbf{k}} e^{\pm i \mathbf{k} \cdot \mathbf{x}_j} \qquad \text{for } j = 1, \ldots, M \tag{2.2}$$

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 $\mathbf{x}_j \in \mathbb{R}^d$, $j = 1, \ldots, M$, be NU locations, with strengths $c_j \in \mathbb{C}$, and let $\mathbf{s}_k$, $k = 1, \ldots, N$ be NU frequencies. Then the type-3 transform evaluates:

$$f_{\mathbf{k}} := \sum_{j=1}^{M} c_j e^{\pm i \mathbf{s}_k \cdot \mathbf{x}_j} \qquad \text{for } k = 1, \ldots, N \tag{2.3}$$

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.

CHAPTER

THREE

CONTENTS OF THE PACKAGE

- `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 (hackers please help)
- `makefile` : GNU makefile (there are no makefiles in subdirectories)
- `src` : main library source and headers. Compiled objects will be built here
- `lib` : dynamic library will be built here
- `lib-static` : static library will be built here
- `test` : validation and performance tests, bash scripts driving compiled C++
  - `test/check_finufft.sh` is the main pass-fail validation bash script
  - `test/nufft testnd.sh` is a simple uniform-point performance test bash script
  - `test/results` : validation comparison outputs (*.refout; do not remove these), and local test and performance outputs (*.out; you may remove these)
- `examples` : simple example codes for calling the library from C++ and C
- `fortran` : wrappers and drivers for Fortran (see `fortran/README`)
- `matlab` : wrappers and examples for MATLAB/octave
- `finufftpy` : python wrappers
- `python_tests` : accuracy and speed tests and examples using the python wrappers
- `setup.py` : needed so pip or pip3 can build and install the python wrappers
- `contrib` : 3rd-party code

CHAPTER

FOUR

USAGE AND INTERFACES

Here we describe calling FINUFFT from C++, C, and Fortran.

We provide Type 1 (nonuniform to uniform), Type 2 (uniform to nonuniform), and Type 3 (nonuniform to nonuniform), in dimensions 1, 2, and 3. This gives nine basic routines. There are also two *advanced interfaces* for multiple 2d1 and 2d2 transforms with the same point locations.

Using the library is a matter of filling your input arrays, allocating the correct output array size, possibly setting fields in the options struct, then calling one of the transform routines below.

> **Warning:** FINUFFT (when compiled with OpenMP) by default uses all available threads, which is often twice the number of cores (full hyperthreading). We have observed that a large thread count can lead to *reduced* performance, presumably because RAM access is the limiting factor. We recommend that one limit the number of threads at most around 24. This can be done in linux via the shell environment, eg `OMP_NUM_THREADS=16`, or using OpenMP commands in the various languages.

Interfaces from C++

We first give a simple example of performing a 1D type-1 transform in double precision from C++, the library's native language, using C++ complex number type. First include the headers:

```
#include "finufft.h"
#include <complex>
using namespace std;
```

Now in the body of the code, assuming `M` has been set to be the number of nonuniform points, we allocate the input arrays:

```
double *x = (double *)malloc(sizeof(double)*M);
complex<double>* c = (complex<double>*)malloc(sizeof(complex<double>)*M);
```

These arrays should now be filled with the user's data: values in `x` should lie in $[-3\pi, 3\pi]$, and `c` can be arbitrary complex strengths (we omit example code for this here). With `N` as the number of modes, allocate the output array:

```
complex<double>* F = (complex<double>*)malloc(sizeof(complex<double>)*N);
```

Before use, set default values in the options struct `opts`:

```
nufft_opts opts; finufft_default_opts(&opts);
```

> **Warning:**
> - Without this call options may take on random values which may cause a crash.
> - This usage has changed from version 1.0 which used C++-style pass by reference. Please make sure you pass a *pointer* to *opts*.

To perform the nonuniform FFT is then one line:

```
int ier = finufft1d1(M,x,c,+1,1e-6,N,F,opts);
```

This fills `F` with the output modes, in increasing ordering from `-N/2` to `N/2-1`. Here `+1` sets the sign of `i` in the exponentials in the *definitions*, `1e-6` chooses 6-digit relative tolerance, and `ier` is a status output which is zero if successful (see below). See `example1d1.cpp`, in the `examples` directory, for a simple full working example. Then to compile, linking to the double-precision static library, use eg:

```
g++ example1d1.cpp -o example1d1 -I FINUFFT/src FINUFFT/lib-static/libfinufft.a -fopenmp -lfftw3_omp
```

where `FINUFFT` denotes the top-level directory of the installed library. The `examples` and `test` directories are good places to see further usage examples. The documentation for all nine routines follows below.

Note: If you have a small-scale 2D task (say less than 10^5 points or modes) with multiple strength or coefficient vectors but fixed nonuniform points, see the *advanced interfaces*.

Data types

There are certain data type names that we found convenient to unify the interfaces. These are used throughout the below.

- `FLT` : this means `double` if compiled in the default double-precision, or `float` if compiled in single precision. This is used for all real-valued input and output arrays.
- `CPX` : means `complex<double>` in double precision, or `complex<float>` in single precision. This is used for all complex-valued input and output arrays. In the documentation this is often referred to as `complex FLT`.
- `BIGINT` : this is the signed integer type used for all potentially-large input arguments, such as `M` and `N` in the example above. It is defined to the signed 64-bit integer type `int64_t`, allowing the number of input points and/or output modes to exceed 2^31 (around 2 billion). Internally, the `BIGINT` type is also used for all relevant indexing; we have not noticed a slow-down relative to using 32-bit integers (the advanced user could explore this by changing its definition in `finufft.h` and recompiling). This is also referred to as `int64` in the documentation.
- `int` : (in contrast to the above) is the usual 32-bit signed integer, and is used for flags (such as the value `+1` used above) and the output error code.

Options

You may override the default options in `opts` by changing the fields in this struct, after setting up with default values as above. This allows control of various parameters such as the mode ordering, FFTW plan mode, upsampling factor σ, and debug/timing output. Here is the list of the options fields you may set (see the header `src/finufft.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: CMCL-style increasing mode ordering (neg to pos), or
                    // 1: FFT-style mode ordering (affects type-1,2 only)
FLT upsampfac;      // upsampling ratio sigma, either 2.0 (standard) or 1.25 (small FFT)
```

Here are their default settings (set in `src/common.cpp:finufft_default_opts`):

```
debug = 0;
spread_debug = 0;
spread_sort = 2;
spread_kerevalmeth = 1;
spread_kerpad = 1;
chkbnds = 0;
fftw = FFTW_ESTIMATE;
modeord = 0;
upsampfac = (FLT)2.0;
```

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`.

Error codes

In the interfaces, the returned value is 0 if successful, otherwise the error code has the following meanings (see `src/defs.h`):

```
1  requested tolerance epsilon too small
2  attemped to allocate internal arrays larger than MAX_NF (defined in defs.h)
3  spreader: fine grid too small compared to spread width
4  spreader: if chkbnds=1, a nonuniform point 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)
8  upsampfac not a value with known Horner eval: currently 2.0 or 1.25 only
9  ndata not valid in "many" interface (should be >= 1)
```

1D transforms

Now we list the calling sequences for the main C++ codes. Please refer to the above *data types*. (Some comments not referring to the interface have been removed; if you want detail about the algorithms, please see comments in code.)

