FINUFFT Documentation

Release 2.0.3

Alex Barnett and the FINUFFT team

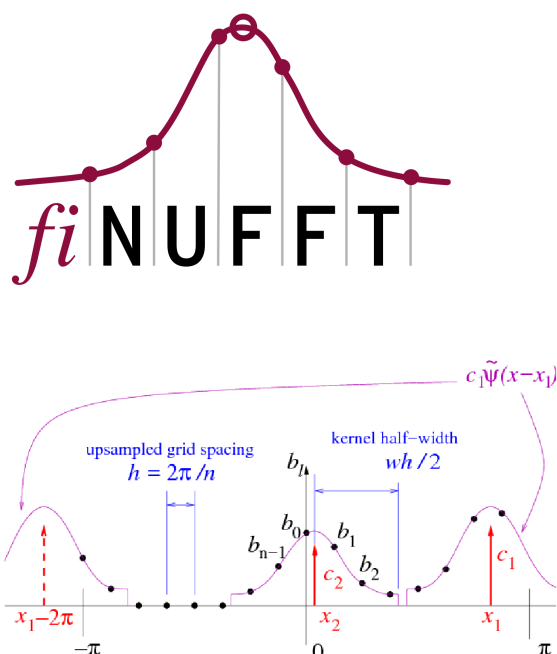**Oct 08, 2021**

CONTENTS

OVERVIEW

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

1.1 What does FINUFFT do?

As an example, given M real numbers $x_j \in [0, 2\pi)$, and complex numbers c_j, with $j = 1, \ldots, M$, and a requested integer number of modes N, FINUFFT can efficiently compute the 1D "type 1" transform, which means to evaluate the N complex outputs

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

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

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

1.2 Why FINUFFT? Features and comparison against other NUFFT libraries

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

- **High speed**. For instance, at similar accuracy, FINUFFT is up to 10x faster than the multi-threaded Chemnitz NFFT3 library, and (in single-thread mode) up to 50x faster than the CMCL NUFFT library. This is achieved via:

    1. a simple new "exponential of semicircle" kernel that is provably close to optimal

    2. quadrature approximation for this kernel's Fourier transform

    3. load-balanced multithreaded spreading/interpolation

    4. bin-sorting of points to improve cache reuse

    5. a low upsampling option for smaller FFTs, especially in type 3 transforms

    6. piecewise polynomial kernel evaluation (additions and multiplications only) that SIMD-vectorizes reliably on open-source compilers

- **Less RAM**. Our kernel is so fast that there is no point in precomputation; it is always evaluated on the fly. Thus our memory footprint is often an order of magnitude less than the fastest (precomputed) modes of competitors such as NFFT3 and MIRT, especially at high accuracy.

- **Automated kernel parameters**. Unlike many competitors, we do not force the user to worry about kernel choice or parameters. The user simply requests a desired relative accuracy, then FINUFFT chooses parameters to target this accuracy as fast as possible.

- **Simplicity**. We provide interfaces that perform a NUFFT with a single command—just like an FFT—from seven common languages/environments. For advanced users we also have "many vector" interfaces that can be much faster than repeated calls to the simple interface with the same points. Finally (like NFFT3) we have a "guru" interface for maximum flexibility, in all of these languages.

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

1.3 Do I even need a NUFFT?

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

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

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

INSTALLATION

Note: If the below instructions fail in any operating system, try the relevant version of our precompiled linux, OSX, and Windows binaries, place it (or them) in your linking path, and try `make test`. We will be adding to these as needed; please email us to contribute or request one.

Note: Python-only users can simply install via `pip install finufft` which downloads a generic binary from PyPI. Only if you prefer a custom compilation, see *below*.

Below we deal with the three standard OSes in order: 1) **linux**, 2) **Mac OSX**, 3) **Windows**. We have some users contributing settings for other OSes, for instance PowerPC. The general procedure to download, then compile for such a special setup is, illustrating with the PowerPC case:

```
git clone https://github.com/flatironinstitute/finufft.git
cd finufft
cp make.inc.powerpc make.inc
make test -j
```

Have a look for `make.inc.*` to see what is available, and/or edit your `make.inc` based on looking in the `makefile` and quirks of your local setup. As of 2021, we have continuous integration which tests the default (linux) settings in this `makefile`, plus those in three OS-specific setup files:

```
make.inc.macosx_clang
make.inc.macosx_gcc-10
make.inc.windows_msys
```

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

2.1 Quick linux install instructions

Make sure you have packages `fftw3` and `fftw3-dev` (or their equivalent on your distro) installed. Then `cd` into your FINUFFT directory and do `make test -j`. 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 segfaults out of 8 tests done
0 fails out of 8 tests done
```

This output repeats for double then single precision (hence, scroll up to check the double also gave no fails). If this fails, see the more detailed instructions below. If it succeeds, please look in `examples/`, `test/`, and the rest of this manual, for examples of how to call and link to the library. Type `make` to see a list of other aspects the user can build (examples, language interfaces, etc).

2.2 Dependencies

This library is fully supported for unix/linux, and partially for Mac OSX for Windows (eg under MSYS or WSL using MinGW compilers).

For the basic libraries you need

- C++ compiler supporting C++14, such `g++` in GCC (version >=5.0), or `clang` (version >=3.4)
- FFTW3 including its development libraries
- GNU `make` and other standard unix/POSIX tools such as `bash`

Optional:

- for Fortran wrappers: compiler such as `gfortran` in GCC
- for MATLAB wrappers: MATLAB (versions at least R2016b up to current work)
- for Octave wrappers: recent Octave version at least 4.4, and its development libraries
- for the python wrappers you will need `python` version at least 3.6 (python 2 is unsupported), with `numpy`.

2.3 1) Linux: tips for installing dependencies and compiling

On a Fedora/CentOS linux system, the base dependencies can be installed by:

```
sudo yum install make gcc gcc-c++ fftw-devel libgomp
```

To add Fortran and Octave language interfaces also do:

```
sudo yum install gcc-gfortran octave octave-devel
```

Note: We are not exactly sure how to install `python3` and `pip3` using `yum`. You may prefer to use `conda` or `virtualenv` to set up a python environment anyway (see bottom).

Alternatively, on Ubuntu linux, base dependencies are:

```
sudo apt-get install make build-essential libfftw3-dev
```

and for Fortran, Python, and Octave language interfaces also do:

```
sudo apt-get gfortran python3 python3-pip octave liboctave-dev
```

In older distros you may have to compile `octave` from source to get the needed >=4.4 version.

You should then compile and test the library via various `make` tasks, eg:

```
make test -j
```

then checking you got `0 fails`. This compiles the main libraries then runs double- and single-precision tests, each of which should report zero segfaults and zero fails.

Note: GCC versions on linux: long-term linux distros ship old GCC versions that may not be C++14 compatible. We recommend that you compile with a recent GCC, at least GCC 7.3 (which we used for benchmarks in 2018 in our SISC paper), or GCC 9+. 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 so poor in Mac OSX (see below).

The make tasks (eg `make lib`) compiles double and single precision functions, which live simultaneously in `libfinufft`, with distinct function names.

The only selectable option at compile time is multithreaded (default, using OpenMP) vs single-threaded (to achieve this append `OMP=OFF` to the make tasks). Since you may always set `opts.nthreads=1` when calling the multithreaded library, the point of having a single-threaded library is mostly for small repeated problems to avoid *any* OpenMP overhead, or for debugging purposes. You *must* do at least `make objclean` before changing this threading option.

Note: By default, neither the multithreaded or single-threaded library (e.g. made by `make lib OMP=OFF`) are thread-safe, due to the FFTW3 plan stage. However, keep reading for the compiler option to fix this if you have a recent FFTW3 version.

Testing. The initial test is `test/basicpassfail` which is the most basic double-precision smoke test, producing the exit code 0 if success, nonzero if fail. You can check the exit code thus:

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

The single-precision version is `test/basicpassfailf`. The make task also runs (`cd test; OMP_NUM_THREADS=4 ./check_finufft.sh`) which is the main validation of the library in double precision, and (`cd test; OMP_NUM_THREADS=4 ./check_finufft.sh SINGLE`) which does it in single precision. Since these call many tiny problem sizes, they will (due to openmp and fftw thread-wise overheads) run much faster with less than the full thread count, explaining our use of 4 threads. Text (and stderr) outputs are written into `test/results/*.out`.

Use `make perftest` for larger spread/interpolation and NUFFT tests taking 10-20 seconds. This writes log files 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.

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

High-level interfaces. See *below* for python compilation.

`make matlab` to compile the MEX interface to matlab, then within MATLAB add the `matlab` directory to your path, cd to `matlab/test` and run `check_finufft` which should run for 5 secs and print a bunch of errors around `1e-6`.

Note: If this MATLAB test crashes, it is most likely to do with incompatible versions of OpemMP. Thus, you will want to make (or add to) a file `make.inc` the line:

```
OMPLIBS=/usr/local/MATLAB/R2020a/sys/os/glnxa64/libiomp5.so
```

---

or appropriate to your MATLAB version. You'll want to check this shared object exists. Then `make clean` and `make test -j`, finally `make matlab` again.

`make octave` to compile and test the MEX-like interface to Octave.

2.3.1 Compilation flags and make.inc settings

This is for experts. Here are all the flags that the FINUFFT source responds to. Activate them by adding a line of the form `CXXFLAGS+=-DMYFLAG` in your `make.inc`:

- `-DFFTW_PLAN_SAFE`: This makes FINUFFT call `fftw_make_planner_thread_safe()` as part of its FFTW3 planner stage; see http://www.fftw.org/fftw3_doc/Thread-safety.html. This makes FINUFFT thread-safe. See `examples/threadsafe1d1.cpp`. This is only available in FFTW version >=3.3.6; for this reason it is not yet the default.

- `-DSINGLE`: This is internally used by our build process to switch (via preprocessor macros) the source from double to single precision. You should not need to use this flag yourself.

Here are some other settings that you may need to adjust in `make.inc`:

- Switching to linking tests, examples, etc, with PTHREADS instead of the default OMP version of FFTW, is achieved by inserting into `make.inc` the line

`FFTWOMPSUFFIX = threads`.

2.4 2) Mac OSX: tips for installing dependencies and compiling

Note: A brew package will come shortly; stay tuned. However, the below has been tested on 10.14 (Mojave) with both clang and gcc-8, and 10.15 (Catalina) with clang.

First you'll want to set up Homebrew, as follows. We assume a fresh OSX machine. If you don't have Xcode, install Command Line Tools (which is a few hundred MB download, much smaller than the now 10 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 was 8.2.0 in Mojave, and 10.2.0 in Catalina, in our tests).

If you are python-only, use:

```
brew install python3
pip3 install finufft
```

Or, for experts to compile python interfaces locally using either clang or gcc, see *below*.

Now to compiling the library for C++/C/fortran/MATLAB/octave use. There are now 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.

2.4.1 The clang route (default)

Once you have downloaded FINUFFT from github, go to its top directory. You now need to decide if you will be wanting to call FINUFFT from MATLAB (and currently have MATLAB installed). If so, do:

```
cp make.inc.macosx_clang_matlab make.inc
```

Else if you don't have MATLAB, do:

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

Note: The difference here is the version of OpenMP linked: MATLAB crashes when `gomp` is linked, so for MATLAB users the OpenMP version used by MATLAB must be linked against (`iomp5`), not `gomp`.

Whichever you picked, now try `make test -j`, and clang should compile and you should get `0 fails`.

clang MATLAB setup. Assuming you chose the MATLAB clang variant above, you should now `make matlab`. To test, open MATLAB, `addpath matlab`, `cd matlab/test`, and `check_finufft`, which should complete in around 5 seconds.

Note: Unfortunately OSX+MATLAB+mex is notoriously poorly supported, and you may need to search the web for help on that, then check you are able to compile a simple mex file first. For instance, on Catalina (10.15.6), `make matlab` fails with a warning involving Xcode `license has not been accepted`, and then an error with `no supported compiler was found`. Eventually this property file hack worked, which simply requires typing /usr/libexec/PlistBuddy -c 'Add :IDEXcodeVersionForAgreedToGMLicense string 10.0' ~/ Library/Preferences/com.apple.dt.Xcode.plist Please also read our https://github.com/flatironinstitute/ finufft/issues and if you *are* able to mex compile, but `make matlab` fails, post a new Issue.

Octave interfaces work out of the box (this also runs a self-test):

```
brew install octave
make octave
```

2.4.2 The GCC route

This is less recommended, unless you need to link from `gfortran`, when it appears to be essential. We have tested on Movaje:

```
cp make.inc.macosx_gcc-8 make.inc
make test -j
make fortran
```

which also compiles and tests the fortran interfaces. In Catalina you'll probably need to edit to `g++-10` in your `make. inc`. We find python may be built as *below*. We found that octave interfaces do not work with GCC; please help. 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 have installed, eg via brew. For pre-2016 MATLAB Mac OSX versions you'll instead want to edit the `maci64` section of `mexopts.sh`.

Note: GCC with OSX is only partially supported. Please help us if you can!

2.5 3) Windows: tips for compiling

We have users who have adjusted the makefile to work - at least to some extent - on Windows 10. If you are only interested in calling from Octave (which already comes with MinGW-w64 and FFTW), then we have been told this can be done very simply: from within Octave, go to the `finufft` directory and do `system('make octave')`. You may have to tweak `OCTAVE` in your `make.inc` in a similar fashion to below.

More generally, please make sure to have a recent version of Mingw at hand, preferably with a 64bit version of gnu-make like the WinLibs standalone build of GCC and MinGW-w64 for Windows. Note that most MinGW-w64 distributions, such as TDM-GCC, do not feature the 64bit gnu-make. Fortunately, this limitation is only relevant to run the tests. To prepare the build of the static and dynamic libraries run:

```
copy make.inc.windows_mingw make.inc
```

Subsequently, open this `make.inc` file with the text editor of your choice and assign the parent directories of the FFTW header file to `FFTW_H_DIR`, of the FFTW libraries to `FFTW_LIB_DIR`, and of the GCC OpenMP library lgomp.dll to `LGOMP_DIR`. Note that you need the last-mentioned only if you plan to build the MEX-interface for MATLAB. Now, you should be able to run:

```
make lib
```

If the command `make` cannot be found and the MinGW binaries are part of your system PATH: Keep in mind that the MinGW installation contains only a file called mingw32-make.exe, not make.exe. Create a copy of this file, call it make.exe, and make sure the corresponding parent folder is part of your system PATH. If the library is compiled successfully, you can try to run the tests. Note that your system has to fulfill the following prerequisites to this end: A Linux distribution set up via WSL (has been tested with Ubuntu 20.04 LTS from the Windows Store) and the 64bit gnu-make mentioned before. Further, make sure that the directory containing the FFTW-DLLs is part of your system PATH. Otherwise the executables built will not run. As soon as you have everything set up, run the following command:

```
make test
```

In a similar fashion, the examples can now be build with `make examples`. This rule of the makefile does neither require WSL nor the 64bit gnu-make and should hopefully work out-of-the-box. Finally, it is also possible to build the MEX file needed to call FINUFFT from MATLAB. Since the MinGW support of MATLAB is somewhat limited, you will probably have to define the environment variable `MW_MINGW64_LOC` and assign the path of your MinGW installation. Hint to avoid misunderstandings: The last-mentioned directory contains folders named `bin`, `include`, and `lib` among others. Then, the following command should generate the required MEX-file:

```
make matlab
```

For users who work with Windows using MSYS and MinGW compilers. Please try:

```
cp make.inc.windows_msys make.inc
make test -j
```

We seek help with Windows support. Also see https://github.com/flatironinstitute/finufft/issues

2.6 Building a python interface to a locally compiled library

Recall that the basic user may simply `pip install finufft`, then check it worked via:

```
python3 python/test/run_accuracy_tests.py
```

However, a user or developer may want to build a python wrapper to their locally compiled FINUFFT library, perhaps for more speed. We now describe this, for all OSes. We assume python3 (hence pip3). First make sure you have pip installed, and that you can already compile the C++ library (eg via `make test`). Next make sure you have the required python packages:

```
pip3 install numpy
```

You may then:

```
make python
```

which builds the `finufft` module, installs (in editable mode) via pip, then runs some tests and examples. You will see that the `finufftc` shared object appears in the `python/finufft` directory. An additional performance test you could then do is:

```
python3 python/test/run_speed_tests.py
```

Note: On OSX, if trouble with python with clang: we have found that the above may fail with an error about `-lstdc++`, in which case you should try setting an environment variable like:

```
export MACOSX_DEPLOYMENT_TARGET=10.14
```

where you should replace 10.14 by your OSX number. We have also in the past found that running:

```
pip3 install ./python
```

in the command line can work even when `make python` does not (probably to do with environment variables).

Note: Our new (v2.0.1) python interface is quite different from Dan Foreman-Mackey's original repo that wrapped finufft: python-finufft, or Jeremy Magland's. The interface is simpler, and the existing library is linked to. Under the hood we now use `ctypes` instead of `pybind11`.

2.6.1 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 from a shell in the FINUFFT top directory (after installing `python-virtualenv`):

```
virtualenv -p /usr/bin/python3 env1
source env1/bin/activate
```

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

```
pip install -r python/requirements.txt
```

Then `pip install finufft` or build as above.

DIRECTORIES IN THIS PACKAGE

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

Main library source:

- `makefile` : the single GNU makefile (there are no makefiles in subdirectories)

- `make.inc.*` : system-specific example overrides to use as your `make.inc`

- `src` : main library C++ sources

- `include` : header files, including those for users to compile against

- `contrib` : 3rd-party codes in the main library

- `lib` : dynamic (`.so`) library will be built here

- `lib-static` : static (`.a`) library will be built here

Examples, tutorials, and docs:

- `examples` : simple example codes for calling the library from C++ and C

- `tutorial` : application demo codes (various languages), supporting `docs/tutorial/`

- `finufft-manual.pdf` : the manual (auto-generated by sphinx, eg via `make docs`)

- `docs` : source files for documentation (`.rst` files are human-readable, kinda)

- `README.md` : github-facing (and human text-only reader) welcome

- `LICENSE` : how you may use this software

- `CHANGELOG` : list of changes, release notes

- `TODO` : list of things needed to fix or extend (also see git Issues)

Testing:

- `test` : main validation tests (C++/bash), including:

    - `test/basicpassfail{f}` simple smoke test with exit code

    - `test/check_finufft.sh` is the main pass-fail validation bash script

    - `test/results` has validation comparison outputs (*.refout; do not remove these), and local test outputs (*.out; you may remove these)

- `perftest` : main performance and developer tests (C++/bash), including:

    - `test/spreadtestnd.sh` spread/interp performance test bash script

    - `test/nuffttestnd.sh` NUFFT performance test bash script

Other language interfaces, further testing:

- `fortran` : wrappers and example drivers for Fortran (see `fortran/README`)

- `matlab` : MATLAB/octave wrappers, tests, and examples

- `python` : python wrappers, examples, and tests

    - `python/ci` continuous integration tests

- `julia` : (not yet functional; for now see FINUFFT.jl)

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

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

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

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

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

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

Finally, the **type 3** (NU to NU) transform does not have restrictions on the NU points, and there is no periodicity. Let $\mathbf{x}_j \in \mathbb{R}^d$, $j = 1, \ldots, M$, 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:

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

For all three transforms, the computational effort scales like the product of the space-bandwidth products (real-space width times frequency-space width) in each dimension. For type 1 and type 2 this means near-linear scaling in the total number of modes $N := N_1 \ldots N_d$. 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.

EXAMPLE USAGE FROM C++ AND C

5.1 Quick-start example in C++

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

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

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

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

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

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

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

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

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

Note: FINUFFT works with a periodicity of 2π for type 1 and 2 transforms; see *definitions*. For example, nonuniform points $x = \pm\pi$ are equivalent. Points must lie in the input domain $[-3\pi, 3\pi)$, which allows the user to assume a convenient periodic domain such as $[-\pi, \pi)$ or $[0, 2\pi)$. To handle points outside of $[-3\pi, 3\pi)$ the user must fold them

back into this domain before passing to FINUFFT. FINUFFT does not handle this case, for speed reasons. To use a different periodicity, linearly rescale your coordinates.

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

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

> **Warning:**
>
> - Without the `finufft_default_opts` call, options may take on arbitrary values which may cause a crash.
>
> - Note that, as of version 2.0, `opts` is passed as a pointer in both places.

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

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

```
g++ simple1d1.cpp -o simple1d1 -I$FINUFFT/include $FINUFFT/lib-static/libfinufft.a -
↪fopenmp -lfftw3_omp -lfftw3 -lm
```

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

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

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

5.2 Quick-start example in C

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

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

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

(continues on next page)

```
int N = 1e6;                  // number of modes
double complex* F = (double complex*)malloc(sizeof(double complex)*N);
nufft_opts opts;                        // make an opts struct
finufft_default_opts(&opts);            // set default opts (must do this)
opts.debug = 2;                         // more debug/timing to stdout
int ier = finufft1d1(M,x,c,+1,1e-9,N,F,&opts);

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

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

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

5.3 2D example in C++

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

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

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

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

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

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

See `examples/simple2d1.cpp` for an example with a math check, to insure that the mode indexing is correctly understood.

5.4 Vectorized interface example

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

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

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

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

5.5 Guru interface example

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

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

```cpp
// (assume x, y, c are filled, and F allocated, as in the 2D code above...)
int type=1, dim=2, ntrans=1;
int64_t Ns[] = {1000,2000};                 // N1,N2 as 64-bit int array
// step 1: make a plan...
finufft_plan plan;
int ier = finufft_makeplan(type, dim, Ns, +1, ntrans, 1e-6, &plan, NULL);
// step 2: send in pointers to M nonuniform points (just x, y in this case)...
finufft_setpts(plan, M, &x[0], &y[0], NULL, 0, NULL, NULL, NULL);
// (user should not change x, y nonuniform point arrays here!)
// step 3: do the planned transform to the c strength data, output to F...
finufft_execute(plan, &c[0], &F[0]);
// ... you could now send in new points, and/or do transforms with new c data
```

(continues on next page)

```
// ...
// step 4: when done, free the memory used by the plan...
finufft_destroy(plan);
```

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

> **Warning:** You must not change the nonuniform point arrays (here `x`, `y`) between passing them to `finufft_setpts` and performing `finufft_execute`. The latter call expects these arrays to be unchanged. We chose this style of interface since it saves RAM and time (by avoiding unnecessary duplication), allowing the largest possible problems to be solved.