```
int finufft1d1(BIGINT nj,FLT* xj,CPX* cj,int iflag,FLT eps,BIGINT ms,
               CPX* fk, nufft_opts opts)

 Type-1 1D complex nonuniform FFT.

            nj-1
   fk(k1) = SUM cj[j] exp(+/-i k1 xj(j))  for -ms/2 <= k1 <= (ms-1)/2
            j=0
 Inputs:
   nj     number of sources (int64, aka BIGINT)
   xj     location of sources (size-nj FLT array), in [-3pi,3pi]
   cj     size-nj FLT complex array of source strengths
          (ie, stored as 2*nj FLTs interleaving Re, Im).
   iflag  if >=0, uses + sign in exponential, otherwise - sign (int)
   eps    precision requested (>1e-16)
   ms     number of Fourier modes computed, may be even or odd (int64);
          in either case the mode range is integers lying in [-ms/2, (ms-1)/2]
   opts   struct controlling options (see finufft.h)
 Outputs:
   fk     size-ms FLT complex array of Fourier transform values
          stored as alternating Re & Im parts (2*ms FLTs),
          order determined by opts.modeord.
   returned value - 0 if success, else see ../docs/usage.rst



int finufft1d2(BIGINT nj,FLT* xj,CPX* cj,int iflag,FLT eps,BIGINT ms,
               CPX* fk, nufft_opts opts)

 Type-2 1D complex nonuniform FFT.

   cj[j] = SUM   fk[k1] exp(+/-i k1 xj[j])      for j = 0,...,nj-1
           k1
   where sum is over -ms/2 <= k1 <= (ms-1)/2.

 Inputs:
   nj     number of targets (int64, aka BIGINT)
   xj     location of targets (size-nj FLT array), in [-3pi,3pi]
   fk     complex Fourier transform values (size ms, ordering set by opts.modeord)
          (ie, stored as 2*nj FLTs interleaving Re, Im).
   iflag  if >=0, uses + sign in exponential, otherwise - sign (int).
   eps    precision requested (>1e-16)
   ms     number of Fourier modes input, may be even or odd (int64);
          in either case the mode range is integers lying in [-ms/2, (ms-1)/2]
   opts   struct controlling options (see finufft.h)
 Outputs:
   cj     complex FLT array of nj answers at targets
   returned value - 0 if success, else see ../docs/usage.rst



int finufft1d3(BIGINT nj,FLT* xj,CPX* cj,int iflag, FLT eps, BIGINT nk,
               FLT* s, CPX* fk, nufft_opts opts)
```

```
Type-3 1D complex nonuniform FFT.

            nj-1
  fk[k]  =  SUM   c[j] exp(+-i s[k] xj[j]),      for k = 0, ..., nk-1
            j=0
Inputs:
  nj     number of sources (int64, aka BIGINT)
  xj     location of sources on real line (nj-size array of FLT)
  cj     size-nj FLT complex array of source strengths
         (ie, stored as 2*nj FLTs interleaving Re, Im).
  nk     number of frequency target points (int64)
  s      frequency locations of targets in R.
  iflag  if >=0, uses + sign in exponential, otherwise - sign (int)
  eps    precision requested (>1e-16)
  opts   struct controlling options (see finufft.h)
Outputs:
  fk     size-nk FLT complex Fourier transform values at target
         frequencies sk
  returned value - 0 if success, else see ../docs/usage.rst
```

2D transforms

```
int finufft2d1(BIGINT nj,FLT* xj,FLT *yj,CPX* cj,int iflag,
             FLT eps, BIGINT ms, BIGINT mt, CPX* fk, nufft_opts opts)

 Type-1 2D complex nonuniform FFT.

                  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.

   The output array is k1 (fast), then k2 (slow), with each dimension
   determined by opts.modeord.
   If iflag>0 the + sign is used, otherwise the - sign is used,
   in the exponential.

 Inputs:
   nj     number of sources (int64, aka BIGINT)
   xj,yj     x,y locations of sources (each a size-nj FLT array) in [-3pi,3pi]
   cj     size-nj complex FLT array of source strengths,
          (ie, stored as 2*nj FLTs interleaving Re, Im).
   iflag  if >=0, uses + sign in exponential, otherwise - sign (int)
   eps    precision requested (>1e-16)
   ms,mt  number of Fourier modes requested in x and y (int64);
          each may be even or odd;
          in either case the mode range is integers lying in [-m/2, (m-1)/2]
   opts   struct controlling options (see finufft.h)
 Outputs:
   fk     complex FLT array of Fourier transform values
          (size ms*mt, fast in ms then slow in mt,
          ie Fortran ordering).
   returned value - 0 if success, else see ../docs/usage.rst
```

```
int finufft2d2(BIGINT nj,FLT* xj,FLT *yj,CPX* cj,int iflag,FLT eps,
             BIGINT ms, BIGINT mt, CPX* fk, nufft_opts opts)

 Type-2 2D complex nonuniform FFT.

   cj[j] =  SUM   fk[k1,k2] exp(+/-i (k1 xj[j] + k2 yj[j]))      for j = 0,...,nj-1
           k1,k2
   where sum is over -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2,

 Inputs:
   nj     number of targets (int64, aka BIGINT)
   xj,yj     x,y locations of targets (each a size-nj FLT array) in [-3pi,3pi]
   fk     FLT complex array of Fourier transform values (size ms*mt,
          increasing fast in ms then slow in mt, ie Fortran ordering).
          Along each dimension the ordering is set by opts.modeord.
   iflag  if >=0, uses + sign in exponential, otherwise - sign (int)
   eps    precision requested (>1e-16)
   ms,mt  numbers of Fourier modes given in x and y (int64)
          each may be even or odd;
          in either case the mode range is integers lying in [-m/2, (m-1)/2].
   opts   struct controlling options (see finufft.h)
 Outputs:
   cj     size-nj complex FLT array of target values
          (ie, stored as 2*nj FLTs interleaving Re, Im).
   returned value - 0 if success, else see ../docs/usage.rst



int finufft2d3(BIGINT nj,FLT* xj,FLT* yj,CPX* cj,int iflag, FLT eps,
               BIGINT nk, FLT* s, FLT *t, CPX* fk, nufft_opts opts)

 Type-3 2D complex nonuniform FFT.

             nj-1
   fk[k]  =  SUM   c[j] exp(+-i (s[k] xj[j] + t[k] yj[j]),    for k=0,...,nk-1
             j=0
 Inputs:
   nj     number of sources (int64, aka BIGINT)
   xj,yj  x,y location of sources in the plane R^2 (each size-nj FLT array)
   cj     size-nj complex FLT array of source strengths,
          (ie, stored as 2*nj FLTs interleaving Re, Im).
   nk     number of frequency target points (int64)
   s,t    (k_x,k_y) frequency locations of targets in R^2.
   iflag  if >=0, uses + sign in exponential, otherwise - sign (int)
   eps    precision requested (>1e-16)
   opts   struct controlling options (see finufft.h)
 Outputs:
   fk     size-nk complex FLT Fourier transform values at the
          target frequencies sk
   returned value - 0 if success, else see ../docs/usage.rst
```

3D transforms

```
int finufft3d1(BIGINT nj,FLT* xj,FLT *yj,FLT *zj,CPX* cj,int iflag,
             FLT eps, BIGINT ms, BIGINT mt, BIGINT mu, CPX* fk,
             nufft_opts opts)
```

```
Type-1 3D complex nonuniform FFT.

                  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.

  The output array is as in opt.modeord in each dimension.
  k1 changes is fastest, k2 middle,
  and k3 slowest, ie Fortran ordering. If iflag>0 the + sign is
  used, otherwise the - sign is used, in the exponential.

Inputs:
  nj     number of sources (int64, aka BIGINT)
  xj,yj,zj   x,y,z locations of sources (each size-nj FLT array) in [-3pi,3pi]
  cj     size-nj complex FLT array of source strengths,
         (ie, stored as 2*nj FLTs interleaving Re, Im).
  iflag  if >=0, uses + sign in exponential, otherwise - sign (int)
  eps    precision requested
  ms,mt,mu  number of Fourier modes requested in x,y,z (int64);
         each may be even or odd;
         in either case the mode range is integers lying in [-m/2, (m-1)/2]
  opts   struct controlling options (see finufft.h)
Outputs:
  fk     complex FLT array of Fourier transform values (size ms*mt*mu,
         changing fast in ms to slowest in mu, ie Fortran ordering).
  returned value - 0 if success, else see ../docs/usage.rst



int finufft3d2(BIGINT nj,FLT* xj,FLT *yj,FLT *zj,CPX* cj,
             int iflag,FLT eps, BIGINT ms, BIGINT mt, BIGINT mu,
             CPX* fk, nufft_opts opts)

Type-2 3D complex nonuniform FFT.

  cj[j] =    SUM   fk[k1,k2,k3] exp(+/-i (k1 xj[j] + k2 yj[j] + k3 zj[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

Inputs:
  nj     number of sources (int64, aka BIGINT)
  xj,yj,zj     x,y,z locations of targets (each size-nj FLT array) in [-3pi,3pi]
  fk     FLT complex array of Fourier series values (size ms*mt*mu,
         increasing fastest in ms to slowest in mu, ie Fortran ordering).
         (ie, stored as alternating Re & Im parts, 2*ms*mt*mu FLTs)
         Along each dimension, opts.modeord sets the ordering.
  iflag  if >=0, uses + sign in exponential, otherwise - sign (int)
  eps    precision requested
  ms,mt,mu  numbers of Fourier modes given in x,y,z (int64);
         each may be even or odd;
         in either case the mode range is integers lying in [-m/2, (m-1)/2].
  opts   struct controlling options (see finufft.h)
Outputs:
```

```
   cj     size-nj complex FLT array of target values,
          (ie, stored as 2*nj FLTs interleaving Re, Im).
   returned value - 0 if success, else see ../docs/usage.rst



int finufft3d3(BIGINT nj,FLT* xj,FLT* yj,FLT *zj, CPX* cj,
             int iflag, FLT eps, BIGINT nk, FLT* s, FLT *t,
             FLT *u, CPX* fk, nufft_opts opts)

 Type-3 3D complex nonuniform FFT.

             nj-1
   fk[k]  =  SUM   c[j] exp(+-i (s[k] xj[j] + t[k] yj[j] + u[k] zj[j]),
             j=0
                        for k=0,...,nk-1
 Inputs:
   nj     number of sources (int64, aka BIGINT)
   xj,yj,zj   x,y,z location of sources in R^3 (each size-nj FLT array)
   cj     size-nj complex FLT array of source strengths
          (ie, interleaving Re & Im parts)
   nk     number of frequency target points (int64)
   s,t,u      (k_x,k_y,k_z) frequency locations of targets in R^3.
   iflag  if >=0, uses + sign in exponential, otherwise - sign (int)
   eps    precision requested (FLT)
   opts   struct controlling options (see finufft.h)
 Outputs:
   fk     size-nk complex FLT array of Fourier transform values at the
          target frequencies sk
   returned value - 0 if success, else see ../docs/usage.rst
```

Interfaces from C

From C one calls the same routines as for C++, and includes the same header files (this unified interface is new as of version 1.1). To recap, one should `#include "finufft.h"` then, as above, initialize the options:

> nufft_opts opts; finufft_default_opts(&opts);

Options fields may then be changed in `opts` before calling `finufft?d?` (where the wildcard `?` denotes an appropriate number).

As above, `FLT` indicates `double` or `float`, but now `CPX` indicates their complex C99-type equivalents (see `src/finufft.h` for the definitions used). For examples see `examples/example1d1c.c` (double precision) and `examples/example1d1cf.c` (single precision).

Interfaces from fortran

We have not yet included control of the options in the fortran wrappers. (Please help create these if you want a simple user project!) The meaning of arguments is as in the C++ documentation above, apart from that now `ier` is an argument which is output to. Examples of calling the basic 9 routines from fortran are in `fortran/nufft?d_demo.f` (for double-precision) and `fortran/nufft?d_demof.f` (single-precision). `fortran/nufft2dmany_demo.f` shows how to use the many-vector interface. Here are the calling commands with fortran types for the default double-precision case (the simple-precision case is analogous)