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

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

5.6 Thread safety and global state

It is possible to call FINUFFT from within multithreaded code, e.g. in an OpenMP parallel block. In this case `opts.nthreads=1` should be set, and FINUFFT must have been compiled with the `-DFFTW_PLAN_SAFE` flag, making it thread-safe. For demos of this, see

- `examples/threadsafe1d1` which runs a 1D type-1 separately on each thread, checking the math, and

- `examples/threadsafe2d2f` which runs a 2D type-2 on each slice, which are parallelized over via an OpenMP parallel for loop (without any math check, just dummy inputs)

which are both built by `make examples` if the above flag has been set.

Note: A design decision of FFTW is to have a global state which stores wisdom and settings. Such global state can cause unforeseen effects on other routines that also use FFTW. In contrast, FINUFFT uses pointers to plans to store its state, and does not have a global state (other than one `static` flag used as a lock on FFTW initialization in the FINUFFT plan stage). This means different FINUFFT calls should not affect each other, although they may affect other codes that use FFTW via FFTW's global state.

DOCUMENTATION OF ALL C++ FUNCTIONS

All functions have double-precision (`finufft`) and single-precision (`finufftf`) versions. Do not forget this `f` suffix in the latter case. We group the simple and vectorized interfaces together, by each of the nine transform types (dimensions 1,2,3, and types 1,2,3). The guru interface functions are defined at the end. You will also want to refer to the *options* and *error codes* which apply to all 46 routines.

A reminder on Fourier mode ordering; see *modeord*. For example, if `N1=8` in a 1D type 1 or type 2 transform:

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

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

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

In all of the simple, vectorized, and plan functions below you may either pass NULL as the last options argument to use default options, or a pointer to a valid `nufft_opts` struct. In this latter case you will first need to create an options struct then set default values by passing a pointer (here `opts`) to the following:

```
void finufft_default_opts(nufft_opts* opts)
void finufftf_default_opts(nufft_opts* opts)

 Set values in a NUFFT options struct to their default values.
```

Be sure to use the first version for double-precision and the second for single-precision. You may then change options with, for example, `opts->debug=1;` and then pass `opts` to the below routines.

6.1 Simple and vectorized interfaces

The "simple" interfaces (the first two listed in each block) perform a single transform, whereas the "vectorized" (the last two listed in each block, with the word "many" in the function name) perform `ntr` transforms with the same set of nonuniform points but stacked complex strengths or coefficients vectors.

Note: The motivations for the vectorized interface (and guru interface, see below) are as follows. 1) It is more efficient to bin-sort the nonuniform points only once if there are not to change between transforms. 2) For small problems, certain start-up costs cause repeated calls to the simple 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.

6.1.1 1D transforms

```
int finufft1d1(int64_t M, double* x, complex<double>* c, int iflag, double eps, int64_t
N1, complex<double>* f, nufft_opts* opts)
int finufftf1d1(int64_t M, float* x, complex<float>* c, int iflag, float eps, int64_t N1,
complex<float>* f, nufft_opts* opts)

int finufft1d1many(int ntr, int64_t M, double* x, complex<double>* c, int iflag, double
eps, int64_t N1, complex<double>* f, nufft_opts* opts)
int finufftf1d1many(int ntr, int64_t M, float* x, complex<float>* c, int iflag, float
eps, int64_t N1, complex<float>* f, nufft_opts* opts)

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

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

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

  Inputs:
    ntr    how many transforms (only for vectorized "many" functions, else ntr=1)
    M      number of nonuniform point sources
    x      nonuniform points in [-3pi,3pi) (length M real array)
    c      source strengths (size M*ntr complex array)
    iflag  if >=0, uses +i in complex exponential, otherwise -i
    eps    desired relative precision; smaller is slower. This can be chosen
           from 1e-1 down to ~ 1e-14 (in double precision) or 1e-6 (in single)
    N1     number of output Fourier modes to be computed
    opts   pointer to options struct (see opts.rst), or NULL for defaults

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

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

```
int finufft1d2(int64_t M, double* x, complex<double>* c, int iflag, double eps, int64_t
N1, complex<double>* f, nufft_opts* opts)
int finufftf1d2(int64_t M, float* x, complex<float>* c, int iflag, float eps, int64_t N1,
complex<float>* f, nufft_opts* opts)

int finufft1d2many(int ntr, int64_t M, double* x, complex<double>* c, int iflag, double
eps, int64_t N1, complex<double>* f, nufft_opts* opts)
int finufftf1d2many(int ntr, int64_t M, float* x, complex<float>* c, int iflag, float
eps, int64_t N1, complex<float>* f, nufft_opts* opts)

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

```
Computes to precision eps, via a fast algorithm, one or more transforms of the form:

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


Inputs:
  ntr     how many transforms (only for vectorized "many" functions, else ntr=1)
  M       number of nonuniform point targets
  x       nonuniform points in [-3pi,3pi) (length M real array)
  iflag   if >=0, uses +i in complex exponential, otherwise -i
  eps     desired relative precision; smaller is slower. This can be chosen
          from 1e-1 down to ~ 1e-14 (in double precision) or 1e-6 (in single)
  N1      number of input Fourier modes
  f       Fourier mode coefficients (size N1*ntr complex array)
  opts    pointer to options struct (see opts.rst), or NULL for defaults

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

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

```
int finufft1d3(int64_t M, double* x, complex<double>* c, int iflag, double eps, int64_t
N, double* s, complex<double>* f, nufft_opts* opts)
int finufftf1d3(int64_t M, float* x, complex<float>* c, int iflag, float eps, int64_t N,
float* s, complex<float>* f, nufft_opts* opts)


int finufft1d3many(int ntr, int64_t M, double* x, complex<double>* c, int iflag, double
eps, int64_t N, double* s, complex<double>* f, nufft_opts* opts)
int finufftf1d3many(int ntr, int64_t M, float* x, complex<float>* c, int iflag, float
eps, int64_t N, float* s, complex<float>* f, nufft_opts* opts)

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

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

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

Inputs:
  ntr     how many transforms (only for vectorized "many" functions, else ntr=1)
  M       number of nonuniform point sources
  x       nonuniform points in R (length M real array)
  c       source strengths (size M*ntr complex array)
  iflag   if >=0, uses +i in complex exponential, otherwise -i
  eps     desired relative precision; smaller is slower. This can be chosen
```

```
              from 1e-1 down to ~ 1e-14 (in double precision) or 1e-6 (in single)
    N        number of nonuniform frequency targets
    s        nonuniform frequency targets in R (length N real array)
    opts    pointer to options struct (see opts.rst), or NULL for defaults

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

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

6.1.2 2D transforms

```
int finufft2d1(int64_t M, double* x, double* y, complex<double>* c, int iflag, double
eps, int64_t N1, int64_t N2, complex<double>* f, nufft_opts* opts)
int finufftf2d1(int64_t M, float* x, float* y, complex<float>* c, int iflag, float eps,
int64_t N1, int64_t N2, complex<float>* f, nufft_opts* opts)

int finufft2d1many(int ntr, int64_t M, double* x, double* y, complex<double>* c, int
iflag, double eps, int64_t N1, int64_t N2, complex<double>* f, nufft_opts* opts)
int finufftf2d1many(int ntr, int64_t M, float* x, float* y, complex<float>* c, int iflag,
float eps, int64_t N1, int64_t N2, complex<float>* f, nufft_opts* opts)

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

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

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

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

  Inputs:
    ntr    how many transforms (only for vectorized "many" functions, else ntr=1)
    M      number of nonuniform point sources
    x,y    nonuniform point coordinates in [-3pi,3pi]^2 (length M real arrays)
    c      source strengths (size M*ntr complex array)
    iflag  if >=0, uses +i in complex exponential, otherwise -i
    eps    desired relative precision; smaller is slower. This can be chosen
            from 1e-1 down to ~ 1e-14 (in double precision) or 1e-6 (in single)
    N1     number of output Fourier modes to be computed (x direction)
    N2     number of output Fourier modes to be computed (y direction)
    opts   pointer to options struct (see opts.rst), or NULL for defaults

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

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

```
int finufft2d2(int64_t M, double* x, double* y, complex<double>* c, int iflag, double
eps, int64_t N1, int64_t N2, complex<double>* f, nufft_opts* opts)
int finufftf2d2(int64_t M, float* x, float* y, complex<float>* c, int iflag, float eps,
int64_t N1, int64_t N2, complex<float>* f, nufft_opts* opts)

int finufft2d2many(int ntr, int64_t M, double* x, double* y, complex<double>* c, int
iflag, double eps, int64_t N1, int64_t N2, complex<double>* f, nufft_opts* opts)
int finufftf2d2many(int ntr, int64_t M, float* x, float* y, complex<float>* c, int iflag,
float eps, int64_t N1, int64_t N2, complex<float>* f, nufft_opts* opts)


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

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

      c[j] =   SUM   f[k1,k2] exp(+/-i (k1 x[j] + k2 y[j]))    for j = 0,...,M-1
              k1,k2
       where the sum is over integers -N1/2 <= k1 <= (N1-1)/2,
                                      -N2/2 <= k2 <= (N2-1)/2.
  Inputs:
    ntr     how many transforms (only for vectorized "many" functions, else ntr=1)
    M       number of nonuniform point targets
    x,y     nonuniform point coordinates in [-3pi,3pi]^2 (length M real arrays)
    iflag   if >=0, uses +i in complex exponential, otherwise -i
    eps     desired relative precision; smaller is slower. This can be chosen
            from 1e-1 down to ~ 1e-14 (in double precision) or 1e-6 (in single)
    N1      number of input Fourier modes (x direction)
    N2      number of input Fourier modes (y direction)
    f       Fourier mode coefficients (size N1*N2*ntr complex array)
    opts    pointer to options struct (see opts.rst), or NULL for defaults

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

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

```
int finufft2d3(int64_t M, double* x, double* y, complex<double>* c, int iflag, double
eps, int64_t N, double* s, double* t, complex<double>* f, nufft_opts* opts)
int finufftf2d3(int64_t M, float* x, float* y, complex<float>* c, int iflag, float eps,
int64_t N, float* s, float* t, complex<float>* f, nufft_opts* opts)
```

```
int finufft2d3many(int ntr, int64_t M, double* x, double* y, complex<double>* c, int
iflag, double eps, int64_t N, double* s, double* t, complex<double>* f, nufft_opts* opts)
int finufftf2d3many(int ntr, int64_t M, float* x, float* y, complex<float>* c, int iflag,
float eps, int64_t N, float* s, float* t, complex<float>* f, nufft_opts* opts)

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

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

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

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

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

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

6.1.3 3D transforms

```
int finufft3d1(int64_t M, double* x, double* y, double* z, complex<double>* c, int iflag,
double eps, int64_t N1, int64_t N2, int64_t N3, complex<double>* f, nufft_opts* opts)
int finufftf3d1(int64_t M, float* x, float* y, float* z, complex<float>* c, int iflag,
float eps, int64_t N1, int64_t N2, int64_t N3, complex<float>* f, nufft_opts* opts)

int finufft3d1many(int ntr, int64_t M, double* x, double* y, double* z, complex<double>*
c, int iflag, double eps, int64_t N1, int64_t N2, int64_t N3, complex<double>* f,
nufft_opts* opts)
int finufftf3d1many(int ntr, int64_t M, float* x, float* y, float* z, complex<float>* c,
int iflag, float eps, int64_t N1, int64_t N2, int64_t N3, complex<float>* f, nufft_opts*
opts)

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

```
Computes to precision eps, via a fast algorithm, one or more transforms of the form:

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


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

 Inputs:
   ntr    how many transforms (only for vectorized "many" functions, else ntr=1)
   M      number of nonuniform point sources
   x,y,z  nonuniform point coordinates in [-3pi,3pi]^3 (length M real arrays)
   c      source strengths (size M*ntr complex array)
   iflag  if >=0, uses +i in complex exponential, otherwise -i
   eps    desired relative precision; smaller is slower. This can be chosen
          from 1e-1 down to ~ 1e-14 (in double precision) or 1e-6 (in single)
   N1     number of output Fourier modes to be computed (x direction)
   N2     number of output Fourier modes to be computed (y direction)
   N3     number of output Fourier modes to be computed (z direction)
   opts   pointer to options struct (see opts.rst), or NULL for defaults

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

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

```
int finufft3d2(int64_t M, double* x, double* y, double* z, complex<double>* c, int iflag,
double eps, int64_t N1, int64_t N2, int64_t N3, complex<double>* f, nufft_opts* opts)
int finufftf3d2(int64_t M, float* x, float* y, float* z, complex<float>* c, int iflag,
float eps, int64_t N1, int64_t N2, int64_t N3, complex<float>* f, nufft_opts* opts)


int finufft3d2many(int ntr, int64_t M, double* x, double* y, double* z, complex<double>*
c, int iflag, double eps, int64_t N1, int64_t N2, int64_t N3, complex<double>* f,
nufft_opts* opts)
int finufftf3d2many(int ntr, int64_t M, float* x, float* y, float* z, complex<float>* c,
int iflag, float eps, int64_t N1, int64_t N2, int64_t N3, complex<float>* f, nufft_opts*
opts)

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

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

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

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

```
                                -N2/2 <= k2 <= (N2-1)/2,
                                -N3/2 <= k3 <= (N3-1)/2.
  Inputs:
    ntr    how many transforms (only for vectorized "many" functions, else ntr=1)
    M      number of nonuniform point targets
    x,y,z  nonuniform point coordinates in [-3pi,3pi]^3 (length M real arrays)
    iflag  if >=0, uses +i in complex exponential, otherwise -i
    eps    desired relative precision; smaller is slower. This can be chosen
           from 1e-1 down to ~ 1e-14 (in double precision) or 1e-6 (in single)
    N1     number of input Fourier modes (x direction)
    N2     number of input Fourier modes (y direction)
    N3     number of input Fourier modes (z direction)
    f      Fourier mode coefficients (size N1*N2*N3*ntr complex array)
    opts   pointer to options struct (see opts.rst), or NULL for defaults

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

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

```
int finufft3d3(int64_t M, double* x, double* y, double* z, complex<double>* c, int iflag,
double eps, int64_t N, double* s, double* t, double* u, complex<double>* f, nufft_opts*
opts)
int finufftf3d3(int64_t M, float* x, float* y, float* z, complex<float>* c, int iflag,
float eps, int64_t N, float* s, float* t, float* u, complex<float>* f, nufft_opts* opts)


int finufft3d3many(int ntr, int64_t M, double* x, double* y, double* z, complex<double>*
c, int iflag, double eps, int64_t N, double* s, double* t, double* u, complex<double>* f,
nufft_opts* opts)
int finufftf3d3many(int ntr, int64_t M, float* x, float* y, float* z, complex<float>* c,
int iflag, float eps, int64_t N, float* s, float* t, float* u, complex<float>* f,
nufft_opts* opts)


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

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

            M-1
    f[k]  =  SUM  c[j] exp(+-i (s[k] x[j] + t[k] y[j] + u[k] z[j])),
            j=0
                        for k = 0,...,N-1.
  Inputs:
    ntr    how many transforms (only for vectorized "many" functions, else ntr=1)
    M      number of nonuniform point sources
    x,y,z  nonuniform point coordinates in R^3 (length M real arrays)
    c      source strengths (size M*ntr complex array)
    iflag  if >=0, uses +i in complex exponential, otherwise -i
```

```
   eps    desired relative precision; smaller is slower. This can be chosen
          from 1e-1 down to ~ 1e-14 (in double precision) or 1e-6 (in single)
   N      number of nonuniform frequency targets
   s,t,u  nonuniform frequency target coordinates in R^3 (length N real arrays)
   opts   pointer to options struct (see opts.rst), or NULL for defaults

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

6.2 Guru plan interface

This provides more flexibility than the simple or vectorized interfaces. Any transform requires (at least) calling the following four functions in order. However, within this sequence one may insert repeated `execute` calls, or another `setpts` followed by more `execute` calls, as long as the transform sizes (and number of transforms `ntr`) are consistent with those that have been set in the `plan` and in `setpts`. Keep in mind that `setpts` retains *pointers* to the user's list of nonuniform points, rather than copying these points; thus the user must not change their nonuniform point arrays until after any `execute` calls that use them.

```
int finufft_makeplan(int type, int dim, int64_t* nmodes, int iflag, int ntr, double eps,
finufft_plan* plan, nufft_opts* opts)
int finufftf_makeplan(int type, int dim, int64_t* nmodes, int iflag, int ntr, float eps,
finufftf_plan* plan, nufft_opts* opts)

  Make a plan to perform one or more general transforms.

  Under the hood, for type 1 and 2, this does FFTW planning and kernel Fourier
  transform precomputation. For type 3, this does very little, since the FFT
  sizes are not yet known.

Inputs:
    type   type of transform (1,2, or 3)
    dim    spatial dimension (1,2, or 3)
   nmodes if type is 1 or 2, numbers of Fourier modes (length dim array),
          ie, {N1} in 1D, {N1,N2} in 2D, or {N1,N2,N3} in 3D.
          If type is 3, it is unused.
   iflag  if >=0, uses +i in complex exponential, otherwise -i
   ntr    how many transforms (only for vectorized "many" functions, else ntr=1)
   eps    desired relative precision; smaller is slower. This can be chosen
          from 1e-1 down to ~ 1e-14 (in double precision) or 1e-6 (in single)
   opts   pointer to options struct (see opts.rst), or NULL for defaults

 Outputs:
    plan   plan object (under the hood this is a pointer to another struct)
   return value  0: success, 1: success but warning, >1: error (see error.rst)

 Notes:
    * All available threads are planned by default (but see opts.nthreads)
    * The vectorized (many vector) plan, ie ntrans>1, can be much faster
      than repeated calls with the same nonuniform points.
```

```
      Note that here the I/O data ordering is stacked rather than interleaved.
    * For more details about the fields in the opts pointer, see opts.rst
```

```
int finufft_setpts(finufft_plan plan, int64_t M, double* x, double* y, double* z, int64_t
N, double* s, double* t, double* z)
int finufftf_setpts(finufftf_plan plan, int64_t M, float* x, float* y, float* z, int64_t
N, float* s, float* t, float* z)

 Input nonuniform points with coordinates x (and possibly y, and possibly z),
 and, if type 3, nonuniform frequency target coordinates s (and possibly t,
 and possibly u), into an existing plan. If type is 1 or 2 then the last four
 arguments are ignored. Unused dimensions are ignored.

 Under the hood, for type 1 or 2, this routine bin-sorts the points (storing
 just the permutation rather than new copies of the coordinates). For type
 3 it also bin-sorts the frequencies, chooses two levels of grid sizes, then
 plans the inner type 2 call (interpolation and FFTW).

 Inputs:
     M       number of nonuniform spatial points (used by all types)
     x       nonuniform point x-coordinates (length M real array)
     y       if dim>1, nonuniform point y-coordinates (length M real array),
             ignored otherwise
     z       if dim>2, nonuniform point z-coordinates (length M real array),
             ignored otherwise
     N       number of nonuniform frequency targets (type 3 only, ignored
             otherwise)
     s       nonuniform frequency x-coordinates (length N real array)
     t       if dim>1, nonuniform frequency y-coordinates (length N real array),
             ignored otherwise
     u       if dim>2, nonuniform frequency z-coordinates (length N real array),
             ignored otherwise

 Input/Outputs:
     plan    plan object

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

 Notes:
     * For type 1 and 2, the values in x (and if nonempty, y and z) must
       lie in the interval [-3pi,3pi). For type 1 they are "sources", but for
       type 2, "targets". In contrast, for type 3 there are no restrictions on
       them, or on s, t, u, other than the resulting size of the internal fine
       grids fitting in memory.
     * The coordinates pointed to by any used arrays x, y, z, s, t, u must
       not be changed between this call and the below execute call!
```

```
int finufft_execute(finufft_plan plan, complex<double>* c, complex<double>* f)
int finufftf_execute(finufftf_plan plan, complex<float>* c, complex<float>* f)
```

```
Perform one or more NUFFT transforms using previously entered nonuniform
points and an existing plan. To summarize, this maps
   type 1: c -> f
   type 2: f -> c
   type 3: c -> f

Inputs:
     plan    plan object

Input/Outputs:
     c       For types 1 and 3, the input strengths at the nonuniform point
             sources (size M*ntr complex array).
             If type 2, the output values at the nonuniform point targets
             (size M*ntr complex array).
     f       If type 1, the output Fourier mode coefficients (size N1*ntr or
             N1*N2*ntr or N1*N2*N3*ntr complex array, when dim = 1, 2, or 3
             respectively).
             If type 2, the input Fourier mode coefficients (size N1*ntr or
             N1*N2*ntr or N1*N2*N3*ntr complex array, when dim = 1, 2, or 3
             respectively).
             If type 3, the output values at the nonuniform frequency targets
             (size N*ntr complex array).

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

Notes:
   * The contents of the arrays x, y, z, s, t, u must not have changed since
     the finufft_setpts call that read them. The execution rereads them
     (this way of doing business saves RAM).
   * f and c are contiguous Fortran-style arrays with the transform number,
     if ntr>1, being the "slowest" (outer) dimension.
```

```
int finufft_destroy(finufft_plan plan)
int finufftf_destroy(finufftf_plan plan)

  Deallocate a plan object. This must be used upon clean-up, or before reusing
  a plan in another call to finufft_makeplan.

  Inputs/Outputs:
       plan    plan object

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

OPTIONS PARAMETERS

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

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

This setting produces more timing output to stdout.

> **Warning:** In C/C++ and Fortran, don't forget to call the command which sets default options (`finufft_default_opts` or `finufftf_default_opts`) before you start changing them and passing them to FINUFFT.

7.1 Summary and quick advice

Here is a 1-line summary of each option, taken from the code (the header `include/nufft_opts.h`):