```
integer ier,iflag,ms,mt,mu,nj,ndata
real*8, allocatable :: xj(:),yj(:),zj(:), sk(:),tk(:),uk(:)
real*8 err,eps
complex*16, allocatable :: cj(:), fk(:)

call finufft1d1_f(nj,xj,cj,iflag,eps, ms,fk,ier)
call finufft1d2_f(nj,xj,cj,iflag, eps, ms,fk,ier)
call finufft1d3_f(nj,xj,cj,iflag,eps, ms,sk,fk,ier)
call finufft2d1_f(nj,xj,yj,cj,iflag,eps,ms,mt,fk,ier)
call finufft2d1many_f(ndata,nj,xj,yj,cj,iflag,eps,ms,mt,fk,ier)
call finufft2d2_f(nj,xj,yj,cj,iflag,eps,ms,mt,fk,ier)
call finufft2d2many_f(ndata,nj,xj,yj,cj,iflag,eps,ms,mt,fk,ier)
call finufft2d3_f(nj,xj,yj,cj,iflag,eps,nk,sk,tk,fk,ier)
call finufft3d1_f(nj,xj,yj,zj,cj,iflag,eps,ms,mt,mu,fk,ier)
call finufft3d2_f(nj,xj,yj,zj,cj,iflag,eps,ms,mt,mu,fk,ier)
call finufft3d3_f(nj,xj,yj,zj,cj,iflag,eps,nk,sk,tk,uk,fk,ier)
```

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>` (typedef'ed 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.

CHAPTER

FIVE

ADVANCED INTERFACES FOR MANY VECTORS WITH SAME NONUNIFORM POINTS

It is common to need repeated NUFFTs with a fixed set of nonuniform points, but different strength or mode coefficient vectors. For large problems, performing sequential plain calls is efficient (although there would be a slight benefit to sorting only once), but when the problem size is smaller, certain start-up costs cause repeated calls to the plain interface to be slower than necessary. In particular, we note that FFTW takes around 0.1 ms per thread to look up stored wisdom, which for small problems (of order 10000 or less input and output data) can, sadly, dominate the runtime. Thus we include interfaces, described here, for multiple stacked strength or coefficient vectors with the same nonuniform points.

These have only been implemented for the 2d1 and 2d2 types so far, for which there are applications in cryo-EM.

For data types in the below, please see *data types*.

2D transforms

```
int finufft2d1many(int ndata, BIGINT nj, FLT* xj, FLT *yj, CPX* c, int iflag,
                   FLT eps, BIGINT ms, BIGINT mt, CPX* fk, nufft_opts opts)

Type-1 2D complex nonuniform FFT for multiple strength vectors, same NU pts.

                   nj
  f[k1,k2,d] =    SUM  c[j,d] exp(+-i (k1 x[j] + k2 y[j]))
                  j=1

  for -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2, d = 0,...,ndata-1

  The output array is in increasing k1 ordering (fast), then increasing
  k2 ordering (slow), then increasing d (slowest). If iflag>0 the + sign
  is used, otherwise the - sign is used, in the exponential.
Inputs:
  ndata  number of data
  nj     number of sources (int64, aka BIGINT)
  xj,yj  x,y locations of sources (each a size-nj FLT array) in [-3pi,3pi]
  c      a size nj*ndata complex FLT array of source strengths,
         increasing fast in nj then slow in ndata.
  iflag  if >=0, uses + sign in exponential, otherwise - sign.
  eps    precision requested (>1e-16)
  ms,mt  number of Fourier modes requested in x and y; each may be even or odd;
         in either case the mode range is integers lying in [-m/2, (m-1)/2].
         ms*mt must not exceed 2^31.
   opts  struct controlling options (see finufft.h)
Outputs:
  fk     complex FLT array of Fourier transform values
```

```
        (size ms*mt*ndata, increasing fast in ms then slow in mt then in ndata
        ie Fortran ordering).
  returned value - 0 if success, else see ../docs/usage.rst

  Note: nthreads times the RAM is needed, so this is good only for small problems.



int finufft2d2many(int ndata, BIGINT nj, FLT* xj, FLT *yj, CPX* c, int iflag,
                   FLT eps, BIGINT ms, BIGINT mt, CPX* fk, nufft_opts opts)

Type-2 2D complex nonuniform FFT for multiple coeff vectors, same NU pts.

  cj[j,d] =  SUM   fk[k1,k2,d] exp(+/-i (k1 xj[j] + k2 yj[j]))
           k1,k2
  for j = 0,...,nj-1,  d = 0,...,ndata-1
  where sum is over -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2

Inputs:
  ndata  number of mode coefficient vectors
  nj     number of targets (int64, aka BIGINT)
  xj,yj  x,y locations of targets (each a size-nj FLT array) in [-3pi,3pi]
  fk     FLT complex array of Fourier transform values (size ms*mt*ndata,
         increasing fast in ms then slow in mt then in ndata, ie Fortran
         ordering). Along each dimension the ordering is set by opts.modeord.
  iflag  if >=0, uses + sign in exponential, otherwise - sign (int)
  eps    precision requested (>1e-16)
  ms,mt  numbers of Fourier modes given in x and y
         each may be even or odd;
         in either case the mode range is integers lying in [-m/2, (m-1)/2].
         ms*mt must not exceed 2^31.
  opts   struct controlling options (see finufft.h)
Outputs:
  cj     size-nj*ndata complex FLT array of target values, (ie, stored as
         2*nj*ndata FLTs interleaving Re, Im), increasing fast in nj then
         slow in ndata.
  returned value - 0 if success, else see ../docs/usage.rst

  Note: nthreads times the RAM is needed, so this is good only for small problems.
```

Design notes

After extensive timing tests, we settled on blocking up the ndata vectors into blocks of size nthreads (the available thread number). Each block is handled together via FFTW and OpenMP parallelism. For instance, for type-1:

1. Each thread calls a single-threaded spreader, reusing a precomputed sorted index list.
2. Apply FFT on nthreads vectors of data using FFTW's "many dft" interface.
3. Each thread calls a single-threaded deconvolve function.

This requires ndata times the RAM overhead than the plain interface.

It would also be possible to call multi-threaded spreading, sequentially on each data vector; we found this slower in all cases, and so close to repeated calls to the plain interface as to not be useful.

For repeated small problems where the nonuniform points and strengths or coefficients change, but the mode grid is fixed, reusing the FFTW plan may still be beneficial; this would require a three-call "plan, execute, destroy" interface which we have not considered worth building yet.

MATLAB/OCTAVE INTERFACES

```
----------------------------------------------------------------------
FINUFFT1D1

[f ier] = finufft1d1(x,c,isign,eps,ms)
[f ier] = finufft1d1(x,c,isign,eps,ms,opts)

Type-1 1D complex nonuniform FFT.

               nj
    f(k1) =  SUM c[j] exp(+/-i k1 x(j))  for -ms/2 <= k1 <= (ms-1)/2
              j=1
  Inputs:
    x     location of sources on interval [-3pi,3pi], length nj
    c     size-nj complex array of source strengths
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps     precision requested (>1e-16)
    ms     number of Fourier modes computed, may be even or odd;
           in either case the mode range is integers lying in [-ms/2, (ms-1)/2]
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default).
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
  Outputs:
    f     size-ms double complex array of Fourier transform values
    ier - 0 if success, else:
          1 : eps too small
          2 : size of arrays to malloc exceed MAX_NF
          other codes: as returned by cnufftspread
----------------------------------------------------------------------------
FINUFFT1D2

[c ier] = finufft1d2(x,isign,eps,f)
[c ier] = finufft1d2(x,isign,eps,f,opts)

Type-2 1D complex nonuniform FFT.

   c[j] = SUM   f[k1] exp(+/-i k1 x[j])      for j = 1,...,nj
           k1
    where sum is over -ms/2 <= k1 <= (ms-1)/2.

 Inputs:
    x     location of NU targets on interval [-3pi,3pi], length nj
```

```
    f     complex Fourier transform values
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps    precision requested (>1e-16)
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default).
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
 Outputs:
    c     complex double array of nj answers at targets
    ier - 0 if success, else:
          1 : eps too small
          2 : size of arrays to malloc exceed MAX_NF
          other codes: as returned by cnufftspread
c = complex(zeros(nj,1));   % todo: change all output to inout & prealloc...
---------------------------------------------------------------------------
FINUFFT1D3

[f ier] = finufft1d3(x,c,isign,eps,s)
[f ier] = finufft1d3(x,c,isign,eps,s,opts)

             nj
    f[k]  =  SUM   c[j] exp(+-i s[k] x[j]),      for k = 1, ..., nk
             j=1
  Inputs:
    x      location of NU sources in R (real line).
    c      size-nj double complex array of source strengths
    s      frequency locations of NU targets in R.
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps    precision requested (>1e-16)
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
  Outputs:
    f     size-nk double complex Fourier transform values at target
           frequencies s
    returned value - 0 if success, else:
                     1 : eps too small
                     2 : size of arrays to malloc exceed MAX_NF
------------------------------------------------------------------------
FINUFFT2D1

[f ier] = finufft2d1(x,y,c,isign,eps,ms,mt)
[f ier] = finufft2d1(x,y,c,isign,eps,ms,mt,opts)

Type-1 2D complex nonuniform FFT.

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

    for -ms/2 <= k1 <= (ms-1)/2,  -mt/2 <= k2 <= (mt-1)/2.

  Inputs:
```

```
    x,y   locations of NU sources on the square [-3pi,3pi]^2, each length nj
    c     size-nj complex array of source strengths
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps     precision requested (>1e-16)
    ms,mt  number of Fourier modes requested in x & y; each may be even or odd
          in either case the mode range is integers lying in [-m/2, (m-1)/2]
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default).
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
  Outputs:
    f     size (ms*mt) double complex array of Fourier transform values
          (ordering given by opts.modeord in each dimension, ms fast, mt slow)
    ier - 0 if success, else:
          1 : eps too small
          2 : size of arrays to malloc exceed MAX_NF
          other codes: as returned by cnufftspread
-------------------------------------------------------------------------
FINUFFT2D1MANY

[f ier] = finufft2d1many(x,y,c,isign,eps,ms,mt)
[f ier] = finufft2d1many(x,y,c,isign,eps,ms,mt,opts)

Type-1 2D complex nonuniform FFT

                    nj
    f[k1,k2,d] =   SUM  c[j,d] exp(+-i (k1 x[j] + k2 y[j]))
                   j=1

    for -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2, d = 1, ..., ndata

  Inputs:
    x,y   locations of NU sources on the square [-3pi,3pi]^2, each length nj
    c     size-(nj,ndata) complex array of source strengths
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps     precision requested (>1e-16)
    ms,mt  number of Fourier modes requested in x & y; each may be even or odd
          in either case the mode range is integers lying in [-m/2, (m-1)/2]
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default).
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
  Outputs:
    f     size (ms,mt,ndata) double complex array of Fourier transform values
          (ordering given by opts.modeord in each dimension, ms fast, mt slow)
    ier - 0 if success, else:
          1 : eps too small
          2 : size of arrays to malloc exceed MAX_NF
          other codes: as returned by cnufftspread

Note: nthreads copies of the fine grid are allocated, limiting this to smaller
 problem sizes.
```

```
--------------------------------------------------------------------------
FINUFFT2D2

[c ier] = finufft2d2(x,y,isign,eps,f)
[c ier] = finufft2d2(x,y,isign,eps,f,opts)

Type-2 2D complex nonuniform FFT.

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

 Inputs:
    x,y   location of NU targets on the square [-3pi,3pi]^2, each length nj
    f     size (ms,mt) complex Fourier transform value matrix
          (mode ordering given by opts.modeord in each dimension)
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps    precision requested (>1e-16)
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default).
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
 Outputs:
    c     complex double array of nj answers at the targets.
    ier - 0 if success, else:
          1 : eps too small
          2 : size of arrays to malloc exceed MAX_NF
          other codes: as returned by cnufftspread
--------------------------------------------------------------------------
FINUFFT2D2MANY

[c ier] = finufft2d2many(x,y,isign,eps,f)
[c ier] = finufft2d2many(x,y,isign,eps,f,opts)

Type-2 2D complex nonuniform FFT.

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

 Inputs:
    x,y   location of NU targets on the square [-3pi,3pi]^2, each length nj
    f     size (ms,mt,ndata) complex Fourier transform value matrix
          (mode ordering given by opts.modeord in each dimension)
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps    precision requested (>1e-16)
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default).
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
 Outputs:
```

```
    c     complex double array of nj*ndata answers at the targets.
    ier - 0 if success, else:
          1 : eps too small
          2 : size of arrays to malloc exceed MAX_NF
          other codes: as returned by cnufftspread

Note: nthreads copies of the fine grid are allocated, limiting this to smaller
 problem sizes.
---------------------------------------------------------------------------
FINUFFT2D3

[f ier] = finufft2d3(x,y,c,isign,eps,s,t)
[f ier] = finufft2d3(x,y,c,isign,eps,s,t,opts)

             nj
    f[k]  =  SUM   c[j] exp(+-i (s[k] x[j] + t[k] y[j])),  for k = 1, ..., nk
             j=1
  Inputs:
    x,y    location of NU sources in R^2, each length nj.
    c      size-nj double complex array of source strengths
    s,t    frequency locations of NU targets in R^2.
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps    precision requested (>1e-16)
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
  Outputs:
    f     size-nk double complex Fourier transform values at target
           frequencies s,t
    returned value - 0 if success, else:
                     1 : eps too small
                     2 : size of arrays to malloc exceed MAX_NF
------------------------------------------------------------------------
FINUFFT3D1

[f ier] = finufft3d1(x,y,z,c,isign,eps,ms,mt,mu)
[f ier] = finufft3d1(x,y,z,c,isign,eps,ms,mt,mu,opts)

Type-1 3D complex nonuniform FFT.