```
// FINUFFT options:
// data handling opts...
int modeord;         // (type 1,2 only): 0 CMCL-style increasing mode order
                     //                  1 FFT-style mode order
int chkbnds;         // 0 don't check NU pts in [-3pi,3pi), 1 do (<few % slower)

// diagnostic opts...
int debug;           // 0 silent, 1 some timing/debug, or 2 more
int spread_debug;    // spreader: 0 silent, 1 some timing/debug, or 2 tonnes
int showwarn;        // 0 don't print warnings to stderr, 1 do

// algorithm performance opts...
int nthreads;        // number of threads to use, or 0 uses all available
int fftw;            // plan flags to FFTW (FFTW_ESTIMATE=64, FFTW_MEASURE=0,...)
```

(continues on next page)

```
int spread_sort;        // spreader: 0 don't sort, 1 do, or 2 heuristic choice
int spread_kerevalmeth; // spreader: 0 exp(sqrt()), 1 Horner piecewise poly (faster)
int spread_kerpad;      // (exp(sqrt()) only): 0 don't pad kernel to 4n, 1 do
double upsampfac;       // upsampling ratio sigma: 2.0 std, 1.25 small FFT, 0.0 auto
int spread_thread;      // (vectorized ntr>1 only): 0 auto, 1 seq multithreaded,
                        //                          2 parallel single-thread spread
int maxbatchsize;       // (vectorized ntr>1 only): max transform batch, 0 auto
int spread_nthr_atomic; // if >=0, threads above which spreader OMP critical goes␣
↪atomic
int spread_max_sp_size; // if >0, overrides spreader (dir=1) max subproblem size
```

Here are their default settings (from `src/finufft.cpp:finufft_default_opts`):

```
o->modeord = 0;
o->chkbnds = 1;

o->debug = 0;
o->spread_debug = 0;
o->showwarn = 1;

o->nthreads = 0;
o->fftw = FFTW_ESTIMATE;
o->spread_sort = 2;
o->spread_kerevalmeth = 1;
o->spread_kerpad = 1;
o->upsampfac = 0.0;
o->spread_thread = 0;
o->maxbatchsize = 0;
o->spread_nthr_atomic = -1;
o->spread_max_sp_size = 0;
```

As for quick advice, the main options you'll want to play with are:

- `modeord` to flip ("fftshift") the Fourier mode ordering

- `debug` to look at timing output (to determine if your problem is spread/interpolation dominated, vs FFT dominated)

- `nthreads` to run with a different number of threads than the current maximum available through OpenMP (a large number can sometimes be detrimental, and very small problems can sometimes run faster on 1 thread)

- `fftw` to try slower plan modes which give faster transforms. The next natural one to try is `FFTW_MEASURE` (look at the FFTW3 docs)

See *Troubleshooting* for good advice on trying options, and read the full options descriptions below.

Warning: Some of the options are for experts only, and will result in slow or incorrect results. Please test options in a small known test case so that you understand their effect.

7.2 Documentation of all options

7.2.1 Data handling options

modeord: Fourier coefficient frequency index ordering in every dimension. For type 1, this is for the output; for type 2 the input. It has no effect in type 3. Here we use `N` to denote the size in any of the relevant dimensions:

- if `modeord=0`: frequency indices are in increasing ordering, namely $\{-N/2, -N/2+1, \ldots, N/2-1\}$ if N is even, or $\{-(N-1)/2, \ldots, (N-1)/2\}$ if N is odd. For example, if `N=6` the indices are `-3,-2,-1,0,1,2`, whereas if `N=7` they are `-3,-2,-1,0,1,2,3`. This is called "CMCL ordering" since it is that of the CMCL NUFFT.

- if `modeord=1`: frequency indices are ordered as in the usual FFT, increasing from zero then jumping to negative indices half way along, namely $\{0, 1, \ldots, N/2-1, -N/2, -N/2+1, \ldots, -1\}$ if N is even, or $\{0, 1, \ldots, (N-1)/2, -(N-1)/2, \ldots, -1\}$ if N is odd. For example, if `N=6` the indices are `0,1,2,-3,-2,-1`, whereas if `N=7` they are `0,1,2,3,-3,-2,-1`.

> **Note:** The index *sets* are the same in the two `modeord` choices; their ordering differs only by a cyclic shift. The FFT ordering cyclically shifts the CMCL indices floor$(N/2)$ to the left (often called an "fftshift").

chkbnds: whether to check the nonuniform points lie in the correct bounds.

- `chkbnds=0`: input nonuniform points in the arrays `x`, `y`, `z`, are fed straight into the spreader which assumes (for speed) that they lie in $[-3\pi, 3\pi)$. Points outside of this will then cause a segfault.

- `chkbnds=1`: the nonuniform points are checked to lie in this interval, and if any are found not to, the library exits with an error code and message to stderr. The trade-off is that simply doing this checking can lose several % in overall speed, especially in low-precision 3D transforms.

7.2.2 Diagnostic options

debug: Controls the amount of overall debug/timing output to stdout.

- `debug=0` : silent

- `debug=1` : print some information

- `debug=2` : prints more information

spread_debug: Controls the amount of debug/timing output from the spreader/interpolator.

- `spread_debug=0` : silent

  - `spread_debug=1` : prints some timing information

  - `spread_debug=1` : prints lots. This can print thousands of lines since it includes one line per *subproblem*.

showwarn: Whether to print warnings (these go to stderr).

- `showwarn=0` : suppresses such warnings

- `showwarn=1` : prints warnings

7.2.3 Algorithm performance options

nthreads: Number of threads to use. This sets the number of threads FINUFFT will use in FFTW, bin-sorting, and spreading/interpolation steps. This number of threads also controls the batch size for vectorized transforms (ie `ntr>1` *here*). Setting `nthreads=0` uses all threads available. For repeated small problems it can be advantageous to use a small number, such as 1.

fftw: FFTW planner flags. This number is simply passed to FFTW's planner; the flags are documented here. A good first choice is `FFTW_ESTIMATE`; however if you will be making multiple calls, consider `FFTW_MEASURE`, which could spend many seconds planning, but will give a faster run-time when called again from the same process. These macros are bit-wise flags defined in `/usr/include/fftw3.h` on a linux system; they currently have the values `FFTW_ESTIMATE=64` and `FFTW_MEASURE=0`. 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); there is a small overhead for lookup of such plans, which with many repeated small problems can motivate use of the *guru interface*.

spread_sort: Sorting mode within the spreader/interpolator.

- `spread_sort=0` : never sorts

- `spread_sort=1` : always sorts

- `spread_sort=2` : uses a heuristic to decide whether to sort or not.

The heuristic bakes in empirical findings such as: generally it is not worth sorting in 1D type 2 transforms, or when the number of nonuniform points is small. Do not change this from its default unless you observer.

spread_kerevalmeth: Kernel evaluation method in spreader/interpolator. This should not be changed from its default value, unless you are an expert wanting to compare against outdated

- `spread_kerevalmeth=0` : direct evaluation of `sqrt(exp(beta(1-x*x)))` in the ES kernel. This is outdated, and it's only possible use could be in exploring upsampling factors σ different from standard (see below).

- `spread_kerevalmeth=1` : use Horner's rule applied to piecewise polynomials with precomputed coefficients. This is faster, less brittle to compiler/glibc/CPU variations, and is the recommended approach. It only works for the standard upsampling factors listed below.

spread_kerpad: whether to pad the number of direct kernel evaluations per dimension and per nonuniform point to a multiple of four; this can help SIMD vectorization. It only applies to the (outdated) `spread_kerevalmeth=0` choice. There is thus little reason for the nonexpert to mess with this option.

- `spread_kerpad=0` : do not pad

- `spread_kerpad=0` : pad to next multiple of four

upsampfac: This is the internal real factor by which the FFT (fine grid) is chosen larger than the number of requested modes in each dimension, for type 1 and 2 transforms. We have built efficient kernels for only two settings, as follows. Otherwise, setting it to zero chooses a good heuristic:

- `upsampfac=0.0` : use heuristics to choose `upsampfac` as one of the below values, and use this value internally. The value chosen is visible in the text output via setting `debug>=2`. This setting is recommended for basic users; however, if you seek more performance it is quick to try the other of the below.

- `upsampfac=2.0` : standard setting of upsampling. This is necessary if you need to exceed 9 digits of accuracy.

- `upsampfac=1.25` : low-upsampling option, with lower RAM, smaller FFTs, but wider spreading kernel. The latter can be much faster than the standard 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 (2 to 3 times) 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). Because the kernel width is limited to 16, currently, thus only 9-digit accuracy can currently be reached when using `upsampfac=1.25`.

---

spread_thread: in the case of multiple transforms per call (`ntr>1`, or the "many" interfaces), controls how multi-threading is used to spread/interpolate each batch of data.

- `spread_thread=0` : makes an automatic choice between the below. Recommended.

- `spread_thread=1` : acts on each vector in the batch in sequence, using multithreaded spread/interpolate on that vector. It can be slightly better than `2` for large problems.

- `spread_thread=2` : acts on all vectors in a batch (of size chosen typically to be the number of threads) simultaneously, assigning each a thread which performs a single-threaded spread/interpolate. It is much better than `1` for all but large problems. (Historical note: this was used by Melody Shih for the original "2dmany" interface in 2018.)

Note: Historical note: A former option `3` has been removed. This was like `2` except allowing nested OMP parallelism, so multi-threaded spread-interpolate was used for each of the vectors in a batch in parallel. This was used by Andrea Malleo in 2019. We have not yet found a case where this beats both `1` and `2`, hence removed it due to complications with changing the OMP nesting state in both old and new OMP versions.

---

maxbatchsize: in the case of multiple transforms per call (`ntr>1`, or the "many" interfaces), set the largest batch size of data vectors. Here `0` makes an automatic choice. If you are unhappy with this, then for small problems it should equal the number of threads, while for large problems it appears that `1` often better (since otherwise too much simultaneous RAM movement occurs). Some further work is needed to optimize this parameter.

spread_nthr_atomic: if non-negative: for numbers of threads up to this value, an OMP critical block for `add_wrapped_subgrid` is used in spreading (type 1 transforms). Above this value, instead OMP atomic writes are used, which scale better for large thread numbers. If negative, the heuristic default in the spreader is used, set in `src/spreadinterp.cpp:setup_spreader()`.

spread_max_sp_size: if positive, overrides the maximum subproblem (chunking) size for multithreaded spreading (type 1 transforms). Otherwise the default in the spreader is used, set in `src/spreadinterp.cpp:setup_spreader()`, which we believe is a decent heuristic for Intel i7 and xeon machines.

ERROR (STATUS) CODES

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

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

When `ier=1` (warning only) the transform(s) is/are still completed, at the smallest epsilon achievable, so, with that caveat, the answer should still be usable.

For any other nonzero values of `ier` the transform may not have been performed and the output should not be trusted. However, we hope that the value of `ier` will help to narrow down the problem.

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

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

8.1 Large internal arrays

In case your input parameters demand the allocation of very large arrays, an internal check is done to see if their size exceeds a rather generous internal limit, set in `defs.h` as `MAX_NF`. The current value in the source code is `1e11`, which corresponds to about 1TB for double precision. Allocations beyond this cause a graceful exit with error code 2 as above. Such a large allocation can be due to enormous N (in types 1 or 2), or M, but also large values of the space-bandwidth product (loosely, range of x_j points times range of k_j points) for type 3 transforms; see Remark 5 in *reference FIN*. Note that mallocs smaller than this, but which still exceed available RAM, may cause segfaults as usual. For simplicity of code, we do not do error checking on every malloc or STL vector creation in the code, and neither is this recommended in modern style guides. If you have a large-RAM machine and want to exceed the above hard-coded limit, you will need to edit `defs.h` and recompile.

NINE

TROUBLESHOOTING

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

9.1 Mathematical "issues" and advice

- When requested tolerance is around 10^{-14} or less in double-precision, or 10^{-6} or less in single-precision, it will most likely be impossible for FINUFFT (or any other NUFFT library) to achieve this, due to inevitable round-off error. Here, "error" is to be understood relative to the norm of the returned vector of values. This is especially true when there is a large number of modes in any single dimension (N_1, N_2 or N_3), since this empirically scales the round-off error (fortunately, round-off does not appear to scale with the total N or M). Such round-off error is analysed and measured in Section 4.2 of our SISC paper.

- If you request a tolerance that FINUFFT knows it cannot achieve, it will return `ier=1` after performing transforms as accurately as it can. However, the status `ier=0` does not imply that the requested accuracy *was* achieved, merely that parameters were chosen to give this estimated accuracy, if possible. As our SISC paper shows, for typical situations, relative ℓ_2 errors match the requested tolerances over a wide range. Users should always check *convergence* (by, for instance, varying `tol` and measuring any changes in their results); this is generally true in scientific computing.

- On the above topic, strangely, in single-precision, requesting tolerance of 10^{-7} or 10^{-6} can give slightly *worse* accuracy than 10^{-5} or 10^{-4}. We are looking into this. It is usually best to request at least 2–3 digits above the respective machine precision, for either single or double precision.

- The type 1 and type 2 transforms are adjoints but **not inverses of each other** (unlike in the plain FFT case, where, up to a constant factor N, the adjoint is the inverse). Therefore, if you are not getting the expected answers, please check that you have not made this assumption. In the *tutorials* we will add examples showing how to invert the NUFFT; also see NFFT3 inverse transforms.

9.2 Speed issues and advice

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

- Try printing debug output to see step-by-step progress by FINUFFT. Do this by setting `opts.debug` to 1 or 2 then looking at the timing information.

- Try reducing the number of threads, either those available via OpenMP, or via `opts.nthreads`, perhaps down to 1 thread, to make sure you are not having collisions between threads, or slowdown due to thread overheads. Hyperthreading (more threads than physical cores) rarely helps much. Thread collisions are possible if large problems are run with a large number of (say more than 64) threads. Another case causing slowness is very

many repetitions of small problems; see `test/manysmallprobs` which exceeds 10^7 points/sec with one thread via the guru interface, but can get ridiculously slower with many threads; see https://github.com/flatironinstitute/finufft/issues/86

- Try setting a crude tolerance, eg `tol=1e-3`. How many digits do you actually need? This has a big effect in higher dimensions, since the number of flops scales like $(\log 1/\epsilon)^d$, but not quite as big an effect as this scaling would suggest, because in higher dimensions the flops/RAM ratio is higher.

- If type 3, make sure your choice of points does not have a massive *space-bandwidth product* (ie, product of the volumes of the smallest d-dimension axes-aligned cuboids enclosing the nonuniform source and the target points); see Remark 5 of our SISC paper. In short, if the spreads of $\mathbf{x}_j$ and of $\mathbf{s}_k$ are both big, you may be in trouble. This can lead to enormous fine grids and hence slow FFTs. Set `opts.debug=1` to examine the `nf1`, etc, fine grid sizes being chosen, and the array allocation sizes. If they are huge, consider direct summation, as discussed *here*.

- The timing of the first FFTW call is complicated, depending on the FFTW flags (plan mode) used. This is really an FFTW planner flag usage question. Such issues are known, and modes benchmarked in other documentation, eg for 2D in poppy. In short, using more expensive FFTW planning modes like `FFTW_MEASURE` can give better performance for repeated FFTW calls, but be **much** more expensive in the first (planning) call. This is why we choose `FFTW_ESTIMATE` as our default `opts.fftw` option.

- Check that you are not using too much RAM, hence swapping to hard disk or SSD. The multithreaded type-1 spreader can use up to another fine grid's worth of storage in the form of subgrids. If RAM is too large, try overriding `opts.spread_max_sp_size` to a nonzero value smaller than the default value set in `src/spreadinterp.cpp:setup_spreader()`, to reduce RAM. However, note that this may slow it down, because we have built in a decent heuristic for the default.

- Make sure you did not override `opts.spread_sort`, which if set to zero does no sorting, which can give very slow RAM access if the nonuniform points are ordered poorly (eg randomly) in larger 2D or 3D problems.

- Are you calling the simple interface a huge number of times for small problems, but these tasks have something in common (number of modes, or locations of nonuniform points)? If so, try the "many vector" or guru interface, which removes overheads in repeated FFTW plan look-up, and in bin-sorting. They can be 10-100x faster.

9.3 Crash (segfault) issues and advice

- The most common problem is passing in pointers to the wrong size of object, eg, single vs double precision, or int32 vs int64. The library includes both precisions, so make sure you are calling the correct one (commands begin `finufft` for double, `finufftf` for single).

- If you use C++/C/Fortran and tried to change options, did you forget to call `finufft_default_opts` first?

- Maybe you have switched off nonuniform point bounds checking (`opts.chkbnds=0`) for a little extra speed? Try switching it on again to catch illegal coordinates.

- Thread-safety: are you calling FINUFFT from inside a multithreaded block of code, without compiling with `-DFFTW_PLAN_SAFE` or setting `opts.nthreads=1`? If `gdb` indicates crashes during FFTW calls, this is another sign.

- To isolate where a crash is occurring, set `opts.debug` to 1 or 2, and check the text output of the various stages. With a debug setting of 2 or above, when `ntrans>1` a large amount of text can be generated.

- To diagnose problems with the spread/interpolation stage, similarly setting `opts.spread_debug` to 1 or 2 will print even more output. Here the setting 2 generates a large amount of output even for a single transform.

9.4 Other known issues with library or interfaces

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

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

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

9.5 Bug reports

If you think you have found a new bug, and have read the above, please file a new issue on the github project page, https://github.com/flatironinstitute/finufft/issues. Include a minimal code which reproduces the bug, along with details about your machine, operating system, compiler, version of FINUFFT, and output with `opts.debug=2`. If you have a known bug and have ideas, please add to the comments for that issue.

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

TUTORIALS AND APPLICATION DEMOS

The following are instructive demos of using FINUFFT for a variety of spectrally-related tasks arising in scientific computing and signal/image processing. We will slowly grow the list (contact us to add one). For conciseness of code, and ease of writing, they are currently in MATLAB (they should work on versions at least back to R2017a).

10.1 Fast evaluation of Fourier series at arbitrary points

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

10.1.1 1D Fourier series

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

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

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

```
M = 1e6; x = L*rand(1,M);          % make random target points in [0,L)
tol = 1e-12;
x_scaled = x * (2*pi/L);           % don't forget to scale to 2pi-periodic!
tic; c = finufft1d2(x_scaled,+1,tol,fk); toc   % evaluate Fourier series at x
```
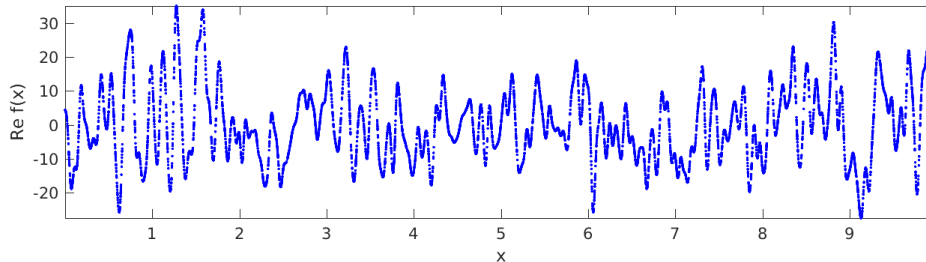
```
Elapsed time is 0.026038 seconds.
```

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

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

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

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

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


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

10.1.2 2D Fourier series

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

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

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

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