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

    for -ms/2 <= k1 <= (ms-1)/2,  -mt/2 <= k2 <= (mt-1)/2,
        -mu/2 <= k3 <= (mu-1)/2.

  Inputs:
    x,y,z locations of NU sources on [-3pi,3pi]^3, each length nj
    c     size-nj complex array of source strengths
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps     precision requested (>1e-16)
    ms,mt,mu number of Fourier modes requested in x,y and z; each may be
          even or odd.
          In either case the mode range is integers lying in [-m/2, (m-1)/2]
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
```

```
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default).
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
  Outputs:
    f     size (ms*mt*mu) double complex array of Fourier transform values
           (ordering given by opts.modeord in each dimension, ms fastest, mu
            slowest).
    ier - 0 if success, else:
          1 : eps too small
          2 : size of arrays to malloc exceed MAX_NF
          other codes: as returned by cnufftspread
---------------------------------------------------------------------------
FINUFFT3D2

[c ier] = finufft3d2(x,y,z,isign,eps,f)
[c ier] = finufft3d2(x,y,z,isign,eps,f,opts)

Type-2 3D complex nonuniform FFT.

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

 Inputs:
    x,y,z location of NU targets on cube [-3pi,3pi]^3, each length nj
    f     size (ms,mt,mu) complex Fourier transform value matrix
          (ordering given by opts.modeord in each dimension; ms fastest to mu
           slowest).
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps    precision requested (>1e-16)
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default).
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
 Outputs:
    c     complex double array of nj answers at the targets.
    ier - 0 if success, else:
          1 : eps too small
          2 : size of arrays to malloc exceed MAX_NF
          other codes: as returned by cnufftspread
---------------------------------------------------------------------------
FINUFFT3D3

[f ier] = finufft3d3(x,y,z,c,isign,eps,s,t,u)
[f ier] = finufft3d3(x,y,z,c,isign,eps,s,t,u,opts)

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

```
  Inputs:
    x,y,z  location of NU sources in R^3, each length nj.
    c      size-nj double complex array of source strengths
    s,t,u   frequency locations of NU targets in R^3.
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps    precision requested (>1e-16)
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
  Outputs:
    f     size-nk double complex Fourier transform values at target
           frequencies s,t,u
    returned value - 0 if success, else:
                     1 : eps too small
                     2 : size of arrays to malloc exceed MAX_NF
```

A note on integer sizes: In Matlab/MEX, mwrap uses `int` types, so that output arrays can only be <2^31. However, input arrays >=2^31 have been tested, and while they don't crash, they result in wrong answers (all zeros). This has yet to be fixed (please help; an updated version of mwrap might be needed).

For a full list of error codes see *Error codes*.

CHAPTER SEVEN

PYTHON INTERFACE

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*, *isign*, *eps*, *ms*, *f*, *debug=0*, *spread_debug=0*, *spread_sort=2*, *fftw=0*, *modeord=0*, *chkbnds=1*, *upsampfac=2.0*)

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]*) – 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]
- **f** (*complex[ms]*) – output Fourier mode values. Should be initialized as a numpy array of the correct size
- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown).
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (prints spreader info)

- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (do sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **modeord** (*int, optional*) – 0 (CMCL increasing mode ordering), 1 (FFT ordering)
- **chkbnds** (*int, optional*) – 0 (don't check NU points valid), 1 (do)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

Note: The output is written into the f array.

Returns 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF, 4 at least one NU point out of range (if chkbnds true)

Return type int

Example

see `python_tests/demo1d1.py`

finufftpy.**nufft1d2** (*x, c, isign, eps, f, debug=0, spread_debug=0, spread_sort=2, fftw=0, modeord=0, chkbnds=1, upsampfac=2.0*)

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]
- **c** (*complex[nj]*) – output values at targets. Should be initialized as a numpy array of the correct size
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)
- **f** (*complex[ms]*) – Fourier mode coefficients, where ms is even or odd In either case the mode indices are integers in [-ms/2, (ms-1)/2]
- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown)
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (print spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **modeord** (*int, optional*) – 0 (CMCL increasing mode ordering), 1 (FFT ordering)
- **chkbnds** (*int, optional*) – 0 (don't check NU points valid), 1 (do)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

Note: The output is written into the c array.

Returns 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF, 4 at least one NU point out of range (if chkbnds true)

Return type int

Example

see `python_tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft1d3`**(*x*, *c*, *isign*, *eps*, *s*, *f*, *debug=0*, *spread_debug=0*, *spread_sort=2*, *fftw=0*, *upsampfac=2.0*)

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]*) – 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
- **f** (*complex[nk]*) – output values at target frequencies. Should be initialized as a numpy array of the correct size
- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown)
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (print spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

Note: The output is written into the f array.

Returns 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF

Return type int

Example

see `python_tests/accuracy_speed_tests.py` finufftpy.**nufft2d1**(*x*, *y*, *c*, *isign*, *eps*, *ms*, *mt*, *f*, *debug=0*, *spread_debug=0*, *spread_sort=2*, *fftw=0*, *modeord=0*, *chkbnds=1*, *upsampfac=2.0*)

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]*) – 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]
- **f** (*complex[ms,mt]*) – output Fourier mode values. Should be initialized as a Fortran-ordered (ie ms fast, mt slow) numpy array of the correct size
- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown)
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (prints spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **modeord** (*int, optional*) – 0 (CMCL increasing mode ordering), 1 (FFT ordering)
- **chkbnds** (*int, optional*) – 0 (don't check NU points valid), 1 (do)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

Note: The output is written into the f array.

Returns 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF, 4 at least one NU point out of range (if chkbnds true)

Return type int

Example

see python/tests/accuracy_speed_tests.py

finufftpy.**nufft2d1many**(*x*, *y*, *c*, *isign*, *eps*, *ms*, *mt*, *f*, *debug=0*, *spread_debug=0*, *spread_sort=2*, *fftw=0*, *modeord=0*, *chkbnds=1*, *upsampfac=2.0*)

2D type-1 (aka adjoint) complex nonuniform fast Fourier transform, for multiple strength vectors with same nonuniform points.

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

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,ndata]*) – 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]
- **f** (*complex[ms,mt,ndata]*) – output Fourier mode values. Should be initialized as a Fortran-ordered (ie ms fast, mt slow) numpy array of the correct size
- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown)
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (prints spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **modeord** (*int, optional*) – 0 (CMCL increasing mode ordering), 1 (FFT ordering)
- **chkbnds** (*int, optional*) – 0 (don't check NU points valid), 1 (do)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

Note: The outputs are written into the f array.

For small problems this routine will be faster than repeated calls to nufft2d1.

Nthreads copies of the fine grid are allocated, limiting this to smaller problem sizes than the plain 2d1 interface.

Returns 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF, 4 at least one NU point out of range (if chkbnds true)

Return type int

Example

see `python/tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft2d2`**(*x, y, c, isign, eps, f, debug=0, spread_debug=0, spread_sort=2, fftw=0, modeord=0, chkbnds=1, upsampfac=2.0*)

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]
- **c** (*complex[nj]*) – output values at targets. Should be initialized as a numpy array of the correct size
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)
- **f** (*complex[ms,mt]*) – 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.
- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown)
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (print spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **modeord** (*int, optional*) – 0 (CMCL increasing mode ordering), 1 (FFT ordering)
- **chkbnds** (*int, optional*) – 0 (don't check NU points valid), 1 (do)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

Note: The output is written into the c array.

Returns 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF, 4 at least one NU point out of range (if chkbnds true)

Return type int

Example

see `python_tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft2d2many`** (*x, y, c, isign, eps, f, debug=0, spread_debug=0, spread_sort=2, fftw=0, modeord=0, chkbnds=1, upsampfac=2.0*)

2D type-2 (aka forward) complex nonuniform fast Fourier transform, for multiple coefficient vectors with same nonuniform points.

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

for j = 0,...,nj-1, and d = 0,...,ndata-1
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]
- **c** (*complex[nj,ndata]*) – output values at targets. Should be initialized as a Fortran-ordered numpy array of the correct size
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)
- **f** (*complex[ms,mt,ndata]*) – 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.
- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown)
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (print spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **modeord** (*int, optional*) – 0 (CMCL increasing mode ordering), 1 (FFT ordering)
- **chkbnds** (*int, optional*) – 0 (don't check NU points valid), 1 (do)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

Note: The outputs are written into the c array.

For small problems this routine will be faster than repeated calls to nufft2d2.

Nthreads copies of the fine grid are allocated, limiting this to smaller problem sizes than the plain 2d2 interface.

Returns 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF, 4 at least one NU point out of range (if chkbnds true)

Return type int

Example

see `python_tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft2d3`**(*x, y, c, isign, eps, s, t, f, debug=0, spread_debug=0, spread_sort=2, fftw=0, upsampfac=2.0*)

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]*) – 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
- **f** (*complex[nk]*) – output values at target frequencies. Should be initialized as a numpy array of the correct size
- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown)
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (print spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

Note: The output is written into the f array.

Returns 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF

Return type int

Example

see `python_tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft3d1`**(*x, y, z, c, isign, eps, ms, mt, mu, f, debug=0, spread_debug=0, spread_sort=2, fftw=0, modeord=0, chkbnds=1, upsampfac=2.0*)

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]*) – 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]
- **f** (*complex[ms,mt,mu]*) – output Fourier mode values. Should be initialized as a Fortran-ordered (ie ms fastest) numpy array of the correct size
- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown)
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (prints spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **modeord** (*int, optional*) – 0 (CMCL increasing mode ordering), 1 (FFT ordering)
- **chkbnds** (*int, optional*) – 0 (don't check NU points valid), 1 (do)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

Note: The output is written into the f array.

Returns 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF, 4 at least one NU point out of range (if chkbnds true)

Return type int

Example

see `python_tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft3d2`** (*x, y, z, c, isign, eps, f, debug=0, spread_debug=0, spread_sort=2, fftw=0, modeord=0, chkbnds=1, upsampfac=2.0*)

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]
- **c** (*complex[nj]*) – output values at targets. Should be initialized as a numpy array of the correct size
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)

- **f** (*complex[ms,mt,mu]*) – 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.
- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown)
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (print spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **modeord** (*int, optional*) – 0 (CMCL increasing mode ordering), 1 (FFT ordering)
- **chkbnds** (*int, optional*) – 0 (don't check NU points valid), 1 (do)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

Note: The output is written into the c array.

Returns 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF, 4 at least one NU point out of range (if chkbnds true)

Return type int

Example

see `python_tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft3d3`**(*x, y, z, c, isign, eps, s, t, u, f, debug=0, spread_debug=0, spread_sort=2, fftw=0, upsampfac=2.0*)

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]*) – 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
- **f** (*complex[nk]*) – output values at target frequencies. Should be initialized as a numpy array of the correct size

- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown)
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (print spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

Note: The output is written into the f array.

Returns 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF

Return type int

Example

see `python_tests/accuracy_speed_tests.py`

CHAPTER

EIGHT

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).

CHAPTER

NINE

APPLICATIONS AND EXAMPLES

For further applications, see the References, and these PDF slides.

1. Periodic Poisson solve on non-Cartesian quadrature grid

As a warm-up, it is standard that FFT can be used 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:

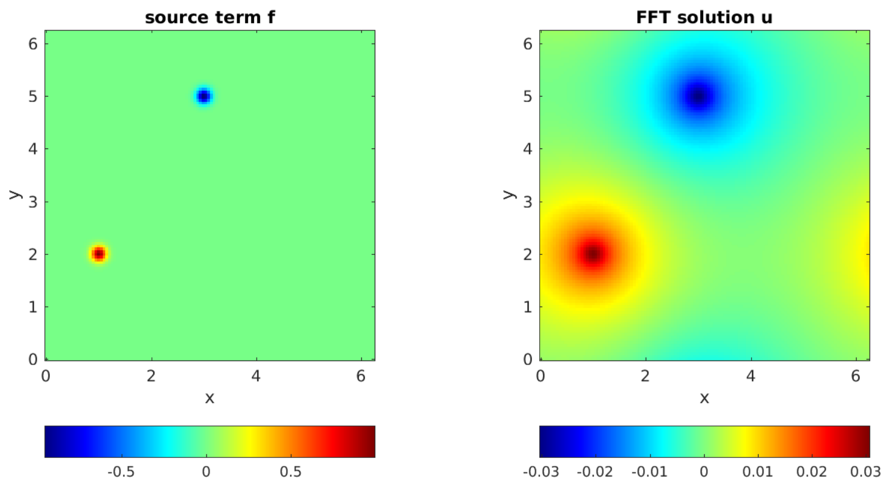
```
ns = 40:20:120;              % convergence study of grid points per side
for i=1:numel(ns), n = ns(i);
  x = 2*pi*(0:n-1)/n;        % grid
  [xx yy] = ndgrid(x,x);     % ordering: x fast, y slow
  f = src(xx,yy);            % eval source on grid
  fhat = ifft2(f);           % step 1: Fourier coeffs 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);    % inverse -Laplacian 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 $x_j \in \mathbb{R}^d$, $j = 1, \ldots, M$, and weights w_j such that, for all smooth functions f,

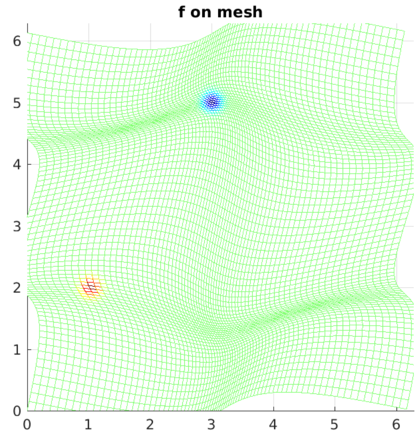
$$\int_{[0,2\pi)^d} f(x)dx \approx \sum_{j=1}^{M} f(x_j) w_j$$

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');
```