```
Elapsed time is 0.092743 seconds.
```
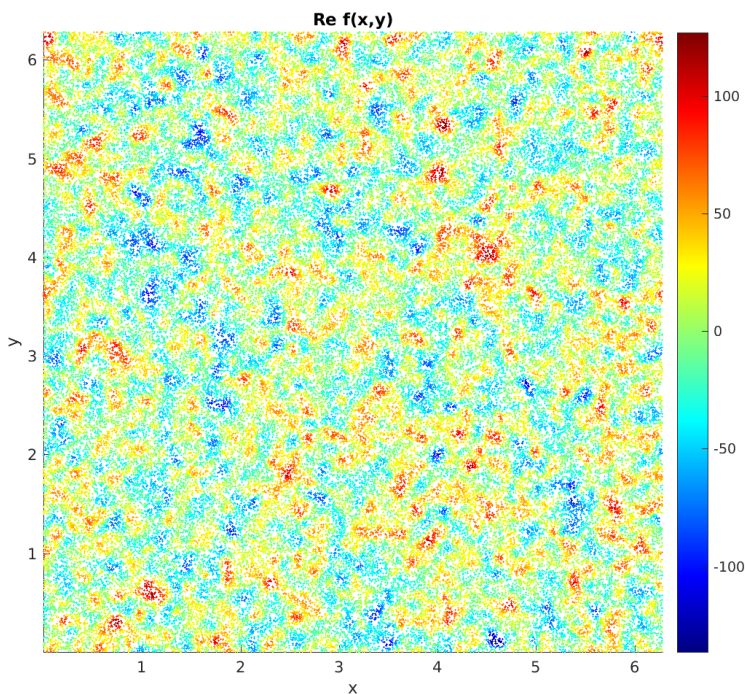
1 million modes to 1 million points in 92 milliseconds on a laptop is decent. We check the math (using a relative error measure) at just one (generic) point:

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

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

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

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


See the full code tutorial/serieseval2d.m.

10.2 Efficient evaluation of (continuous) Fourier transforms

Say you want to evaluate the continous (as opposed to discrete) Fourier *transform* (FT) of a given function, but you do not know the analytic formula for the FT. You need a numerical method. It is common to assume that the FFT is the right tool to do this, but this rarely so ...unless you are content with very poor accuracy! The reason is that the FFT applies only to equispaced data samples, which enforces the use of N equispaced nodes in any quadrature scheme for the Fourier integral. Thus, unless you apply endpoint weight corrections (which are available only in 1D, and stable only up to around 8th order; see references at the bottom of this page), you are generally stuck with 1st or 2nd order (the standard trapezoid rule) convergence with respect to N. And there are many situations where a FFT-based scheme would be even worse: this includes nonsmooth or singular functions (which demand custom quadrature rules even in

1D), smooth functions with varying scales (demanding *adaptive* quadrature for efficiency), and possibly nonsmooth functions on complicated domains in higher dimensions.

Here we show that the NUFFT is often the right tool for efficient and accurate Fourier tranform evaluation, since it allows the user to apply their favorite quadrature scheme as appropriate for whatever nasty function they desire. As long as N is bigger than around 10, the NUFFT becomes more efficient than direct evaluation of exponential sums; as we know, most quadrature rules, especially in 2D or 3D, involve many more points than this.

10.2.1 1D FTs evaluated at arbitrary frequencies

Given a function f, we'll need a single quadrature scheme with nodes x_j and weights w_j, $j = 1, \ldots, N$, that allows *accurate* approximation of its Fourier integral
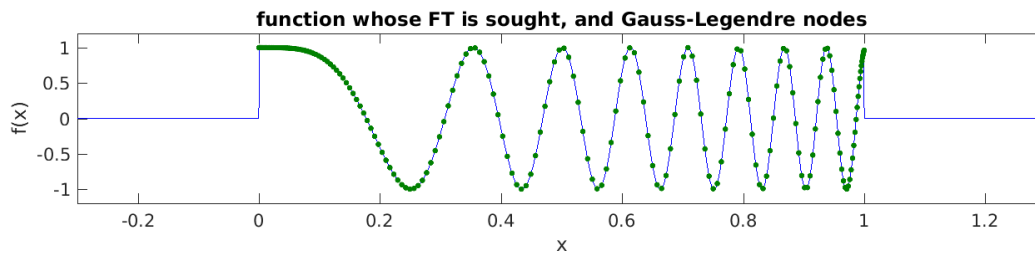
$$\hat{f}(k) = \int f(x)e^{ikx}dx \approx \sum_{j=1}^{N} f(x_j)e^{ikx_j}w_j \tag{10.1}$$

for all "target" frequencies k in some domain of interest. You can apply the below to any f for which you have such a rule.

For simplicity let's take f a smooth (somewhat oscillatory) function on (a, b), choose a million random frequency targets out to some `kmax`, then pick Gauss-Legendre quadrature for (a, b):
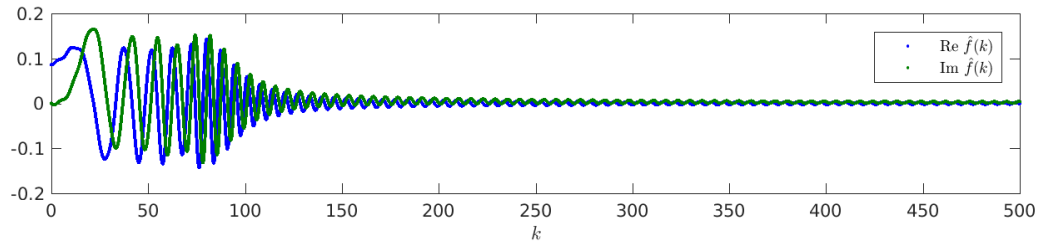
```
a=0; b=1;                       % interval
f = @(x) cos(50*x.^2);          % our smooth function defined on (a,b), zero elsewhere
M = 1e6;                        % # targets we want to compute the FT at
kmax = 500;
k = kmax * (2*rand(1,M)-1);     % desired target frequencies
N = 200;                        % how many quadrature nodes
[xj,wj] = lgwt(N,a,b);          % quadrature rule for smooth funcs on (a,b)
```

Below is f with the 200-node rule overlayed on it. You'll notice that the rule seems to be excessively fine (over-resolving $f(x)$), but that's because it actually needs to be able to resolve $f(x)e^{ikx}$ for all of our k values:

Notice (10.1) is simply a type 3 NUFFT with strengths $c_j = f(x_j)w_j$, so we evaluate it by calling FINUFFT (this takes 0.1 sec) then plot the resulting FT at its target k points:

```
tol = 1e-10;
fhat = finufft1d3(xj, f(xj).*wj, +1, tol, k);
plot(k, [real(fhat),imag(fhat)], '.');
```
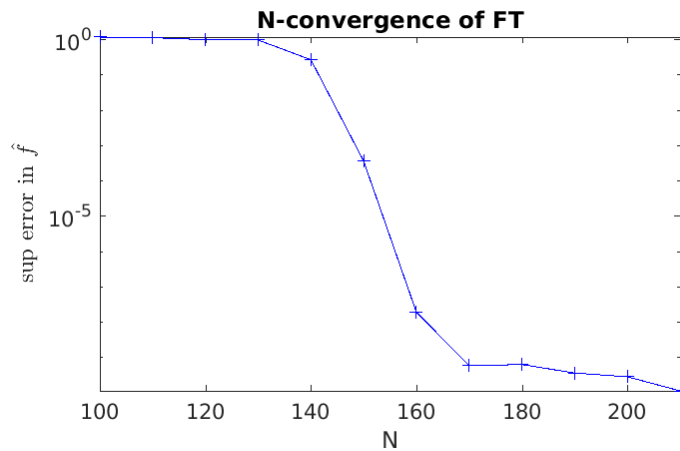
This looks like a continuous curve, but is actually (half a) million discrete points. Notice that because f was discontinuous on $\mathbb{R}$, $\hat{f}(k)$ decays slowly like $|k|^{-1}$. How do we know to trust the answer? A convergence study in N shows that 200 nodes was indeed enough to reduce the quadrature error to below the 10^{-10} NUFFT tolerance:

```
Ns = 100:10:220;                 % N values to check convergence
for i=1:numel(Ns), N=Ns(i);
  [xj,wj] = lgwt(N,a,b);         % N-node quadrature scheme for smooth funcs on (a,b)
  fhats{i} = finufft1d3(xj, f(xj).*wj, +1, tol, k);
end
f0 = norm(fhats{end},inf);       % compute rel sup norm of fhat vs highest-N case
for i=1:numel(Ns)-1, errsup(i) = norm(fhats{i}-fhats{end},inf)/f0; end
semilogy(Ns(1:end-1),errsup,'+-');
```


Remember: always do a convergence study! We see rapid spectral convergence as the quadrature rule resolves the oscillations in e^{ikx} at $|k| = k_{\max}$. See tutorial/contft1d.m for the full code.

Note: If you cared about only a few very high k values, numerical steepest descent applied at the endpoints a and b would eventually beat the above.

10.2.2 Faster FTs when frequencies lie on a grid

When the target frequencies lie on a uniform grid, the above type 3 NUFFT can be replaced by a type 1, which is faster, by a simple rescaling. Say that we replace the random targets in the above example by this uniform grid with spacing dk:

```
dk = 2*kmax/M;                   % spacing of target k grid
k = dk * (-M/2:(M/2-1));         % a particular uniform M-grid of this spacing
```

Reusing our quadrature `xj`, `wj` from above, we wish to stretch the frequency grid from spacing `dk` to have unit spacing, which is the integer (Fourier mode) grid implied by (1.1), the definition of the type 1. This is equivalent to squeezing the inputs x_j by the same factor, which we do as we send them in:

```
cj = f(xj).*wj;                % strengths (same as before)
fhat = finufft1d1(dk*xj, cj, +1, tol, M);   % type 1, requesting M modes
```

This took only 0.05 sec, around twice as fast as before. We must check it is giving what we want:

```
fhat3 = finufft1d3(xj, cj, +1, tol, k);     % old type 3 method
norm(fhat-fhat3,inf)
```

which reports around `3e-11`, so it worked. Note the specific offset of the `k` grid matched that of the Fourier mode indices; if you want a different offset, you will have to shift (by it to this specific offset, then post-multiply `fhat` with a corresponding phase.

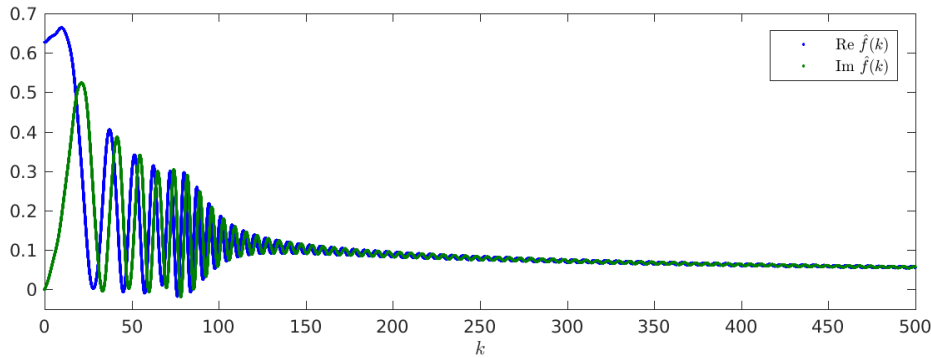### 10.2.3 1D FTs of singular functions

The above f was merely discontinuous. But you can now go further and easily replace (x_j, w_j) by a rule that is accurate for a function with known singularities. Eg, say $f(x) = x^{-1/2}g(x)$ where g is smooth on $[0, 1]$, then the change of variable $x = y^2$ means that $\int_0^1 f(x)dx = \int_0^1 2yf(y)dy$, the latter having a smooth integrand to which plain Gauss-Legendre can be applied, giving a new rule $x_j' = x_j^2$ and $w_j' = 2x_j w_j$. Notice how this bypassed the pain of building a $(\alpha = 0, \beta = -1/2)$ Gauss-Jacobi quadrature!

Let's try out this new rule on a suitably singular function, keeping other aspects the same as the above type 1 method:

```
f = @(x) cos(50*x.^2)./sqrt(x);    % singular function defined on (0,1), zero elsewhere
Ns = 180:20:240;                   % N values to check convergence
for i=1:numel(Ns), N=Ns(i);
  [xj,wj] = lgwt(N,a,b);           % N-node scheme for smooth funcs on (0,1)
  wj = 2*xj.*wj; xj = xj.*xj;      % convert to rule for -1/2 power singularity @ 0
  fhats{i} = finufft1d1(dk*xj, f(xj).*wj, +1, tol, M);     % type 1 as above
end
f0 = norm(fhats{end},inf);         % compute rel sup norm of fhat vs highest-N case
for i=1:numel(Ns)-1, errsup(i) = norm(fhats{i}-fhats{end},inf)/f0; end
disp([Ns(1:3); errsup(1:3)]')
fhat = fhats{end}; plot(k, [real(fhats),imag(fhat)], '.');
```

This exhibits rapid convergence kinking in at a slightly higher N, while $\hat{f}(k)$ now has even slower decay (which one can check is $|k|^{-1/2}$):

```
180         0.208975054515039
200      3.04233050928417e-05
220       1.9202016281569e-10
```

Neither f nor $\hat{f}$ is in $L^2(\mathbb{R})$. Other rules (adaptive, etc) can be designed to efficiently handle various other features of even nastier f choices.
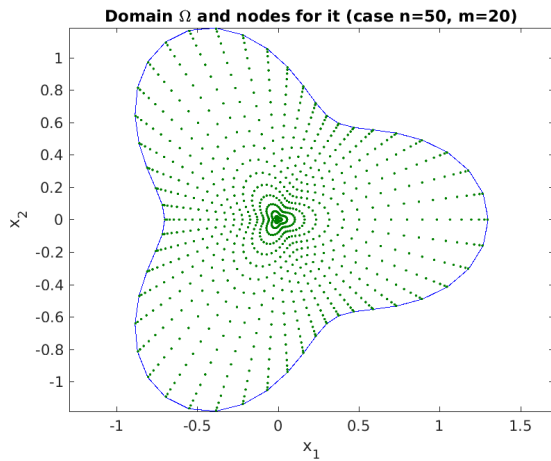
10.2.4 2D FTs

In higher dimensions, the idea is the same: set up a good quadrature rule for your function f, then apply it to the Fourier integral

$$\hat{f}(\mathbf{k}) = \int f(\mathbf{x})e^{i\mathbf{k}\cdot\mathbf{x}}d\mathbf{x} \approx \sum_{j=1}^{N} f(\mathbf{x}_j)e^{i\mathbf{k}\cdot\mathbf{x}_j}w_j \tag{10.2}$$
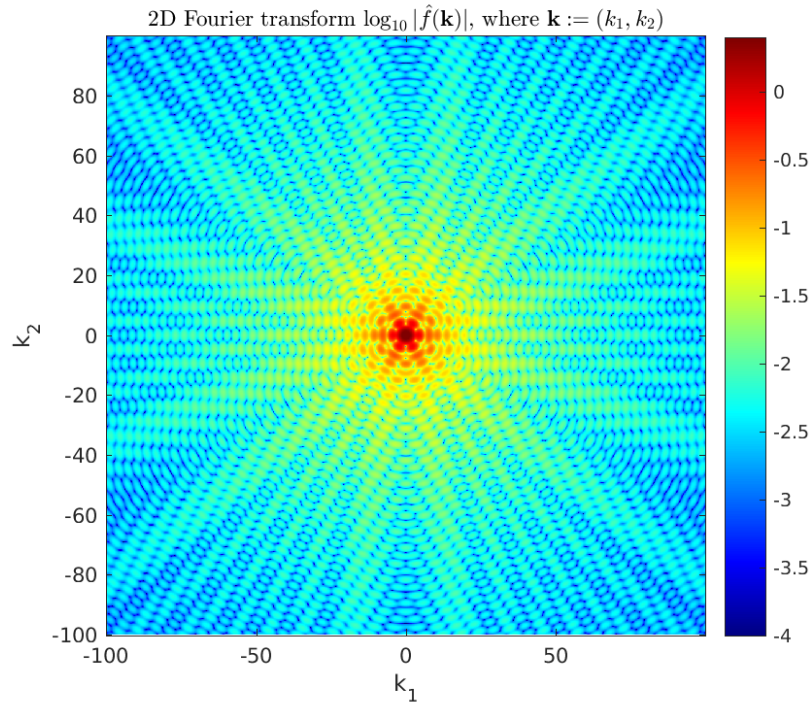
for all "target" frequencies $\mathbf{k}$ in your domain of interest. We demo the case of $f = \chi_\Omega$, the characteristic function of a bounde domain $\Omega \subset \mathbb{R}^2$, that is, $f(\mathbf{x}) = 1$ for $x \in \Omega$, and 0 otherwise. For simplicity, let's take Ω with boundary described in polar coordinates by $g(\theta) = 1 + 0.3\cos 5\theta$. This enables a simple two-level quadrature scheme, namely an outer n-node periodic trapezoid rule in θ, whose integrand is an inner m-node Gauss-Legendre rule applied to the radial integral. Since g is smooth, this will have spectral convergence in n and m. Here is a fresh code to make this quadrature over Ω:

```
g = @(t) 1 + 0.3*cos(3*t);                         % boundary shape
n = 280;                                           % # theta nodes
t = 2*pi*(1:n)/n; wt = (2*pi/n);                   % theta nodes, const weights
bx = cos(t).*g(t); by = sin(t).*g(t);             % boundary points
m = 70;                                            % # r nodes
[xr,wr] = lgwt(m,0,1);                             % rule for (0,1)
xj = nan(n*m,1); yj = xj; wj = xj;
for i=1:n                                          % loop over angles
  r = g(t(i)); jj = (1:m) + (i-1)*m;               % this radius; index list
  xj(jj) = cos(t(i))*r*xr; yj(jj) = sin(t(i))*r*xr;   % line of nodes
  wj(jj) = wt*r^2*xr.*wr;                % theta weight times rule for r.dr on (0,r)
end
plot([bx bx(1)],[by by(1)],'-'); hold on; plot(xj,yj,'.');
```

Note that we reduced the numbers of nodes in the plot for clarity. Say we want $\hat{f}$ on a 2D square grid of frequency targets. We apply the 2D version of the above type 1 scheme. The function is identically 1 in the domain, so the weights simply become the source strengths. We also image the result on a log scale:

```
kmax = 100;                                     % half the side length in k
M1 = 1e3;                                       % target grid will be M1-by-M1
dk = 2*kmax/M1;
k1 = dk * (-M1/2:(M1/2-1));                     % same 1D freq grid as before
tol = 1e-9;
fhat = finufft2d1(dk*xj, dk*yj, wj, +1, tol, M1, M1);  % M1^2 output nodes
imagesc(k1,k1,log10(abs(fhat))'); axis xy equal tight; colorbar
```

2D Fourier transform $\log_{10} |\hat{f}(\mathbf{k})|$, where $\mathbf{k} := (k_1, k_2)$

Thus we have computed the 2D FT of a discontinous function on a million-point grid to around 10-digit accuracy in 0.05 sec (the FINUFFT transform time). Note that, as with 1D discontinuous functions, the decay with $k := |\mathbf{k}|$ is slow (it is like $1/k$). See the full code tutorial/contft2d.m also for the study that shows that, for the above kmax, convergence to the tolerance has occurred by n=280 and m=70, needing $N = 19600$ nodes. A more efficient set would vary m with θ.

Note: An application of the above to optics is that Ω is a planar scatterer (or its complement, an aperture, via Babinet's principle) upon which a monochromatic plane wave is incident. The wavelength is small compared to the size of Ω, so that a scalar Kirchhoff diffraction model is a good one. If a downstream planar detector is very distant (the Fraunhofer diffraction limit), and the angles of scattering are small, then $|\hat{f}|^2$ is a good model for the detected scattered intensity.

10.2.5 Further reading

Higher-order end corrections to the trapezoid rule in 1D settings can allow all but $\mathcal{O}(1)$ of the nodes to be on a regular grid. They also can be useful for known singularities (log, 1/sqrt, etc):

- Kapur, S., Rokhlin, V. High-order corrected trapezoidal quadrature rules for singular functions. SIAM J. Numer. Anal. 34(4), 1331–1356 (1997)

- Alpert, B. K. Hybrid Gauss-Trapezoidal Quadrature Rules, SIAM J. Sci. Comput. 20(5), 1551–1584 (1999)

Kirchhoff approximation and Fraunhofer diffraction in optics:

- M. Born and E. Wolf, *Principles of Optics*, 6th edition. Section 8.3.

10.3 Periodic Poisson solve on non-Cartesian quadrature grid

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

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

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

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

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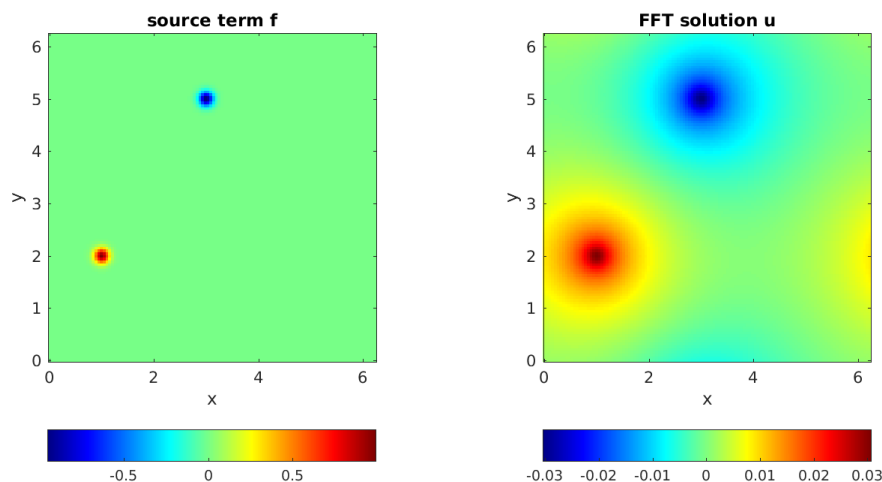
```
ns = 40:20:120;                % convergence study of grid points per side
for i=1:numel(ns), n = ns(i);
  x = 2*pi*(0:n-1)/n;          % grid
  [xx yy] = ndgrid(x,x);       % ordering: x fast, y slow
  f = src(xx,yy);              % eval source on grid
  fhat = ifft2(f);             % step 1: Fourier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

We observe spectral convergence to 14 digits:

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

Here we plot the FFT solution:

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

Now let's say you wish to do a similar Poisson solve on a **non-Cartesian grid** covering the same domain. There are two cases: a) the grid is unstructured and you do not know the weights of a quadrature scheme, or b) you do know the weights of a quadrature scheme (which ususally implies that the grid is structured, such as arising from a different coordinate system or an adaptive subdivision). By *quadrature scheme* we mean nodes $x_j \in \mathbb{R}^d$, $j = 1, \ldots, M$, and weights w_j such that, for all smooth functions f,
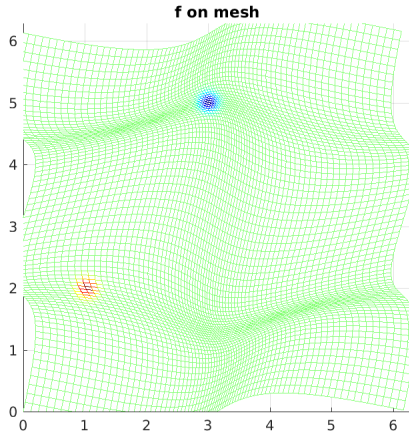
$$\int_{[0,2\pi)^d} f(x)dx \approx \sum_{j=1}^{M} f(x_j)w_j$$

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');
```
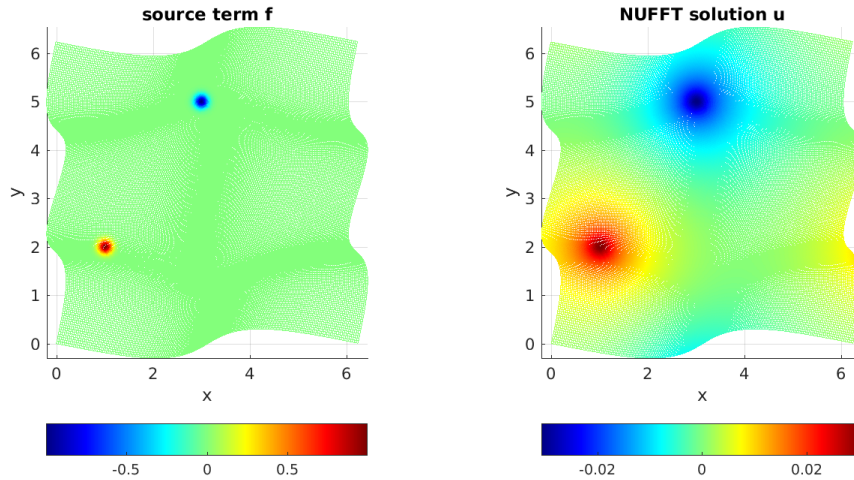
f on mesh

To solve on this grid, replace step 1 above by evaluating the Euler-Fourier formula using the quadrature scheme, which needs a type-1 NUFFT, and step 3 (evaluation on the nonuniform grid) by a type-2 NUFFT. Step 2 (the frequency filter) remains the same. Here is the demo code:
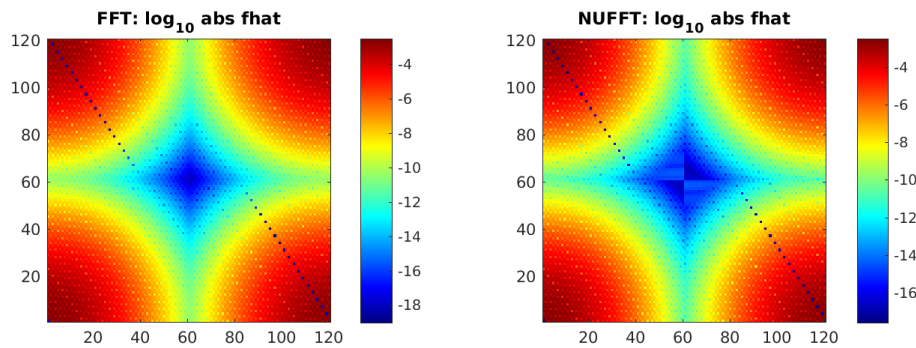
```
tol = 1e-12;              % NUFFT precision
ns = 80:40:240;           % convergence study of grid points per side
for i=1:numel(ns), n = ns(i);
  t = 2*pi*(0:n-1)/n;     % 1d unif grid
  [tt ss] = ndgrid(t,t);
  xxx = map(tt(:)',ss(:)');
  xx = reshape(xxx(1,:),[n n]); yy = reshape(xxx(2,:),[n n]);  % 2d NU pts
  J = mapJ(tt(:)',ss(:)');
  detJ = J(1,1:n^2).*J(2,n^2+1:end) - J(2,1:n^2).*J(1,n^2+1:end);
  ww = detJ / n^2;        % 2d quadr weights, including 1/(2pi)^2 in E-F integr
  f = src(xx,yy);
  Nk = 0.5*n; Nk = 2*ceil(Nk/2);  % modes to trust due to quadr err
  o.modeord = 1;          % use fft output mode ordering
  fhat = finufft2d1(xx(:),yy(:),f(:).*ww(:),1,tol,Nk,Nk,o);  % do E-F
  k = [0:Nk/2-1 -Nk/2:-1];   % Fourier mode grid
  [kx ky] = ndgrid(k,k);
  kfilter = 1./(kx.^2+ky.^2); % inverse -Laplacian in k-space (as above)
  kfilter(1,1) = 0; kfilter(Nk/2+1,:) = 0; kfilter(:,Nk/2+1) = 0;
  u = finufft2d2(xx(:),yy(:),-1,tol,kfilter.*fhat,o);  % eval filt F series @ NU
  u = reshape(real(u),[n n]);
  fprintf('n=%d:\tNk=%d\tu(0,0) = %.15e\n',n,Nk,u(1,1))  % check conv at same pt
end
```

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

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

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

The full code is at tutorial/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.

10.4 Efficient sampling from Gaussian Random Fields (GRFs)

A GRF is a random function defined by its power spectral density (PSD) $\hat{C}(k)$ as a function of wavevector k. It is thus stationary, ie, its statistical properties are translationally invariant. It is also known as a Gaussian process prior. Given a GRF model, it is important to be able to 1) draw independent samples from the GRF at a given set of target points, and 2) compute the log likelihood (under the model) of a given function values at a given set of target points. We focus on the first application. (For the second, see reference at the bottom of this page.)

For certain PSDs, independent samples may be generated by solving a PDE with iid random forcing; however this introduces boundary effects. Here we simply use Fourier methods to sample directly from any PSD that is a smooth function of k, and evaluate at arbitrary target points. The idea is that a *regular* Fourier grid k_j be used, with spacing sufficiently small based upon the size of the spatial domain in which target points lie. This serves as a quadrature scheme. The GRF model corresponds to random iid coefficients at these Fourier values, withzero mean and variance $\hat{C}(k_j)$. These coefficients are then evaluated at target points using the type 2 NUFFT.

The code is rudimentary but commented to be self-explanatory:

```matlab
% demo making indep samples from a 1D Gaussian random field, defined by its
% power spectral density \hat{C}(k), using Fourier methods without any BCs.
% Case of NU target pts, hence needs FINUFFT. (Note FFT is ok for unif targs.)
% Barnett 5/24/19
clear

% choose a GRF spectral density function \hat{C}(k)...
form = 'y';   % 'g' is for checking the variance; 'y' is what you care about.
if form=='g'        % Gaussian form (can get superalg convergence wrt K)
  K0 = 2e3;         % freq width, makes smooth below scale 1/K0
  Chat = @(k) (1/sqrt(2*pi)/K0)*exp(-(k/K0).^2/2);
elseif form=='y'    % desired Yukawa form (merely algebraic conv wrt K)
  alpha = 1.0; beta = 100; gamma = 1;    % sqrt(gamma/beta)=0.1 spatial scale
  Chat = @(k) (gamma*k.^2 + beta).^-alpha;
end

% user params...
L = 1;     % we want to evaluate on domain length D=[0,L].
M = 1e4;            % # targs
x = rand(1,M)*L;    % user can choose any target pts
tol = 1e-6;         % desired NUFFT precision
K = 1e4;            % max freq to go out to in Fourier transform for GRF

% alg param setup...
eta = 1.5;          % k-quad scheme convergence param, >=1; maybe =1 is enough?
dk_nyq = pi/L;      % Nyquist k-grid spacing for the width of target domain
dk = dk_nyq/eta;    % k-grid
P = 2*pi/dk         % actual period the NUFFT thinks there is
N = 2*K/dk;         % how many modes, if # k quadr pts.
N = 2*ceil(N/2)     % make even integer
k = (-N/2:N/2-1)*dk;  % the k grid
```
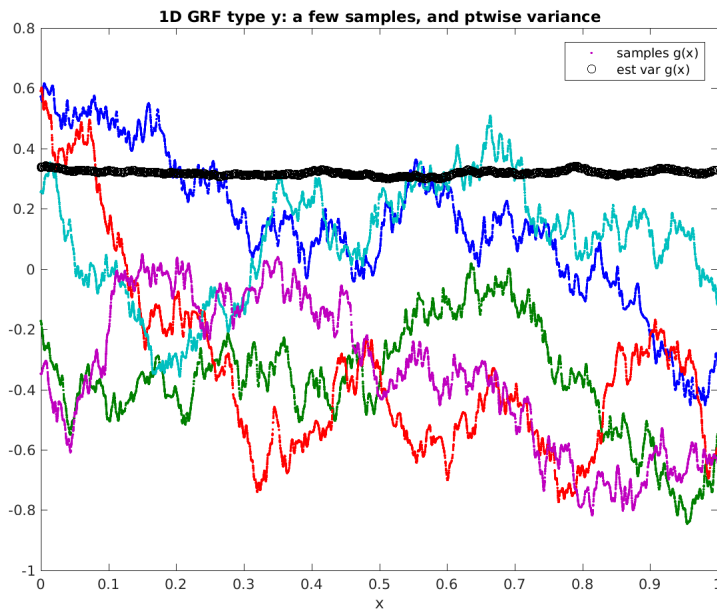
(continues on next page)

```matlab
% check the integral of spectral density (non-stochastic)... 1 for form='g'
I = dk*sum(Chat(k))    % plain trap rule
%figure; loglog(k,Chat(k),'+-'); xlabel('k'); title('check k-quadr scheme');

nr = 3e2;    % # reps, ie indep samples from GRF drawn.. 1e3 takes 3 sec
fprintf('sampling %d times... ',nr);
gvar = zeros(M,1);    % accumulate the ptwise variance
figure;
for r=1:nr
  fk = ((randn(1,N)+1i*randn(1,N))/sqrt(2)) .* sqrt(dk*Chat(k));    % spec ampl
  g = finufft1d2(x/P,+1,tol,fk);                    % do it, outputting g at targs
  if r<=5, h=plot(x,real(g),'.'); hold on; end  % show a few samples
  gvar = gvar + abs(g).^2;
end
gvar = gvar/nr;
fprintf('done. mean ptwise var g = %.3f\n',mean(gvar))

xlabel('x'); h2=plot(x,gvar, 'ko'); legend([h h2],'samples g(x)','est var g(x)')
v=axis; v(1:2)=[0 L]; axis(v);
title(sprintf('1D GRF type %s: a few samples, and ptwise variance',form));
```

The result is as follows, showing 5 samples from a Yukawa or Matern kernel, plus the pointwise variance estimated from 300 samples:

The calculation takes around 1 second.

If $\hat{C}(k)$ has singularities or discontinuities, a different k-space quadrature scheme would be needed. For an example of this, sampling from a 2D GRF, see random plane waves.

10.4.1 Further reading

For background on Gaussian random fields, aka, Gaussian processes, see:

- C E Rasmussen & C K I Williams, *Gaussian Processes for Machine Learning*, the MIT Press, 2006. http://www.GaussianProcess.org/gpml

- Fast Direct Methods for Gaussian Processes, Sivaram Ambikasaran, Daniel Foreman-Mackey, Leslie Greengard, David W. Hogg, and Michael O'Neil. arxiv:1403.6015 (2015)

For further applications, see *references*, and:

- Fast Fresnel diffraction for optics and acoustics applications.

- Tutorials for PyNUFFT with 1D and 2D reconstruction examples here.

- The numerical sampling of random plane waves.

- These seminar PDF slides.

USAGE FROM FORTRAN

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

11.1 Quick-start example

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

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

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

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

which writes the output to `fk`, and the status to the integer `ier`. Since the default is CMCL mode ordering, the output for frequency index `k` is found in `fk(k+N/2+1)`. `ier=0` indicates success, otherwise error codes are as in *here*. All available OMP threads are used, unless FINUFFT was built single-threaded. (Note that here the unallocated `null` is simply a way to pass a NULL pointer to our C++ wrapper; another would be `%val(0_8)`.) For a minimally complete test code demonstrating the above see `fortran/examples/simple1d1.f`.

Note: Higher-dimensional arrays are stored in Fortran ordering with x (N1) the fastest direction, and, in the vectorized ("many") calls, the transform number is slowest (transforms are stacked not interleaved). For instance, for the 2D type 1 vectorized transform `finufft2d1many(ntrans,M,xj,yj,cj,iflag,tol,N1,N2,fk,opts,ier)` with CMCL mode-ordering, the `(k1,k2)` frequency coefficient from transform number `t` is to be found at `fk(k1+N1/2+1 + (k2+N2/2)*N1 + t*N1*N2)`.

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

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

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

```
gfortran -I $(FINUFFT)/include simple1d1.f -o simple1d1 $(FINUFFT)/lib-static/libfinufft.
↪a -lfftw3 -lfftw3_omp -lgomp -lstdc++
```

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

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

11.2 Changing options

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

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

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

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

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

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

11.3 Summary of Fortran interface

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

```
      include 'finufft.fh'

      integer ier,iflag,ntrans,type,dim
```

<div align="right">(continues on next page)</div>

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

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

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

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

The single-precision (ie, `real*4` and `complex*8`) functions are identical except with the replacement of `finufft` with `finufftf` in each function name. All are defined (from the C++ side) in `fortran/finufftfort.cpp`.

11.4 Code examples

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

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

These are the double-precision file names; the single precision have a suffix `f` before the `.f`. The last four here are

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

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

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

MATLAB/OCTAVE INTERFACES

Note: See the *installation page* for how to build these interfaces, or look here.

12.1 Quick-start examples

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

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

The column vector output f should be interpreted as the Fourier coefficients with frequency indices `k = -N/2:N/2-1`. (This is because N is even; otherwise `k = -(N-1)/2:(N-1)/2`.) The values in f are accurate (relative to this vector's 2-norm) to roughly 12 digits, as requested by the tolerance argument `1e-12`. Choosing a larger (ie, worse) tolerance leads to faster transforms. The +1 controls the sign in the exponential; recall equation (1.1). All *options* maybe be changed from their defaults, for instance:

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

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

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

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

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

```matlab
type = 1; ntr = 1; o.modeord = 1;    % transform type, #transforms, opts
N = 2e5;                             % how many desired Fourier modes?
plan = finufft_plan(1,N,+1,ntr,1e-12,o);     % plan for N output modes
M = 1e5;                             % number of NU source points
x = 2*pi*rand(M,1);                  % array of NU source points
plan.setpts(x,[],[]);                % pass pointer to this array (M inferred)
% (note: the x array should now not be altered until all executes are done!)
c = randn(M,1)+1i*randn(M,1);        % iid random complex data (row or col vec)
f = plan.execute(c);                 % do the transform (0.008 sec, ie, faster)
% ...one could now change the points with setpts, and/or do new transforms
% with new c data...
delete(plan);                        % don't forget to clean up
```

> **Warning:** If an existing array is passed to `setpts`, then this array must not be altered before `execute` is called! This is because, in order to save RAM (allowing larger problems to be solved), internally FINUFFT stores only *pointers* to `x` (etc), rather than unnecessarily duplicating this data. This is not true if an *expression* such as `-x` or `2*pi*rand(M,1)` is passed to `setpts`, since in those cases the `plan` object does make internal copies, as per MATLAB's usual shallow-copy argument passing.

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

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

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

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

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

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

12.2 Full documentation

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

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

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

The individual commands have the following help documentation:

```
FINUFFT1D1   1D complex nonuniform FFT of type 1 (nonuniform to uniform).

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

This computes, to relative precision eps, via a fast algorithm:

              nj
   f(k1) =  SUM c[j] exp(+/-i k1 x(j))  for -ms/2 <= k1 <= (ms-1)/2
              j=1
 Inputs:
   x     locations of nonuniform sources on interval [-3pi,3pi), length nj
   c     length-nj complex vector of source strengths. If numel(c)>nj,
         expects a stack of vectors (eg, a nj*ntrans matrix) each of which is
         transformed with the same source locations.
   isign if >=0, uses + sign in exponential, otherwise - sign.
   eps   relative precision requested (generally between 1e-15 and 1e-1)
   ms    number of Fourier modes computed, may be even or odd;
         in either case, mode range is integers lying in [-ms/2, (ms-1)/2]
   opts   optional struct with optional fields controlling the following:
   opts.debug:   0 (silent, default), 1 (timing breakdown), 2 (debug info).
   opts.spread_debug: spreader: 0 (no text, default), 1 (some), or 2 (lots)
```

(continues on next page)

```
      opts.spread_sort:  0 (don't sort NU pts), 1 (do), 2 (auto, default)
      opts.spread_kerevalmeth:  0: exp(sqrt()), 1: Horner ppval (faster)
      opts.spread_kerpad: (iff kerevalmeth=0)  0: don't pad to mult of 4, 1: do
      opts.fftw: FFTW plan mode, 64=FFTW_ESTIMATE (default), 0=FFTW_MEASURE, etc
      opts.upsampfac:   sigma.  2.0 (default), or 1.25 (low RAM, smaller FFT)
      opts.spread_thread:   for ntrans>1 only. 0:auto, 1:seq multi, 2:par, etc
      opts.maxbatchsize:  for ntrans>1 only. max blocking size, or 0 for auto.
      opts.nthreads:    number of threads, or 0: use all available (default)
      opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
      opts.chkbnds: 0 (don't check NU points valid), 1 (do, default)
  Outputs:
      f     size-ms complex column vector of Fourier coefficients, or, if
            ntrans>1, a matrix of size (ms,ntrans).

Notes:
 * The vectorized (many vector) interface, ie ntrans>1, can be much faster
   than repeated calls with the same nonuniform points. Note that here the I/O
   data ordering is stacked rather than interleaved. See ../docs/matlab.rst
 * The class of input x (double vs single) controls whether the double or
   single precision library are called; precisions of all data should match.
 * For more details about the opts fields, see ../docs/opts.rst
 * See ERRHANDLER, VALID_* and FINUFFT_PLAN for possible warning/error IDs.
 * Full documentation is given in ../finufft-manual.pdf and online at
   http://finufft.readthedocs.io
```

```
FINUFFT1D2   1D complex nonuniform FFT of type 2 (uniform to nonuniform).

c = finufft1d2(x,isign,eps,f)
c = finufft1d2(x,isign,eps,f,opts)

This computes, to relative precision eps, via a fast algorithm:

   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 nonuniform targets on interval [-3pi,3pi], length nj
     f     complex Fourier coefficients. If a vector, length sets ms
           (with mode ordering given by opts.modeord). If a matrix, each
           of ntrans columns is transformed with the same nonuniform targets.
     isign if >=0, uses + sign in exponential, otherwise - sign.
     eps   relative precision requested (generally between 1e-15 and 1e-1)
     opts   optional struct with optional fields controlling the following:
     opts.debug:   0 (silent, default), 1 (timing breakdown), 2 (debug info).
     opts.spread_debug: spreader: 0 (no text, default), 1 (some), or 2 (lots)
     opts.spread_sort:  0 (don't sort NU pts), 1 (do), 2 (auto, default)
     opts.spread_kerevalmeth:  0: exp(sqrt()), 1: Horner ppval (faster)
     opts.spread_kerpad: (iff kerevalmeth=0)  0: don't pad to mult of 4, 1: do
     opts.fftw: FFTW plan mode, 64=FFTW_ESTIMATE (default), 0=FFTW_MEASURE, etc
     opts.upsampfac:   sigma.  2.0 (default), or 1.25 (low RAM, smaller FFT)
     opts.spread_thread:   for ntrans>1 only. 0:auto, 1:seq multi, 2:par, etc
```

```
    opts.maxbatchsize:  for ntrans>1 only. max blocking size, or 0 for auto.
    opts.nthreads:   number of threads, or 0: use all available (default)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default)
 Outputs:
    c      complex column vector of nj answers at targets, or,
           if ntrans>1, matrix of size (nj,ntrans).

Notes:
 * The vectorized (many vector) interface, ie ntrans>1, can be much faster
   than repeated calls with the same nonuniform points. Note that here the I/O
   data ordering is stacked rather than interleaved. See ../docs/matlab.rst
 * The class of input x (double vs single) controls whether the double or
   single precision library are called; precisions of all data should match.
 * For more details about the opts fields, see ../docs/opts.rst
 * See ERRHANDLER, VALID_* and FINUFFT_PLAN for possible warning/error IDs.
 * Full documentation is given in ../finufft-manual.pdf and online at
   http://finufft.readthedocs.io
```

```
FINUFFT1D3    1D complex nonuniform FFT of type 3 (nonuniform to nonuniform).

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

This computes, to relative precision eps, via a fast algorithm:

            nj
  f[k]  =  SUM   c[j] exp(+-i s[k] x[j]),       for k = 1, ..., nk
            j=1
 Inputs:
    x      locations of nonuniform sources on R (real line), length-nj vector.
    c      length-nj complex vector of source strengths. If numel(c)>nj,
           expects a stack of vectors (eg, a nj*ntrans matrix) each of which is
           transformed with the same source and target locations.
    isign if >=0, uses + sign in exponential, otherwise - sign.
    eps    relative precision requested (generally between 1e-15 and 1e-1)
    s      frequency locations of nonuniform targets on R, length-nk vector.
    opts   optional struct with optional fields controlling the following:
    opts.debug:   0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.spread_debug: spreader: 0 (no text, default), 1 (some), or 2 (lots)
    opts.spread_sort:  0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.spread_kerevalmeth:  0: exp(sqrt()), 1: Horner ppval (faster)
    opts.spread_kerpad: (iff kerevalmeth=0)  0: don't pad to mult of 4, 1: do
    opts.fftw: FFTW plan mode, 64=FFTW_ESTIMATE (default), 0=FFTW_MEASURE, etc
    opts.upsampfac:   sigma.  2.0 (default), or 1.25 (low RAM, smaller FFT)
    opts.spread_thread:    for ntrans>1 only. 0:auto, 1:seq multi, 2:par, etc
    opts.maxbatchsize:  for ntrans>1 only. max blocking size, or 0 for auto.
    opts.nthreads:   number of threads, or 0: use all available (default)
 Outputs:
    f      length-nk complex vector of values at targets, or, if ntrans>1,
           a matrix of size (nk,ntrans)
```

```
Notes:
 * The vectorized (many vector) interface, ie ntrans>1, can be much faster
   than repeated calls with the same nonuniform points. Note that here the I/O
   data ordering is stacked rather than interleaved. See ../docs/matlab.rst
 * The class of input x (double vs single) controls whether the double or
   single precision library are called; precisions of all data should match.
 * For more details about the opts fields, see ../docs/opts.rst
 * See ERRHANDLER, VALID_* and FINUFFT_PLAN for possible warning/error IDs.
 * Full documentation is given in ../finufft-manual.pdf and online at
   http://finufft.readthedocs.io
```

```
FINUFFT2D1   2D complex nonuniform FFT of type 1 (nonuniform to uniform).