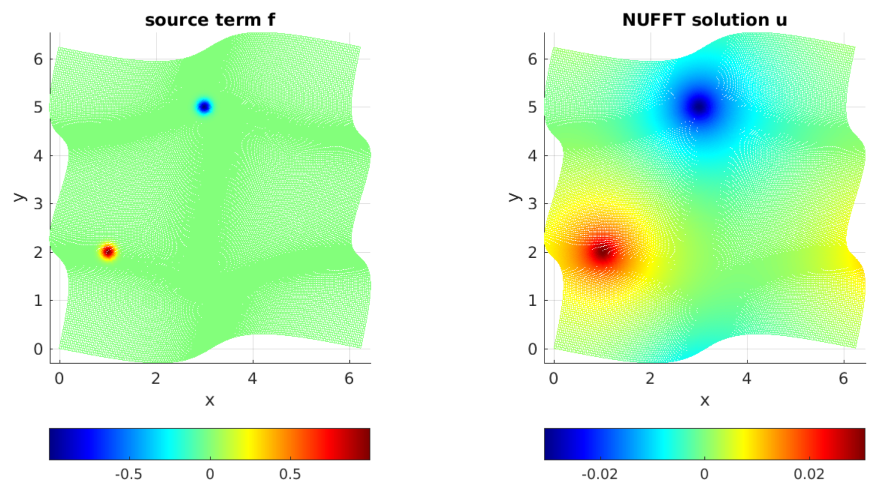


To solve on this grid, replace step 1 above by evaluating the Euler-Fourier formula using the quadrature scheme, which needs a type-1 NUFFT, and step 3 (evaluation on the nonuniform grid) by a type-2 NUFFT. Step 2 (the frequency filter) remains the same. Here is the demo code:

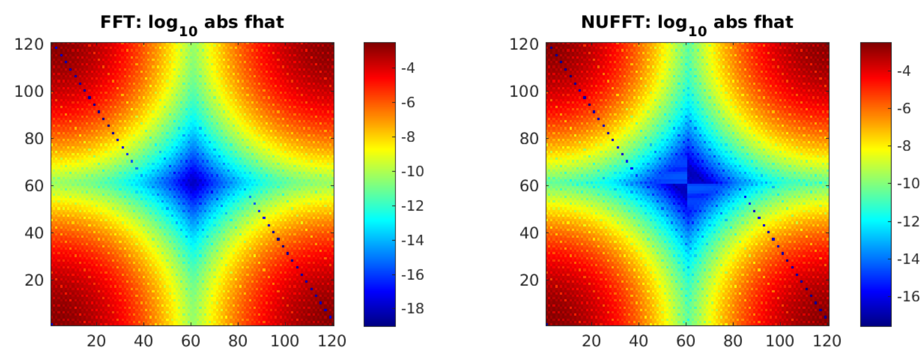
```
tol = 1e-12;            % NUFFT precision
ns = 80:40:240;         % convergence study of grid points per side
for i=1:numel(ns), n = ns(i);
  t = 2*pi*(0:n-1)/n;   % 1d unif grid
  [tt ss] = ndgrid(t,t);
  xxx = map(tt(:)',ss(:)');
  xx = reshape(xxx(1,:),[n n]); yy = reshape(xxx(2,:),[n n]);  % 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 filtered 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.

CHAPTER

TEN

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.

CHAPTER

ELEVEN

KNOWN ISSUES

One should also check the github issues for the project page, https://github.com/flatironinstitute/finufft/issues

Also see notes in the `TODO` file.

Issues with library

- When requested accuracy is `1e-14` or less, it is sometimes not possible to match this, especially when there are a large number of input and/or output points. This is believed to be unavoidable round-off error.
- Currently in Mac OSX, `make lib` fails to make the shared object library (.so).
- The timing of the first FFTW call is complicated, depending on whether FFTW_ESTIMATE (the default) or FFTW_MEASURE is used. Such issues are known, and discussed in other documentation, eg https://pythonhosted.org/poppy/fft_optimization.html . We would like to find a way of pre-storing some Intel-specific FFTW plans (as MATLAB does) to avoid the large FFTW planning times.
- Currently, a single library name is used for single- and multi-threaded versions. Thus, i) you need to `make clean` before changing such make options, and ii) if you wish to maintain multiple such versions you need to move them around and maintain them yourself, eg by duplicating the directory.
- The overhead for small problem sizes (<10000 data points) is too high, due to things such as the delay in FFTW looking up pre-stored wisdom. A unified advanced interface with a plan stage is in the works.

Issues with interfaces

- MATLAB, octave and python cannot exceed input or output data sizes of 2^31.
- MATLAB, octave and python interfaces do not handle single precision.
- Fortran interface does not allow control of options, nor data sizes exceeding 2^31.

Bug reports

If you think you have found a bug, please file an 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, and version of FINUFFT.

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

CHAPTER

TWELVE

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 using our new 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, window function. https://arxiv.org/abs/1712.04732

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

CHAPTER

THIRTEEN

ACKNOWLEDGMENTS

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

- Alex Barnett

Major code contributions by:

- Jeremy Magland - multithreaded spreader, benchmark vs other codes
- 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.

CHAPTER FOURTEEN

REFERENCES

References for this software and the underlying mathematics include:

[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:1808.06736v2

[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:2001.09405

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

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

[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)

[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).

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

This code 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).

The original NUFFT analysis using truncated Gaussians is:

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

N