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

This computes, to relative precision eps, via a fast algorithm:

                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    coordinates of nonuniform sources on the square [-3pi,3pi)^2,
           each a length-nj vector
    c      length-nj complex vector of source strengths. If numel(c)>nj,
           expects a stack of vectors (eg, a nj*ntrans matrix) each of which is
           transformed with the same source locations.
    isign if >=0, uses + sign in exponential, otherwise - sign.
    eps    relative precision requested (generally between 1e-15 and 1e-1)
    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   optional struct with optional fields controlling the following:
    opts.debug:   0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.spread_debug: spreader: 0 (no text, default), 1 (some), or 2 (lots)
    opts.spread_sort:  0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.spread_kerevalmeth:  0: exp(sqrt()), 1: Horner ppval (faster)
    opts.spread_kerpad: (iff kerevalmeth=0)  0: don't pad to mult of 4, 1: do
    opts.fftw: FFTW plan mode, 64=FFTW_ESTIMATE (default), 0=FFTW_MEASURE, etc
    opts.upsampfac:   sigma.  2.0 (default), or 1.25 (low RAM, smaller FFT)
    opts.spread_thread:   for ntrans>1 only. 0:auto, 1:seq multi, 2:par, etc
    opts.maxbatchsize:  for ntrans>1 only. max blocking size, or 0 for auto.
    opts.nthreads:   number of threads, or 0: use all available (default)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default)
  Outputs:
    f      size (ms,mt) complex matrix of Fourier coefficients
           (ordering given by opts.modeord in each dimension; ms fast, mt slow),
           or, if ntrans>1, a 3D array of size (ms,mt,ntrans).
```

```
Notes:
 * The vectorized (many vector) interface, ie ntrans>1, can be much faster
   than repeated calls with the same nonuniform points. Note that here the I/O
   data ordering is stacked rather than interleaved. See ../docs/matlab.rst
 * The class of input x (double vs single) controls whether the double or
   single precision library are called; precisions of all data should match.
 * For more details about the opts fields, see ../docs/opts.rst
 * See ERRHANDLER, VALID_* and FINUFFT_PLAN for possible warning/error IDs.
 * Full documentation is given in ../finufft-manual.pdf and online at
   http://finufft.readthedocs.io
```

```
FINUFFT2D2   2D complex nonuniform FFT of type 2 (uniform to nonuniform).

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

This computes, to relative precision eps, via a fast algorithm:

   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   coordinates of nonuniform targets on the square [-3pi,3pi)^2,
          each a vector of length nj
    f     complex Fourier coefficient matrix, whose size determines (ms,mt).
          (Mode ordering given by opts.modeord, in each dimension.)
          If a 3D array, 3rd dimension sets ntrans, and each of ntrans
          matrices is transformed with the same nonuniform targets.
    isign if >=0, uses + sign in exponential, otherwise - sign.
    eps   relative precision requested (generally between 1e-15 and 1e-1)
    opts   optional struct with optional fields controlling the following:
    opts.debug:   0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.spread_debug: spreader: 0 (no text, default), 1 (some), or 2 (lots)
    opts.spread_sort:  0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.spread_kerevalmeth:  0: exp(sqrt()), 1: Horner ppval (faster)
    opts.spread_kerpad: (iff kerevalmeth=0)  0: don't pad to mult of 4, 1: do
    opts.fftw: FFTW plan mode, 64=FFTW_ESTIMATE (default), 0=FFTW_MEASURE, etc
    opts.upsampfac:   sigma. 2.0 (default), or 1.25 (low RAM, smaller FFT)
    opts.spread_thread:   for ntrans>1 only. 0:auto, 1:seq multi, 2:par, etc
    opts.maxbatchsize:  for ntrans>1 only. max blocking size, or 0 for auto.
    opts.nthreads:   number of threads, or 0: use all available (default)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default)
 Outputs:
    c     complex column vector of nj answers at targets, or,
          if ntrans>1, matrix of size (nj,ntrans).

Notes:
 * The vectorized (many vector) interface, ie ntrans>1, can be much faster
   than repeated calls with the same nonuniform points. Note that here the I/O
   data ordering is stacked rather than interleaved. See ../docs/matlab.rst
```

```
 * The class of input x (double vs single) controls whether the double or
   single precision library are called; precisions of all data should match.
 * For more details about the opts fields, see ../docs/opts.rst
 * See ERRHANDLER, VALID_* and FINUFFT_PLAN for possible warning/error IDs.
 * Full documentation is given in ../finufft-manual.pdf and online at
   http://finufft.readthedocs.io
```

```
FINUFFT2D3   2D complex nonuniform FFT of type 3 (nonuniform to nonuniform).

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

This computes, to relative precision eps, via a fast algorithm:

              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    coordinates of nonuniform sources in R^2, each a length-nj vector.
    c      length-nj complex vector of source strengths. If numel(c)>nj,
           expects a stack of vectors (eg, a nj*ntrans matrix) each of which is
           transformed with the same source and target locations.
    isign if >=0, uses + sign in exponential, otherwise - sign.
    eps    relative precision requested (generally between 1e-15 and 1e-1)
    s,t    frequency coordinates of nonuniform targets in R^2,
           each a length-nk vector.
    opts   optional struct with optional fields controlling the following:
    opts.debug:   0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.spread_debug: spreader: 0 (no text, default), 1 (some), or 2 (lots)
    opts.spread_sort:  0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.spread_kerevalmeth:  0: exp(sqrt()), 1: Horner ppval (faster)
    opts.spread_kerpad: (iff kerevalmeth=0)  0: don't pad to mult of 4, 1: do
    opts.fftw: FFTW plan mode, 64=FFTW_ESTIMATE (default), 0=FFTW_MEASURE, etc
    opts.upsampfac:   sigma.  2.0 (default), or 1.25 (low RAM, smaller FFT)
    opts.spread_thread:   for ntrans>1 only. 0:auto, 1:seq multi, 2:par, etc
    opts.maxbatchsize:  for ntrans>1 only. max blocking size, or 0 for auto.
    opts.nthreads:   number of threads, or 0: use all available (default)
  Outputs:
    f      length-nk complex vector of values at targets, or, if ntrans>1,
           a matrix of size (nk,ntrans)

Notes:
 * The vectorized (many vector) interface, ie ntrans>1, can be much faster
   than repeated calls with the same nonuniform points. Note that here the I/O
   data ordering is stacked rather than interleaved. See ../docs/matlab.rst
 * The class of input x (double vs single) controls whether the double or
   single precision library are called; precisions of all data should match.
 * For more details about the opts fields, see ../docs/opts.rst
 * See ERRHANDLER, VALID_* and FINUFFT_PLAN for possible warning/error IDs.
 * Full documentation is given in ../finufft-manual.pdf and online at
   http://finufft.readthedocs.io
```

```
FINUFFT3D1    3D complex nonuniform FFT of type 1 (nonuniform to uniform).

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

This computes, to relative precision eps, via a fast algorithm:

                      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  coordinates of nonuniform sources on the cube [-3pi,3pi]^3,
           each a length-nj vector
    c      length-nj complex vector of source strengths. If numel(c)>nj,
           expects a stack of vectors (eg, a nj*ntrans matrix) each of which is
           transformed with the same source locations.
    isign if >=0, uses + sign in exponential, otherwise - sign.
    eps    relative precision requested (generally between 1e-15 and 1e-1)
    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   optional struct with optional fields controlling the following:
    opts.debug:   0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.spread_debug: spreader: 0 (no text, default), 1 (some), or 2 (lots)
    opts.spread_sort:  0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.spread_kerevalmeth:  0: exp(sqrt()), 1: Horner ppval (faster)
    opts.spread_kerpad: (iff kerevalmeth=0)  0: don't pad to mult of 4, 1: do
    opts.fftw: FFTW plan mode, 64=FFTW_ESTIMATE (default), 0=FFTW_MEASURE, etc
    opts.upsampfac:   sigma. 2.0 (default), or 1.25 (low RAM, smaller FFT)
    opts.spread_thread:   for ntrans>1 only. 0:auto, 1:seq multi, 2:par, etc
    opts.maxbatchsize:  for ntrans>1 only. max blocking size, or 0 for auto.
    opts.nthreads:   number of threads, or 0: use all available (default)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default)
  Outputs:
    f      size (ms,mt,mu) complex array of Fourier coefficients
           (ordering given by opts.modeord in each dimension; ms fastest, mu
           slowest), or, if ntrans>1, a 4D array of size (ms,mt,mu,ntrans).

Notes:
 * The vectorized (many vector) interface, ie ntrans>1, can be much faster
   than repeated calls with the same nonuniform points. Note that here the I/O
   data ordering is stacked rather than interleaved. See ../docs/matlab.rst
 * The class of input x (double vs single) controls whether the double or
   single precision library are called; precisions of all data should match.
 * For more details about the opts fields, see ../docs/opts.rst
 * See ERRHANDLER, VALID_* and FINUFFT_PLAN for possible warning/error IDs.
 * Full documentation is given in ../finufft-manual.pdf and online at
   http://finufft.readthedocs.io
```

```
FINUFFT3D2    3D complex nonuniform FFT of type 2 (uniform to nonuniform).

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

This computes, to relative precision eps, via a fast algorithm:

   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 coordinates of nonuniform targets on the cube [-3pi,3pi)^3,
          each a vector of length nj
    f     complex Fourier coefficient array, whose size sets (ms,mt,mu).
          (Mode ordering given by opts.modeord, in each dimension.)
          If a 4D array, 4th dimension sets ntrans, and each of ntrans
          3D arrays is transformed with the same nonuniform targets.
    isign if >=0, uses + sign in exponential, otherwise - sign.
    eps   relative precision requested (generally between 1e-15 and 1e-1)
    opts   optional struct with optional fields controlling the following:
    opts.debug:   0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.spread_debug: spreader: 0 (no text, default), 1 (some), or 2 (lots)
    opts.spread_sort:  0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.spread_kerevalmeth:  0: exp(sqrt()), 1: Horner ppval (faster)
    opts.spread_kerpad: (iff kerevalmeth=0)  0: don't pad to mult of 4, 1: do
    opts.fftw: FFTW plan mode, 64=FFTW_ESTIMATE (default), 0=FFTW_MEASURE, etc
    opts.upsampfac:   sigma.  2.0 (default), or 1.25 (low RAM, smaller FFT)
    opts.spread_thread:    for ntrans>1 only. 0:auto, 1:seq multi, 2:par, etc
    opts.maxbatchsize:  for ntrans>1 only. max blocking size, or 0 for auto.
    opts.nthreads:   number of threads, or 0: use all available (default)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default)
 Outputs:
    c      complex column vector of nj answers at targets, or,
           if ntrans>1, matrix of size (nj,ntrans).

Notes:
 * The vectorized (many vector) interface, ie ntrans>1, can be much faster
   than repeated calls with the same nonuniform points. Note that here the I/O
   data ordering is stacked rather than interleaved. See ../docs/matlab.rst
 * The class of input x (double vs single) controls whether the double or
   single precision library are called; precisions of all data should match.
 * For more details about the opts fields, see ../docs/opts.rst
 * See ERRHANDLER, VALID_* and FINUFFT_PLAN for possible warning/error IDs.
 * Full documentation is given in ../finufft-manual.pdf and online at
   http://finufft.readthedocs.io
```

```
FINUFFT3D3    3D complex nonuniform FFT of type 3 (nonuniform to nonuniform).
```

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

This computes, to relative precision eps, via a fast algorithm:

            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  coordinates of nonuniform sources in R^3, each a length-nj vector.
   c      length-nj complex vector of source strengths. If numel(c)>nj,
          expects a stack of vectors (eg, a nj*ntrans matrix) each of which is
          transformed with the same source and target locations.
   isign if >=0, uses + sign in exponential, otherwise - sign.
   eps    relative precision requested (generally between 1e-15 and 1e-1)
   s,t,u  frequency coordinates of nonuniform targets in R^3,
          each a length-nk vector.
   opts   optional struct with optional fields controlling the following:
   opts.debug:   0 (silent, default), 1 (timing breakdown), 2 (debug info).
   opts.spread_debug: spreader: 0 (no text, default), 1 (some), or 2 (lots)
   opts.spread_sort:  0 (don't sort NU pts), 1 (do), 2 (auto, default)
   opts.spread_kerevalmeth:  0: exp(sqrt()), 1: Horner ppval (faster)
   opts.spread_kerpad: (iff kerevalmeth=0)  0: don't pad to mult of 4, 1: do
   opts.fftw: FFTW plan mode, 64=FFTW_ESTIMATE (default), 0=FFTW_MEASURE, etc
   opts.upsampfac:   sigma. 2.0 (default), or 1.25 (low RAM, smaller FFT)
   opts.spread_thread:   for ntrans>1 only. 0:auto, 1:seq multi, 2:par, etc
   opts.maxbatchsize:  for ntrans>1 only. max blocking size, or 0 for auto.
   opts.nthreads:   number of threads, or 0: use all available (default)
 Outputs:
   f      length-nk complex vector of values at targets, or, if ntrans>1,
          a matrix of size (nk,ntrans)


Notes:
 * The vectorized (many vector) interface, ie ntrans>1, can be much faster
   than repeated calls with the same nonuniform points. Note that here the I/O
   data ordering is stacked rather than interleaved. See ../docs/matlab.rst
 * The class of input x (double vs single) controls whether the double or
   single precision library are called; precisions of all data should match.
 * For more details about the opts fields, see ../docs/opts.rst
 * See ERRHANDLER, VALID_* and FINUFFT_PLAN for possible warning/error IDs.
 * Full documentation is given in ../finufft-manual.pdf and online at
   http://finufft.readthedocs.io
```

```
FINUFFT_PLAN   is a class which wraps the guru interface to FINUFFT.

 Full documentation is given in ../finufft-manual.pdf and online at
 http://finufft.readthedocs.io
 Also see examples in the matlab/examples and matlab/test directories.

PROPERTIES
  mwptr - opaque pointer to a C++ finufft_plan object (see MWrap manual),
```

```
          whose properties cannot be accessed directly
  floatprec - either 'double' or 'single', tracks what precision of C++
          library is being called
  type, dim, n_modes, n_trans, nj, nk - other plan parameters
 Note: the user should never alter these plan properties directly! Rather,
 the below methods should be used to create, use, and destroy plans.


METHODS
  finufft_plan - create guru plan object for one/many general nonuniform FFTs.
  setpts       - process nonuniform points for general FINUFFT transform(s).
  execute      - execute single or many-vector FINUFFT transforms in a plan.


General notes:
 * use delete(plan) to remove a plan after use.
 * See ERRHANDLER, VALID_*, and this code for warning/error IDs.




========== Detailed description of guru methods ==========================

1) FINUFFT_PLAN create guru plan object for one/many general nonuniform FFTs.

plan = finufft_plan(type, n_modes_or_dim, isign, ntrans, eps)
plan = finufft_plan(type, n_modes_or_dim, isign, ntrans, eps, opts)


Creates a finufft_plan MATLAB object in the guru interface to FINUFFT, of
 type 1, 2 or 3, and with given numbers of Fourier modes (unless type 3).

Inputs:
    type           transform type: 1, 2, or 3
    n_modes_or_dim  if type is 1 or 2, the number of Fourier modes in each
                    dimension: [ms] in 1D, [ms mt] in 2D, or [ms mt mu] in 3D.
                    Its length sets the dimension, which must be 1, 2 or 3.
                    If type is 3, in contrast, its *value* fixes the dimension
    isign if >=0, uses + sign in exponential, otherwise - sign.
    eps   relative precision requested (generally between 1e-15 and 1e-1)
    opts   optional struct with optional fields controlling the following:
    opts.debug:   0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.spread_debug: spreader: 0 (no text, default), 1 (some), or 2 (lots)
    opts.spread_sort:  0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.spread_kerevalmeth:  0: exp(sqrt()), 1: Horner ppval (faster)
    opts.spread_kerpad: (iff kerevalmeth=0)  0: don't pad to mult of 4, 1: do
    opts.fftw: FFTW plan mode, 64=FFTW_ESTIMATE (default), 0=FFTW_MEASURE, etc
    opts.upsampfac:   sigma. 2.0 (default), or 1.25 (low RAM, smaller FFT)
    opts.spread_thread:   for ntrans>1 only. 0:auto, 1:seq multi, 2:par, etc
    opts.maxbatchsize:  for ntrans>1 only. max blocking size, or 0 for auto.
    opts.nthreads:   number of threads, or 0: use all available (default)
    opts.floatprec: library precision to use, 'double' (default) or 'single'.
    for type 1 and 2 only, the following opts fields are also relevant:
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default)
Outputs:
```

```
    plan            finufft_plan object (opaque pointer)

Notes:
 * For type 1 and 2, this does the FFTW planning and kernel-FT precomputation.
 * For type 3, this does very little, since the FFT sizes are not yet known.
 * Be default all threads are planned; control how many with opts.nthreads.
 * The vectorized (many vector) plan, ie ntrans>1, can be much faster
   than repeated calls with the same nonuniform points. Note that here the I/O
   data ordering is stacked rather than interleaved. See ../docs/matlab.rst
 * For more details about the opts fields, see ../docs/opts.rst


2) SETPTS   process nonuniform points for general FINUFFT transform(s).

plan.setpts(xj)
plan.setpts(xj, yj)
plan.setpts(xj, yj, zj)
plan.setpts(xj, [], [], s)
plan.setpts(xj, yj, [], s, t)
plan.setpts(xj, yj, zj, s, t, u)

 When plan is a finufft_plan MATLAB object, brings in nonuniform point
 coordinates (xj,yj,zj), and additionally in the type 3 case, nonuniform
 frequency target points (s,t,u). Empty arrays may be passed in the case of
 unused dimensions. For all types, sorting is done to internally store a
 reindexing of points, and for type 3 the spreading and FFTs are planned.
 The nonuniform points may be used for multiple transforms.

Inputs:
    xj     vector of x-coords of all nonuniform points
    yj     empty (if dim<2), or vector of y-coords of all nonuniform points
    zj     empty (if dim<3), or vector of z-coords of all nonuniform points
    s      vector of x-coords of all nonuniform frequency targets
    t      empty (if dim<2), or vector of y-coords of all frequency targets
    u      empty (if dim<3), or vector of z-coords of all frequency targets
Input/Outputs:
    plan   finufft_plan object

Notes:
 * For type 1 and 2, the values in xj (and if nonempty, yj and zj) must
   lie in the interval [-3pi,3pi). For type 1 they are "sources", but for
   type 2, "targets". In contrast, for type 3 there are no restrictions other
   than the resulting size of the internal fine grids.
 * s (and t and u) are only relevant for type 3, and may be omitted otherwise
 * The matlab vectors xj,... and s,... should not be changed before calling
   future execute calls, because the plan stores only pointers to the
   arrays (they are not duplicated internally).
 * The precision (double/single) of all inputs must match that chosen at the
   plan stage using opts.floatprec, otherwise an error is raised.


3) EXECUTE   execute single or many-vector FINUFFT transforms in a plan.
```

```
result = plan.execute(data_in);

 For plan a previously created finufft_plan object also containing all
 needed nonuniform point coordinates, do a single (or if ntrans>1 in the
 plan stage, multiple) NUFFT transform(s), with the strengths or Fourier
 coefficient inputs vector(s) from data_in. The result of the transform(s)
 is returned as a (possibly multidimensional) array.

Inputs:
    plan      finufft_plan object
    data_in   strengths (types 1 or 3) or Fourier coefficients (type 2)
              vector, matrix, or array of appropriate size. For type 1 and 3,
              this is either a length-M vector (where M is the length of xj),
              or an (M,ntrans) matrix when ntrans>1. For type 2, in 1D this is
              length-ms, in 2D size (ms,mt), or in 3D size (ms,mt,mu), or
              each of these with an extra last dimension ntrans if ntrans>1.
Outputs:
    result    vector of output strengths at targets (types 2 or 3), or array
              of Fourier coefficients (type 1), or, if ntrans>1, a stack of
              such vectors or arrays, of appropriate size.
              Specifically, if ntrans=1, for type 1, in 1D
              this is a length-ms column vector, in 2D a matrix of size
              (ms,mt), or in 3D an array of size (ms,mt,mu); for types 2 and 3
              it is a column vector of length M (the length of xj in type 2),
              or nk (the length of s in type 3). If ntrans>1 its is a stack
              of such objects, ie, it has an extra last dimension ntrans.

Notes:
 * The precision (double/single) of all inputs must match that chosen at the
   plan stage using opts.floatprec, otherwise an error is raised.


4) To deallocate (delete) a nonuniform FFT plan, use delete(plan)

This deallocates all stored FFTW plans, nonuniform point sorting arrays,
 kernel Fourier transforms arrays, etc.
```

PYTHON INTERFACE

13.1 Quick-start examples

The easiest way to install is to run `pip install finufft`, which downloads and installs the latest precompiled binaries from PyPI. Please note that the `finufftpy` package is obsolete. If you would like to compile from source, see *the Python installation instructions*.

To calculate a 1D type 1 transform, from nonuniform to uniform points, we import `finufft`, specify the nonuniform points x, their strengths c, and call `nufft1d1`:

```python
import numpy as np
import finufft

# number of nonuniform points
M = 100000

# the nonuniform points
x = 2 * np.pi * np.random.uniform(size=M)

# their complex strengths
c = (np.random.standard_normal(size=M)
    + 1J * np.random.standard_normal(size=M))

# desired number of Fourier modes (uniform outputs)
N = 200000

# calculate the transform
f = finufft.nufft1d1(x, c, N)
```

The input here is a set of complex strengths c, which are used to approximate (1) in *Mathematical definitions of transforms*. That approximation is stored in f, which is indexed from `-N // 2` up to `N // 2 - 1` (since N is even; if odd it would be `-(N - 1) // 2` up to `(N - 1) // 2`). The approximation is accurate to a tolerance of `1e-6`, which is the default tolerance of `nufft1d1`. It can be modified using the `eps` argument:

```python
# calculate the transform to higher accuracy
f = finufft.nufft1d1(x, c, N, eps=1e-12)
```

Note, however, that a lower tolerance (that is, a higher accuracy) results in a slower transform. See `python/examples/simple1d1.py` for the demo code that includes a basic math test (useful to check both the math and the indexing).

For higher dimensions, we would specify point locations in more than one dimension:

```
# 2D nonuniform points (x,y coords)
x = 2 * np.pi * np.random.uniform(size=M)
y = 2 * np.pi * np.random.uniform(size=M)

# desired number of Fourier modes (in x, y directions respectively)
N1 = 1000
N2 = 2000

# the 2D transform outputs f array of shape (N1, N2)
f = finufft.nufft2d1(x, y, c, (N1, N2))
```

See `python/examples/simple2d1.py` for the demo code that includes a basic math test (useful to check both the math and the indexing).

We can also go the other way, from uniform to non-uniform points, using a type 2 transform:

```
# input Fourier coefficients
f = (np.random.standard_normal(size=(N1, N2))
     + 1J * np.random.standard_normal(size=(N1, N2)))

# calculate the 2D type 2 transform
c = finufft.nufft2d2(x, y, f)
```

Now the output is a complex vector of length `M` approximating (2) in *Mathematical definitions of transforms*, that is the adjoint (but not inverse) of (1). (Note that the default sign in the exponential is negative for type 2 in the Python interface.)

In addition to tolerance `eps`, we can adjust other options for the transform. These are listed in *Options parameters* and are specified as keyword arguments in the Python interface. For example, to change the mode ordering to FFT style (that is, in each dimension `Ni = N1` or `N2`, the indices go from `0` to `Ni // 2 - 1`, then from `-Ni // 2` to `-1`, since each `Ni` is even), we call

```
f = finufft.nufft2d1(x, y, c, (N1, N2), modeord=1)
```

We can also specify a preallocated output array using the `out` keyword argument. This would be done by

```
# allocate the output array
f = np.empty((N1, N2), dtype='complex128')

# calculate the transform
finufft.nufft2d1(x, y, c, out=f)
```

In this case, we do not need to specify the output shape since it can be inferred from `f`.

Note that the above functions are all vectorized, which means that they can take multiple inputs stacked along the first dimension (that is, in row-major order) and process them simultaneously. This can bring significant speedups for small inputs by avoiding multiple short calls to FINUFFT. For the 2D type 1 vectorized interface, we would call

```
# number of transforms
K = 4

# generate K stacked coefficient arrays
c = (np.random.standard_normal(size=(K, M))
     + 1J * np.random.standard_normal(size=(K, M)))
```

(continues on next page)

```
# calculate the K transforms simultaneously (K is inferred from c.shape)
f = finufft.nufft2d1(x, y, c, (N1, N2))
```

The output array `f` would then have the shape (`K, N1, N2`). See the complete demo in `python/examples/many2d1.py`.

More fine-grained control can be obtained using the plan (or *guru*) interface. Instead of preparing the transform, setting the nonuniform points, and executing the transform all at once, these steps are seperated into different function calls. This can speed up calculations if multiple transforms are executed for the same grid size, since the same FFTW plan can be reused between calls. Additionally, if the same nonuniform points are reused between calls, we gain an extra speedup since the points only have to be sorted once. To perform the call above using the plan interface, we would write

```
# specify type 1 transform
nufft_type = 1

# instantiate the plan (note ntrans must be set here)
plan = finufft.Plan(nufft_type, (N1, N2), n_trans=K)

# set the nonuniform points
plan.setpts(x, y)

# execute the plan
f = plan.execute(c)
```

See the complete demo in `python/examples/guru2d1.py`. All interfaces support both single and double precision, but for the plan, this must be specified at initialization time using the `dtype` argument

```
# convert input data to single precision
x = x.astype('float32')
y = y.astype('float32')
c = c.astype('complex64')

# instantiate the plan and set the points
plan = finufft.Plan(nufft_type, (N1, N2), n_trans=K, dtype='float32')
plan.setpts(x, y)

# execute the plan, giving single-precision output
f = plan.execute(c)
```

See the complete demo, with math test, in `python/examples/guru2d1f.py`.

13.2 Full documentation

The Python interface to FINUFFT is divided into two parts: the simple interface (through the `nufft*` functions) and the more advanced plan interface (through the `Plan` class). The former allows the user to perform an NUFFT in a single call while the latter allows for more efficient reuse of resources when the same NUFFT is applied several times to different data by saving FFTW plans, sorting the nonuniform points, and so on.

finufft.**nufft1d1**(*x*, *c*, *n_modes=None*, *out=None*, *eps=1e-06*, *isign=1*, ***kwargs*)
     1D type-1 (nonuniform to uniform) complex NUFFT

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

    for -N1/2 <= k1 <= (N1-1)/2
```

Parameters

- **x** (`float[M]`) – nonuniform points, valid only in [-3pi, 3pi].

- **c** (`complex[M] or complex[ntransf, M]`) – source strengths.

- **n_modes** (`integer or integer tuple of length 1, optional`) – number of uniform Fourier modes requested (N1,). May be even or odd; in either case, modes k1 are integers satisfying -N1/2 <= k1 <= (N1-1)/2. Must be specified if `out` is not given.

- **out** (`complex[N1] or complex[ntransf, N1], optional`) – output array for Fourier mode values. If n_modes is specifed, the shape must match, otherwise **n_modes** is inferred from **out**.

- **eps** (`float, optional`) – precision requested (>1e-16).

- **isign** (`int, optional`) – if non-negative, uses positive sign in exponential, otherwise negative sign.

- ****kwargs** (`optional`) – for more options, see *Options parameters*.

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

Returns The resulting array.

Return type complex[N1] or complex[ntransf, N1]

Example:

```
# number of nonuniform points
M = 100

# the nonuniform points
x = 2 * np.pi * np.random.uniform(size=M)

# their complex strengths
c = (np.random.standard_normal(size=M)
     + 1J * np.random.standard_normal(size=M))

# desired number of Fourier modes
N1 = 50

# calculate the type-1 NUFFT
f = finufft.nufft1d1(x, c, (N1, ))
```

See also python/examples/simple1d1.py, python/examples/simpleopts1d1.py.

finufft.**nufft1d2**(*x*, *f*, *out=None*, *eps=1e-06*, *isign=- 1*, ***kwargs*)
    1D type-2 (uniform to nonuniform) complex NUFFT

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

    for j = 0, ..., M-1, where the sum is over -N1/2 <= k1 <= (N1-1)/2
```

Parameters

- **x** (`float[M]`) – nonuniform points, valid only in [-3pi, 3pi].

- **f** (`complex[N1] or complex[ntransf, N1]`) – Fourier mode coefficients, where N1 may be even or odd. In either case the mode indices k1 satisfy -N1/2 <= k1 <= (N1-1)/2.

- **out** (`complex[M] or complex[ntransf, M], optional`) – output array at targets.

- **eps** (`float, optional`) – precision requested (>1e-16).

- **isign** (`int, optional`) – if non-negative, uses positive sign in exponential, otherwise negative sign.

- ****kwargs** (`optional`) – for more options, see *Options parameters*.

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

Returns The resulting array.

Return type complex[M] or complex[ntransf, M]

Example:

```
# number of nonuniform points
M = 100

# the nonuniform points
x = 2 * np.pi * np.random.uniform(size=M)

# number of Fourier modes
N1 = 50

# the Fourier mode coefficients
f = (np.random.standard_normal(size=(N1, ))
    + 1J * np.random.standard_normal(size=(N1, )))

# calculate the type-2 NUFFT
c = finufft.nufft1d2(x, f)
```

See also python/test/accuracy_speed_tests.py.

finufft.**nufft1d3**(*x*, *c*, *s*, *out=None*, *eps=1e-06*, *isign=1*, ***kwargs*)
    1D type-3 (nonuniform to nonuniform) complex NUFFT

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

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

Parameters

- **x** (*float[M]*) – nonuniform source points.

- **c** (*complex[M] or complex[ntransf, M]*) – source strengths.

- **s** (*float[N]*) – nonuniform target points.

- **out** (*complex[N] or complex[ntransf, N]*) – output values at target frequencies.

- **eps** (*float, optional*) – precision requested (>1e-16).

- **isign** (*int, optional*) – if non-negative, uses positive sign in exponential, otherwise negative sign.

- ****kwargs** (*optional*) – for more options, see *Options parameters*.

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

Returns The resulting array.

Return type complex[M] or complex[ntransf, M]

Example:

```
# number of source points
M = 100

# number of target points
N = 200

# the source points
x = 2 * np.pi * np.random.uniform(size=M)

# the target points
s = 2 * np.pi * np.random.uniform(size=N)

# their complex strengths
c = (np.random.standard_normal(size=M)
     + 1J * np.random.standard_normal(size=M))

# calcuate the type-3 NUFFT
f = finufft.nufft1d3(x, c, s)
```

See also python/test/accuracy_speed_tests.py.

finufft.**nufft2d1**(*x, y, c, n_modes=None, out=None, eps=1e-06, isign=1, **kwargs*)
   2D type-1 (nonuniform to uniform) complex NUFFT

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

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

Parameters

- **x** (`float[M]`) – nonuniform points, valid only in [-3pi, 3pi].

- **y** (`float[M]`) – nonuniform points, valid only in [-3pi, 3pi].

- **c** (`complex[M] or complex[ntransf, M]`) – source strengths.

- **n_modes** (`integer or integer tuple of length 2, optional`) – number of uni-
  form Fourier modes requested (N1, N2). May be even or odd; in either case, modes k1,
  k2 are integers satisfying -N1/2 <= k1 <= (N1-1)/2, -N2/2 <= k2 <= (N2-1)/2. Must be
  specified if `out` is not given.

- **out** (`complex[N1, N2] or complex[ntransf, N1, N2], optional`) – output array
  for Fourier mode values. If `n_modes` is specifed, the shape must match, otherwise `n_modes`
  is inferred from `out`.

- **eps** (`float, optional`) – precision requested (>1e-16).

- **isign** (`int, optional`) – if non-negative, uses positive sign in exponential, otherwise
  negative sign.

- ****kwargs** (`optional`) – for more options, see *Options parameters*.

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

> **Returns** The resulting array.
>
> **Return type** complex[N1, N2] or complex[ntransf, N1, N2]

Example:

```
# number of nonuniform points
M = 100

# the nonuniform points
x = 2 * np.pi * np.random.uniform(size=M)
y = 2 * np.pi * np.random.uniform(size=M)

# their complex strengths
c = (np.random.standard_normal(size=M)
    + 1J * np.random.standard_normal(size=M))

# desired number of Fourier modes
N1, N2 = 50, 75

# calculate the type-1 NUFFT
f = finufft.nufft2d1(x, y, c, (N1, N2))
```

See also python/examples/simple2d1.py, python/examples/many2d1.py.

finufft.**nufft2d2**(*x, y, f, out=None, eps=1e-06, isign=- 1, **kwargs*)
    2D type-2 (uniform to nonuniform) complex NUFFT

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

13.2. Full documentation   **81**

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

Parameters

- **x** (`float[M]`) – nonuniform points, valid only in [-3pi, 3pi].

- **y** (`float[M]`) – nonuniform points, valid only in [-3pi, 3pi].

- **f** (`complex[N1, N2] or complex[ntransf, N1, N2]`) – Fourier mode coefficients, where N1, N2 may be even or odd. In either case the mode indices k1, k2 satisfy -N1/2 <= k1 <= (N1-1)/2, -N2/2 <= k2 <= (N2-1)/2.

- **out** (`complex[M] or complex[ntransf, M], optional`) – output array at targets.

- **eps** (`float, optional`) – precision requested (>1e-16).

- **isign** (`int, optional`) – if non-negative, uses positive sign in exponential, otherwise negative sign.

- ****kwargs** (`optional`) – for more options, see *Options parameters*.

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

Returns The resulting array.

Return type complex[M] or complex[ntransf, M]

Example:

```
# number of nonuniform points
M = 100

# the nonuniform points
x = 2 * np.pi * np.random.uniform(size=M)
y = 2 * np.pi * np.random.uniform(size=M)

# number of Fourier modes
N1, N2 = 50, 75

# the Fourier mode coefficients
f = (np.random.standard_normal(size=(N1, N2))
     + 1J * np.random.standard_normal(size=(N1, N2)))

# calculate the type-2 NUFFT
c = finufft.nufft2d2(x, y, f)
```

See also `python/test/accuracy_speed_tests.py`.

finufft.**nufft2d3**(*x*, *y*, *c*, *s*, *t*, *out=None*, *eps=1e-06*, *isign=1*, ***kwargs*)
    2D type-3 (nonuniform to nonuniform) complex NUFFT

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

```
        j=0

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

Parameters

- **x** (`float[M]`) – nonuniform source points.
- **y** (`float[M]`) – nonuniform source points.
- **c** (`complex[M] or complex[ntransf, M]`) – source strengths.
- **s** (`float[N]`) – nonuniform target points.
- **t** (`float[N]`) – nonuniform target points.
- **out** (`complex[N] or complex[ntransf, N]`) – output values at target frequencies.
- **eps** (`float, optional`) – precision requested (>1e-16).
- **isign** (`int, optional`) – if non-negative, uses positive sign in exponential, otherwise negative sign.
- ****kwargs** (`optional`) – for more options, see *Options parameters*.

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

Returns The resulting array.

Return type complex[M] or complex[ntransf, M]

Example:

```python
# number of source points
M = 100

# number of target points
N = 200

# the source points
x = 2 * np.pi * np.random.uniform(size=M)
y = 2 * np.pi * np.random.uniform(size=M)

# the target points
s = 2 * np.pi * np.random.uniform(size=N)
t = 2 * np.pi * np.random.uniform(size=N)

# their complex strengths
c = (np.random.standard_normal(size=M)
     + 1J * np.random.standard_normal(size=M))

# calcuate the type-3 NUFFT
f = finufft.nufft2d3(x, y, c, s, t)
```

See also `python/test/accuracy_speed_tests.py`.

finufft.**nufft3d1**(*x, y, z, c, n_modes=None, out=None, eps=1e-06, isign=1, **kwargs*)

3D type-1 (nonuniform to uniform) complex NUFFT

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

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

Parameters

- **x** (*float[M]*) – nonuniform points, valid only in [-3pi, 3pi].

- **y** (*float[M]*) – nonuniform points, valid only in [-3pi, 3pi].

- **z** (*float[M]*) – nonuniform points, valid only in [-3pi, 3pi].

- **c** (*complex[M] or complex[ntransf, M]*) – source strengths.

- **n_modes** (*integer or integer tuple of length 3, optional*) – number of uniform Fourier modes requested (N1, N2, N3). May be even or odd; in either case, modes k1, k2, k3 are integers satisfying -N1/2 <= k1 <= (N1-1)/2, -N2/2 <= k2 <= (N2-1)/2, -N3/2 <= k3 <= (N3-1)/2. Must be specified if `out` is not given.

- **(complex[N1** (*out*) – optional): output array for Fourier mode values. If `n_modes` is specifed, the shape must match, otherwise `n_modes` is inferred from `out`.

- **N2** – optional): output array for Fourier mode values. If `n_modes` is specifed, the shape must match, otherwise `n_modes` is inferred from `out`.

- **complex[ntransf** (*N3] or*) – optional): output array for Fourier mode values. If `n_modes` is specifed, the shape must match, otherwise `n_modes` is inferred from `out`.

- **N1** – optional): output array for Fourier mode values. If `n_modes` is specifed, the shape must match, otherwise `n_modes` is inferred from `out`.

- **N2** – optional): output array for Fourier mode values. If `n_modes` is specifed, the shape must match, otherwise `n_modes` is inferred from `out`.

- **N3]** – optional): output array for Fourier mode values. If `n_modes` is specifed, the shape must match, otherwise `n_modes` is inferred from `out`.

:param [optional): output array for Fourier mode values. If `n_modes`] is specifed, the shape must match, otherwise `n_modes` is inferred from `out`.

Parameters

- **eps** (*float, optional*) – precision requested (>1e-16).

- **isign** (*int, optional*) – if non-negative, uses positive sign in exponential, otherwise negative sign.

- ****kwargs** (*optional*) – for more options, see *Options parameters*.

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

Returns The resulting array.

> **Return type** complex[N1, N2, N3] or complex[ntransf, N1, N2, N3]

Example:

```
# number of nonuniform points
M = 100

# the nonuniform points
x = 2 * np.pi * np.random.uniform(size=M)
y = 2 * np.pi * np.random.uniform(size=M)
z = 2 * np.pi * np.random.uniform(size=M)

# their complex strengths
c = (np.random.standard_normal(size=M)
     + 1J * np.random.standard_normal(size=M))

# desired number of Fourier modes
N1, N2, N3 = 50, 75, 100

# calculate the type-1 NUFFT
f = finufft.nufft3d1(x, y, z, c, (N1, N2, N3))
```

See also `python/test/accuracy_speed_tests.py`.

finufft.**nufft3d2**(*x*, *y*, *z*, *f*, *out=None*, *eps=1e-06*, *isign=- 1*, ***kwargs*)

> 3D type-2 (uniform to nonuniform) complex NUFFT

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

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

> **Parameters**
>
> - **x** (`float[M]`) – nonuniform points, valid only in [-3pi, 3pi].
>
> - **y** (`float[M]`) – nonuniform points, valid only in [-3pi, 3pi].
>
> - **z** (`float[M]`) – nonuniform points, valid only in [-3pi, 3pi].
>
> - **f** (`complex[N1, N2, N3] or complex[ntransf, N1, N2, N3]`) – Fourier mode co-efficients, where N1, N2, N3 may be even or odd. In either case the mode indices k1, k2, k3 satisfy -N1/2 <= k1 <= (N1-1)/2, -N2/2 <= k2 <= (N2-1)/2, -N3/2 <= k3 <= (N3-1)/2.
>
> - **out** (`complex[M] or complex[ntransf, M], optional`) – output array at targets.
>
> - **eps** (`float, optional`) – precision requested (>1e-16).
>
> - **isign** (`int, optional`) – if non-negative, uses positive sign in exponential, otherwise negative sign.
>
> - ****kwargs** (`optional`) – for more options, see *Options parameters*.

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

Returns The resulting array.

Return type complex[M] or complex[ntransf, M]

Example:

```
# number of nonuniform points
M = 100

# the nonuniform points
x = 2 * np.pi * np.random.uniform(size=M)
y = 2 * np.pi * np.random.uniform(size=M)
z = 2 * np.pi * np.random.uniform(size=M)

# number of Fourier modes
N1, N2, N3 = 50, 75, 100

# the Fourier mode coefficients
f = (np.random.standard_normal(size=(N1, N2, N3))
     + 1J * np.random.standard_normal(size=(N1, N2, N3)))

# calculate the type-2 NUFFT
c = finufft.nufft3d2(x, y, z, f)
```

See also python/test/accuracy_speed_tests.py.

finufft.**nufft3d3**(*x, y, z, c, s, t, u, out=None, eps=1e-06, isign=1, **kwargs*)

3D type-3 (nonuniform to nonuniform) complex NUFFT

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

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

Parameters

- **x** (*float[M]*) – nonuniform source points.

- **y** (*float[M]*) – nonuniform source points.

- **z** (*float[M]*) – nonuniform source points.

- **c** (*complex[M] or complex[ntransf, M]*) – source strengths.

- **s** (*float[N]*) – nonuniform target points.

- **t** (*float[N]*) – nonuniform target points.

- **u** (*float[N]*) – nonuniform target points.

- **out** (*complex[N] or complex[ntransf, N]*) – output values at target frequencies.

- **eps** (*float, optional*) – precision requested (>1e-16).

- **isign** (*int, optional*) – if non-negative, uses positive sign in exponential, otherwise negative sign.

- ****kwargs** (*optional*) – for more options, see *Options parameters*.

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

> **Returns** The resulting array.
>
> **Return type** complex[M] or complex[ntransf, M]

Example:

```
# number of source points
M = 100

# number of target points
N = 200

# the source points
x = 2 * np.pi * np.random.uniform(size=M)
y = 2 * np.pi * np.random.uniform(size=M)
z = 2 * np.pi * np.random.uniform(size=M)

# the target points
s = 2 * np.pi * np.random.uniform(size=N)
t = 2 * np.pi * np.random.uniform(size=N)
u = 2 * np.pi * np.random.uniform(size=N)

# their complex strengths
c = (np.random.standard_normal(size=M)
     + 1J * np.random.standard_normal(size=M))

# calcuate the type-3 NUFFT
f = finufft.nufft3d3(x, y, z, c, s, t, u)
```

See also `python/test/accuracy_speed_tests.py`.

class `finufft.Plan`(*nufft_type*, *n_modes_or_dim*, *n_trans=1*, *eps=1e-06*, *isign=None*, ***kwargs*)
    A non-uniform fast Fourier transform (NUFFT) plan

The `Plan` class lets the user exercise more fine-grained control over the execution of an NUFFT. First, the plan is created with a certain set of parameters (type, mode configuration, tolerance, sign, number of simultaneous transforms, and so on). Then the nonuniform points are set (source or target depending on the type). Finally, the plan is executed on some data, yielding the desired output.

In the simple interface, all these steps are executed in a single call to the `nufft*` functions. The benefit of separating plan creation from execution is that it allows for plan reuse when certain parameters (like mode configuration) or nonuniform points remain the same between different NUFFT calls. This becomes especially important for small inputs, where execution time may be dominated by initialization steps such as allocating and FFTW plan and sorting the nonuniform points.

Example:

```
import numpy as np
import finufft

# set up parameters
n_modes = (1000, 2000)
```

13.2. Full documentation 87

```
n_pts = 100000
nufft_type = 1
n_trans = 4

# generate nonuniform points
x = 2 * np.pi * np.random.uniform(size=n_pts)
y = 2 * np.pi * np.random.uniform(size=n_pts)

# generate source strengths
c = (np.random.standard_normal(size=(n_trans, n_pts)),
     + 1J * np.random.standard_normal(size=(n_trans, n_pts)))

# initialize the plan
plan = finufft.Plan(nufft_type, n_modes, n_trans)

# set the nonuniform points
plan.setpts(x, y)

# execute the plan
f = plan.execute(c)
```

Also see python/examples/guru1d1.py and python/examples/guru2d1.py.

> **Parameters**
>
> - **nufft_type** (`int`) – type of NUFFT (1, 2, or 3).
> - **n_modes_or_dim** (`int or tuple of ints`) – if nufft_type is 1 or 2, this should be a tuple specifying the number of modes in each dimension (for example, `(50, 100)`), otherwise, if *nufft_type`* is 3, this should be the number of dimensions (between 1 and 3).
> - **n_trans** (`int, optional`) – number of transforms to compute simultaneously.
> - **eps** (`float, optional`) – precision requested (>1e-16).
> - **isign** (`int, optional`) – if non-negative, uses positive sign exponential, otherwise negative sign.
> - ****kwargs** (`optional`) – for more options, see *Options parameters*.

setpts(*x=None*, *y=None*, *z=None*, *s=None*, *t=None*, *u=None*)
> Set the nonuniform points
>
> For type 1, this sets the coordinates of the M nonuniform source points, for type 2, it sets the coordinates of the M target points, and for type 3 it sets both the M source points and the N target points.
>
> The dimension of the plan determines the number of arguments supplied. For example, if dim == 2, we provide x and y (as well as s and t for a type-3 transform).
>
> > **Parameters**
> >
> > - **x** (`float[M]`) – first coordinate of the nonuniform points (source for type 1 and 3, target for type 2).
> > - **y** (`float[M], optional`) – second coordinate of the nonuniform points (source for type 1 and 3, target for type 2).
> > - **z** (`float[M], optional`) – third coordinate of the nonuniform points (source for type 1 and 3, target for type 2).

- **s** (*float[N], optional*) – first coordinate of the nonuniform points (target for type 3).

- **t** (*float[N], optional*) – second coordinate of the nonuniform points (target for type 3).

- **u** (*float[N], optional*) – third coordinate of the nonuniform points (target for type 3).

execute(*data*, *out=None*)

 Execute the plan

Performs the NUFFT specified at plan instantiation with the points set by `setpts`. For type-1 and type-3 transforms, the input is a set of source strengths, while for a type-2 transform, it consists of an array of size `n_modes`. If `n_transf` is greater than one, `n_transf` inputs are expected, stacked along the first axis.

 Parameters

- **data** (*complex[M], complex[n_transf, M], complex[n_modes], or complex[n_transf, n_modes]*) – The input source strengths (type 1 and 3) or source modes (type 2).

- **out** (*complex[n_modes], complex[n_transf, n_modes], complex[M], or complex[n_transf, M], optional*) – The array where the output is stored. Must be of the right size.

 Returns The output array of the transform(s).

 Return type complex[n_modes], complex[n_transf, n_modes], complex[M], or complex[n_transf, M]

FOURTEEN

JULIA INTERFACE

Ludvig af Klinteberg has built FINUFFT.jl, an interface from the Julia language. This package supports 32-bit and 64-bit precision, and automatically downloads and runs pre-built binaries of the FINUFFT library for Linux, macOS, Windows and FreeBSD (for a full list see finufft_jll).

CHANGELOG

```
List of features / changes made / release notes, in reverse chronological order

* fortran examples: avoided clash with keywords "type" and "null", and correct
  creation of null ptr for default opts (issues #195-196, Jiri Kulda).
* fixed modeord=1 failure for type 3 even though should never be used anyway
  (issue #194).
* fixed spreadcheck NaN failure to detect bug introduced in 2.0.3 (9566511).
* Dan Fortunato found and fixed MATLAB setpts temporary array loss, issue #185.

V 2.0.3 (4/22/20)

* finufft (plan) now thread-safe via OMP lock (if nthr=1 and -DFFTW_PLAN_SAFE)
  + new example/threadsafe*.cpp demos. Needs FFTW>=3.3.6 (Issues #72 #180 #183)
* fixed bug in checkbounds that falsely reported NU pt as invalid if exactly 1
  ULP below +pi, for certain N values only, egad! (Issue #181)
* GH workflows continuous integration (CI) in four setups (linux, osx*2, mingw)
* fixed memory leak in type 3.
* corrected C guru execute documentation.

V 2.0.2 (12/5/20)

* fixed spreader segfault in obscure use case: single-precision N1>1e7, where
  rounding error is O(1) anyway. Now uses consistent int(ceil()) grid index.
* Improved large-thread scaling of type-1 (spreading) via transition from OMP
  critical to atomic add_wrapped_subgrid() operations; thanks Rob Blackwell.
* Increased heuristic t1 spreader max_subproblem_size, faster in 2D, 3D, and
  allowed this and the above atomic threshold to be controlled as nufft_opts.
* Removed MAX_USEFUL_NTHREADS from defs.h and all code, for simplicity, since
  large thread number now scales better.
* multithreaded one-mode accuracy test in C++ tests, t1 & t3, for faster tests.

V 2.0.1 (10/6/20)

* python (under-the-hood) interfacing changed from pybind11 to cleaner ctypes.
* non-stochastic test/*.cpp routines, zeroing small chance of incorrect failure
* Windows compatible makefile
* mac OSX improved installation instructions and make.inc.*

V 2.0.0 (8/28/20)
```

```
* major changes to code, internally, and major improvements to operation and
  language interfaces.

        WARNING!: Here are all the interface compatibility changes from 1.1.2:
        - opts (nufft_opts) is now always passed as a pointer in C++/C, not
          pass-by-reference as in v1.1.2 or earlier.
        - Fortran simple calls are now finufft?d?(..) not finufft?d?_f(..), and
          they add a penultimate opts argument.
        - Python module name is now finufft not finufftpy, and the interface has
          been completely changed (allowing major improvements, see below).
        - ier=1 is now a warning not an error; this indicates requested tol
          was too small, but that a transform *was* done at the best possible
          accuracy.
        - opts.fftw directly controls the FFTW plan mode consistently in all
          language interfaces (thus changing the meaning of fftw=0 in MATLAB).
        - Octave now needs version >= 4.4, since OO features used by guru.

        These changes were deemed necessary to rationalize and improve FINUFFT
        for the long term.
        There are also many other new interface options (many-vector, guru)
        added; see docs.
* the C++ library is now dual-precision, with distinct function interfaces for
  double vs single precision operation, that are C and C++ compatible. Under
  the hood this is achieved via simple C macros. All language interfaces now
  have dual precision options.
* completely new (although backward compatible) MATLAB/octave interface,
  including object-style wrapper around the guru interface, dual precision.
* completely new Fortran interface, allowing >2^31 sized (int64) arrays,
  all simple, many-vector and guru interface, with full options control,
  and dual precisions.
* all simple and many-vector interfaces now call guru interface, for much
  better maintainability and less code repetition.
* new guru interface, by Andrea Malleo and Alex Barnett, allowing easier
  language wrapping and control of point-setting, reuse of sorting and FFTW
  plans. This finally bypasses the 0.1ms/thread cost of FFTW looking up previous
  wisdom, which slowed down performance for many small problems.
* removed obsolete -DNEED_EXTERN_C flag.
* major rewrite of documentation, plus tutorial application examples in MATLAB.
* numdiff dependency is removed for pass-fail library validation.
* new (professional!) logo for FINUFFT. Sphinx HTML and PDF aesthetics.

V 1.1.2 (1/31/20)

* Ludvig's padding of Horner loop to w=4n, speeds up kernel, esp for GCC5.4.
* Bansal's Mingw32 python patches.

V 1.1.1 (11/2/18)

* Mac OSX installation on clang and gcc-8, clearer install docs.
* LIBSOMP split off in makefile.
* printf(...%lld..) w/ long long typecast
* new basic passfail tester
```

```
* precompiled binaries


V 1.1 (9/24/18)


* NOTE TO USERS: changed interface for setting default opts in C++ and C, from
  pass by reference to pass by value of a pointer (see docs/). Unifies C++/C
  interfaces in a clean way.
* fftw3_omp instead of fftw3_threads (on linux), is faster.
* rationalized header files.


V 1.0.1 (9/14/18)


* Ludvig's removal of omp chunksize in dir=2, another 20%+ speedup.
* Matlab doesn't change omp internal state.


V 1.0 (8/20/18)


* repo transferred to flatironinstitute
* usage doc simpler
* 2d1many and 2d2many interfaces by Melody Shih, for multiple vectors with same
  nonuniform points. All tests and docs for these interfaces.
* horner optimized kernel for sigma=5/4 (low upsampling), to go along with the
  default sigma=2. Cmdline arg to change sigma in finufft?d_test.
* simplified various int types: only BIGINT remains.
* clearer docs.
* remaining C interfaces, with opts control.


V 0.99 (4/24/18)


* piecewise polynomial kernel evaluation by Horner, for faster spreading esp at
  low accuracy and 1d or 2d.
* various heuristic decisions re whether to sort, and if sorting is single or
  multi-threaded.
* single-precision libs get an "f" suffix so can coexist with double-prec.


V 0.98 (3/1/18)


* makefile includes make.inc for OS-specific defs.
* decided that, since Legendre nodes code of GLR alg by Hale/Burkhardt is LGPL
  licensed, our use (not changing source) is not a "derived work", therefore
  ok under our Apache v2 license. See:
  https://tldrlegal.com/license/gnu-lesser-general-public-license-v3-(lgpl-3)
  https://www.apache.org/licenses/GPL-compatibility.html
  https://softwareengineering.stackexchange.com/questions/233571/
    open-source-what-is-the-definition-of-derivative-work-and-how-does-it-impact
  * fixed MATLAB FFTW incompat alloc crash, by hack of Joakim, calling fft()
  first.
* python tests fixed, brought into makefile.
* brought in af Klinteberg spreader optimizations & SSE tricks.
* logo


V 0.97 (12/6/17)
```

```
* tidied all docs -> readthedocs.io host. README.md now a stub. TODO tidied.
* made sort=1 in tests for xeon (used to be 0)
* removed mcwrap and python dirs
* changed name of py routines to nufft* from finufft*
* python interfaces doc, up-to-date. Removed ms,.. from type-2 interfaces.
* removed RESCALEs from lower dims in bin_sort, speeds up a few % in 1D.
* allowed NU pts to be currectly folded from +-1 periods from central box, as
  per David Stein request. Adds 5% to time at 1e-2 accuracy, less at higher acc.
* corrected dynamic C++ array allocs in spreader (some made static, 5% speedup)
* removed all C++11 dependencies, mainly that opts structs are all explicitly
  initialized.
* fixed python interface to have chkbnds.
* tidied MEX interface
* removed memory leaks (!)
* opts.modeord implemented and exposed to matlab/python interfaces. Also removes
  looping backwards in RAM in deconvolveshuffle.


V 0.96 (10/15/17)


* apache v2 license, exposed flags in python interface.


V 0.95 (10/2/17)


* brought in JFM's in-package python wrapper & doc, create lib-static dir,
  removed devel dir.


V 0.9: (6/17/17)


* adapted adv4 into main code, inner loops separate by dim, kill
  the current spreader. Incorporate old ideas such as: checkerboard
  per-thread grid cuboids, compare speed in 2d and 3d against
  current 1d slicing. See cnufftspread:set_thread_index_box()
* added FFTW_MEAS vs FFTW_EST (set as default) opts flag in nufft_opts, and
  matlab/python interfaces
* removed opts.maxnalloc in favor of #defined MAX_NF
* fixed the 1-target case in type-3, all dims, to avoid nan; clarified logic
  for handling X=0 and/or S=0. 6/12/17
* changed arraywidcen to snap to C=0 if relative shift is <0.1, avoids cexps in
  type-3.
* t3: if C1 < X1/10 and D1 < S1/10 then don't rephase. Same for d=2,3.
* removed the 1/M type-1 prefactor, also in all test routines. 6/6/17
* removed timestamp-based make decision whether to rebuild matlab/finufft.cpp,
  since git clone creates files with random timestamp order!
* theory work on exp(sqrt) being close to PSWF. Analysis.
* fix issue w/ needing mwrap when do make matlab.
* makefile has variables customizing openmp and precision, non-omp tested
* fortran single-prec demos (required all direct ft's in single prec too!)
* examples changed to err rel to max F.
* matlab interface control of opts.spread_sort.
* matlab interface using doubles as big ints w/ correct typecasting.
* twopispread removed, used flag in spread_opts for [-pi,pi] input instead.
```

```
* testfinufft* use same integer type INT as for interfaces, typecast all %ld in
  printf warnings, use omp rand array filling
* INT64 for necessary size-setting arrays, removed all %lf printf warnings in
  finufft*
* all internal array indexing is BIGINT, switchable from long long to int via
  SMALLINT compile flag (default off, see utils.h)
* all integers in interfaces are type INT, default 64-bit, switchable to 32 bit
  by INTERGER32 compile flag (see utils.h)
* test big probs (speed, crashing) and decide if BIGINT is long long or int?
  slows any array access down, or spreading? allows I/O sizes
  (M, N1*N2*N3) > 2^31. Note June-Yub int*8 in nufft-1.3.x slowed things by
  factor 2-3.
* tidy up spreader to be BIGINT = long long compatible and test > 2^31.
* spreadtest parallel rand()
* sort flag passed to spreader via finufft, and test scripts check if Xeon
  (-> sort=0)
* opts in the manual
* removed all xk2, dNU2 sorted arrays, and not-needed dims y,z; halved RAM usage


V 0.8: (3/27/17)


* bnderr checking done in dir=1,2 main loops, not before.
* all kx2, dNU2 arrays removed, just done by permutation index when needed.
* MAC OSX test, makefile, instructions.
* matlab wrappers in 3D
* matlab wrappers, mcwrap issue w/ openmp, mex, and subdirs. Ship mex
  executables for linux. Link to .a
* matlab wrappers need ier output? yes, and internal omp numthreads control
  (since matlab's is poor)
* wrappers for MEX octave, instructions. Ship .mex for octave.
* python wrappers - Dan Foreman-Mackey starting to add something similar to
        https://github.com/dfm/python-nufft
* check is done before attempting to malloc ridiculous array sizes, eg if a
  large product of x width and k width is requested in type 3 transforms.
* draft make python
* basic manual (txt)


V. 0.7:


* build static & shared lib
* fixed bug when Nth>Ntop
* fortran drivers use dynamic malloc to prevent stack segfaults that CMCL had
* bugs found in fortran drivers, removed
* split out devel text files (TODO, etc)
* made pass-fail test script counting crashes and numdiff fails.
* finufft?d_test have a no-timings option, and exit with ier.
* global error codes
* made finufft routines & testers return error codes rather than exit().
* dumbinput test executable
* found nan returned error for nj=0 in type-1, fixed so returns the zero array.
* fixed type 2 to not segfault when ms,mt, or mu=0, doing dir=2 0-padding right
* array utils use pointers to make which vars they write to explicit.
```

```
* don't do final type-3 rephase if C1 nan or 0.
* finished all dumbinputs, all dims
* fortran compilation fixed
* makefile self-documents
* nf1 (etc) size check before alloc, exit gracefully if exceeds RAM
* integrate into nufft_comparison, esp vs NFFT - jfm did
* simple examples, simpler than the test drivers
* fortran link via gfortran, better fortran docs
* boilerplate stuff as in CMCL page


pre-V. 0.7: (Jan-Feb 2017)


* efficient modulo in spreader, done by conditionals
* removed data-zeroing bug in t-II spreader, slowness of large arrays in t-I.
* clean dir tree
* spreader dir=1,2 math tests in 3d, then nd.
* Jeremy's request re only computing kernel vals needed (actually
  was vital for efficiency in dir=1 openmp version), Ie fix KB kereval in
  spreader so doesn't wdo 3d fill when 1 or 2 will do.
* spreader removed modulo altogether in favor of ifs
* OpenMP spreader, all dims
* multidim spreader test, command line args and bash driver
* cnufft->finufft names, except spreader still called cnufft
* make ier report accuracy out of range, malloc size errors, etc
* moved wrappers to own directories so the basic lib is clean
* fortran wrapper added ier argument
* types 1,2 in all dims, using 1d kernel for all dims.
* fix twopispread so doesn't create dummy ky,kz, and fix sort so doesn't ever
  access unused ky,kz dims.
* cleaner spread and nufft test scripts
* build universal ndim Fourier coeff copiers in C and use for finufft
* makefile opts and compiler directives to link against FFTW.
* t-I, t-II convergence params test: R=M/N and KB params
* overall scale factor understand in KB
* check J's bessel10 approx is ok. - became irrelevant
* meas speed of I_0 for KB kernel eval - became irrelevant
* understand origin of dfftpack (netlib fftpack is real*4) - not needed
* [spreader: make compute_sort_indices sensible for 1d and 2d. not needed]
* next235even for nf's
* switched pre/post-amp correction from DFT of kernel to F series (FT) of
  kernel, more accurate
* Gauss-Legendre quadrature for direct eval of kernel FT, openmp since cexp slow
* optimize q (# G-L nodes) for kernel FT eval on reg and irreg grids
  (common.cpp). Needs q a bit bigger than like (2-3x the PTR, when 1.57x is
  expected). Why?
* type 3 segfault in dumb case of nj=1 (SX product = 0). By keeping gam>1/S
* optimize that phi(z) kernel support is only +-(nspread-1)/2, so w/ prob 1 you
  only use nspread-1 pts in the support. Could gain several % speed for same acc
* new simpler kernel entirely
* cleaned up set_nf calls and removed params from within core libs
* test isign=-1 works
* type 3 in 2d, 3d
```

```
* style: headers should only include other headers needed to compile the .h;
  all other headers go in .cpp, even if that involves repetition I guess.
* changed library interface and twopispread to dcomplex
* fortran wrappers (rmdir greengard_work, merge needed into fortran)

Started: mid-January 2017.
```

DEVELOPER NOTES

- Developers needing to update documentation in general, including our readthedocs website, see `docs/README`.

- To update the version number, this needs to be done by hand in the following places:

    - `docs/conf.py` for sphinx

    - `python/setup.py` for the python pkg version

    - `include/defs.h`

    - `CHANGELOG`: don't forget to describe the new features and changes, folding lines at 80 chars.

    You will then need to update the docs as in `docs/README`.

- There are some sphinx tags in the source code, indicated by @ in comments. Please leave these alone since they are needed by the doc generation.

- If you add a new option field (recall it must be plain C style only, no special types) to `include/nufft_opts.h`, don't forget to add it to `include/finufft.fh`, `matlab/finufft.mw`, `python/finufft/_finufft.py`, and the julia interface, as well a paragraph describing its use in the docs. Also to set its default value in `src/finufft.cpp`. You will then need to update the docs as in `docs/README`.

- Developers changing MATLAB/octave interfaces or docs, also see `matlab/README`.

- For testing and performance measuring routines see `test/README` and `perftest/README`. We need more of the latter, eg, something making performance graphs that enable rapid eyeball comparison of various settings/machines.

- Continuous Integration (CI). See files for this in `.github/workflows/`. It currently tests the default `makefile` settings in linux, and three other `make.inc.*` files covering OSX and Windows (MinGW).

- **Installing MWrap**. This is needed only to rebuild the matlab/octave interfaces. MWrap is a very useful MEX interface generator by Dave Bindel, now maintained and expanded by Zydrunas Gimbutas. Make sure you have `flex` and `bison` installed to build it. As of FINUFFT v.2.0 you will need a recent (>=0.33.10) version of MWrap. Make sure to override the location of MWrap by adding a line such as:

```
MWRAP = your-path-to-mwrap-executable
```

to your `make.inc`, and then you can use the `make mex` task.

SEVENTEEN

RELATED PACKAGES

17.1 Other recommended NUFFT libraries

- 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.

- 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 is very easy to compile.

- 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 10^{-1}.

- PyNUFFT Python code supporting CPU and GPU operation. We have not compared against FINUFFT yet.

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

DEPENDENT PACKAGES, USERS, AND CITATIONS

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

18.1 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.

18.2 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, J. Comput. Phys., 409 (2020) 109353; uses 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

8. Efficient wide-field radio interferometry response. P. Arras, M. Reinecke, R. Westermann, T.A. Ensslin, Astron. Astrophys. (2020). https://doi.org/10.1051/0004-6361/202039723

9. Johannes Blaschke, Jeff Donatelli, and collaborators at NSERC/LBNL use FINUFFT and cuFINUFFT for single-particle X-ray imaging.

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

1. Davood Shamshirgar and Anna-Karin Tornberg, "Fast Ewald summation for electrostatic potentials with arbitrary periodicity", exploit our "Barnett-Magland" (BM), aka exp-sqrt (ES) window function. https://arxiv.org/abs/1712.04732

2. Martin Reinecke: codes for radio astronomy reconstruction including https://gitlab.mpcdf.mpg.de/mtr/ducc

18.3 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

5. https://arxiv.org/abs/2010.05295

NINETEEN

ACKNOWLEDGMENTS

- Jonas Krimmer - thread safety of FFTW, Windows makefile

- Timo Heister - pass/fail numdiff testing ideas

- Vladimir Rokhlin - piecewise polynomial approximation on complex boxes

Logo design: Sherry Choi with input from Alex Barnett and Lucy Reading-Ikkanda.

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

REFERENCES

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

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

[B20] Aliasing error of the $\exp(\beta\sqrt{1-z^2})$ kernel in the nonuniform fast Fourier transform. A. H. Barnett. Appl. Comput. Harmon. Anal. 51, 1-16 (2021). arxiv version

20.1 Background references

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

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

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

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

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

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

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

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

The appendix of the last of the above contains the first known published proof of the Kaiser–Bessel Fourier transform pair. This next two papers prove error estimates for sinh-type and other kernels closely related (and possibly slightly more optimal) than ours:

[PT] Uniform error estimates for the NFFT. D. Potts and M. Tasche. (2020). arxiv

[PT2] Continuous window functions for NFFT. D. Potts and M. Tasche. (2020). arxiv. In revision, Adv. Comput. Math.

In late 2020 it was pointed out to us by Piero Angeletti that the exponential of semicircle kernel developed for FINUFFT had in fact been independently proposed:

[AN] A new window based on exponential function. K. Avci and A. Nacaroğlu. 2008 Ph.D. Research in Microelectronics and Electronics, Istanbul. 69-72 (2008). doi:10.1109/RME.2008.4595727.

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

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

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

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

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

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

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

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

20.2 Talk slides

These PDF slides may be a useful introduction.