Nonuniform FFTs at Flatiron

— ∼ * OR ∗ ∼ —

Lessons from developing a small numerical library

Alex Barnett\textsuperscript{1}, with much help from: Jeremy Magland\textsuperscript{1}, Ludvig af Klinteberg (Mälardalen U.), Melody Shih (Nvidia), Joakim Andén (KTH), Libin Lu\textsuperscript{1}, Robert Blackwell (SCC), Andrea Malleo (Bloomberg), and many others...

FWAM5  Friday, October 20, 2023

\textsuperscript{1}Center for Computational Mathematics, Flatiron Institute
What is this non-uniform fast Fourier transform? (NUFFT)

1D case, “type 1” transform:

Given locations \(x_1, \ldots, x_M\), and their “strengths” \(c_1, \ldots, c_M\), return

\[
f_k = \sum_{j=1}^{M} e^{ikx_j}c_j, \quad k = -N/2, -N/2 + 1, \ldots, N/2 - 2, N/2 - 1.
\]

- Has \(N\) outputs, each a sum of \(M\) terms: naively \(O(NM)\) cost (flops)
- Off-grid version of fast Fourier transform (FFT) there \(M = N\) and \(x_j = 2\pi j / N\)
- NUFFT does this fast in \(O(M + N \log N)\) nearly linear cost
- approximates it to a user-requested \(\epsilon = 10^{-1}\) (fastest) to \(\epsilon = 10^{-14}\) (slowest)

“Type 2” is its adjoint (but not inverse!)

\[
g_j = \sum_{k=1}^{N} f_k e^{-ikx_j} U \rightarrow N\)

- evaluates a given Fourier series at arbitrary targets \(\{x_j\}\), fast

“Type 3” as type 1 but arbitrary target freqs. \(\{s_k\}\):

\[
f_k = \sum_{j=1}^{M} e^{is_kx_j}c_j U \rightarrow N\)

\(M=6\)

\(N=10\)
What is this non-uniform fast Fourier transform? (NUFFT)

1D case, “type 1” transform:

Given locations $x_1, \ldots, x_M$, and their “strengths” $c_1, \ldots, c_M$, return

$$f_k = \sum_{j=1}^{M} e^{ikx_j}c_j,$$

for $k = -\frac{N}{2}, -\frac{N}{2} + 1, \ldots, \frac{N}{2} - 2, \frac{N}{2} - 1$

- Has $N$ outputs, each a sum of $M$ terms: naively $O(NM)$ cost (flops)
- Off-grid version of fast Fourier transform (FFT) there $M = N$ and $x_j = 2\pi j/N$
What is this non-uniform fast Fourier transform? (NUFFT)

1D case, “type 1” transform:

Given locations $x_1, \ldots, x_M$, and their “strengths” $c_1, \ldots, c_M$, return

$$f_k = \sum_{j=1}^{M} e^{ikx_j}c_j, \quad \text{for } k = -\frac{N}{2}, -\frac{N}{2} + 1, \ldots, \frac{N}{2} - 2, \frac{N}{2} - 1$$

- Has $N$ outputs, each a sum of $M$ terms: naively $O(NM)$ cost (flops)
- Off-grid version of fast Fourier transform (FFT) there $M = N$ and $x_j = 2\pi j/N$
- NUFFT does this fast in $O(M + N \log N)$ nearly linear cost
- approximates it to a user-requested $\varepsilon$ $\varepsilon = 10^{-1}$ (fastest) to $\varepsilon = 10^{-14}$ (slowest)
What is this non-uniform fast Fourier transform? (NUFFT)

1D case, “type 1” transform:

Given locations $x_1, \ldots, x_M$, and their “strengths” $c_1, \ldots, c_M$, return

$$f_k = \sum_{j=1}^{M} e^{ikx_j} c_j, \quad \text{for } k = \frac{-N}{2}, \frac{-N}{2} + 1, \ldots, \frac{N}{2} - 2, \frac{N}{2} - 1$$

- Has $N$ outputs, each a sum of $M$ terms: naively $O(NM)$ cost (flops)
- Off-grid version of fast Fourier transform (FFT) there $M = N$ and $x_j = \frac{2\pi j}{N}$
- NUFFT does this fast in $O(M + N \log N)$ nearly linear cost
- approximates it to a user-requested $\varepsilon$ $\varepsilon = 10^{-1}$ (fastest) to $\varepsilon = 10^{-14}$ (slowest)

“Type 2” is its adjoint (but not inverse!) $g_j = \sum_{N} f_k e^{-ikx_j}$ $U \rightarrow NU$
- evaluates a given Fourier series at arbitrary targets $\{x_j\}$, fast
What is this non-uniform fast Fourier transform? (NUFFT)

1D case, “type 1” transform:

Given locations \( x_1, \ldots, x_M \), and their “strengths” \( c_1, \ldots, c_M \), return

\[
f_k = \sum_{j=1}^{M} e^{ikx_j} c_j, \quad \text{for } k = -\frac{N}{2}, -\frac{N}{2} + 1, \ldots, \frac{N}{2} - 2, \frac{N}{2} - 1
\]

- Has \( N \) outputs, each a sum of \( M \) terms: naively \( O(NM) \) cost (flops)
- Off-grid version of fast Fourier transform (FFT) where \( M = N \) and \( x_j = 2\pi j/N \)
- NUFFT does this fast in \( O(M + N \log N) \) nearly linear cost
- approximates it to a user-requested \( \varepsilon \) \( \varepsilon = 10^{-1} \) (fastest) to \( \varepsilon = 10^{-14} \) (slowest)

“Type 2” is its adjoint (but not inverse!) \( g_j = \sum_N f_k e^{-ikx_j} \quad U \rightarrow NU \)

- evaluates a given Fourier series at arbitrary targets \( \{x_j\} \), fast

“Type 3” as type 1 but arbitrary target freqs. \( \{s_k\}_{k=1}^{N} \): \( f_k = \sum_{j=1}^{M} e^{is_k x_j} c_j \quad NU \rightarrow NU \)
Higher dimensions also needed

2D type 1: \[ f_{k,\ell} = \sum_{j=1}^{M} e^{i(kx_j + \ell y_j)} c_j \]
for \( k, \ell \) in a rectangle of modes

3D type 1: \[ f_{k,\ell,m} = \sum_{j=1}^{M} e^{i(kx_j + \ell y_j + m z_j)} c_j \]

- dimensions \{1, 2, 3\} \times \{1, 2, 3\} = 9 transforms
  
  Software design: how to avoid code repetition?

- 9 transforms \times \{\text{float}, \text{double}\} = 18 \text{ functions}

  How reduce number of functions to write and maintain?
Who uses such transforms?

1) Fourier image reconstruction: $f_{k,\ell}$ is unknown pixel intensities
   apparatus measures strengths $c_j$ at nonuniform frequency points $(x_j, y_j)$
   - MRI (either 2D slice, or 3D)
   - coherent diffraction/powder imaging (X-ray)
   - very long baseline interferometry (VLBI)
   - cryo electron microscopy

2) Numerical forward solvers, simulation
   - electrostatics or fluid problems in periodic box
     spectral Ewald method: Poisson solve trivial in Fourier space
   - numerical PDE
     eg, interpolating between overlapping grids
     eval Fourier transform by numerical quadrature (type 1)
   - Fresnel diffraction (optics)

3) Spatial/temporal statistics
   - power-spectrum of NU time-series, or point-masses galaxies
   - fast kernel apply in Gaussian process regression
Who uses such transforms?

1) Fourier image reconstruction: \( f_{k,\ell} \) is unknown pixel intensities
   apparatus measures strengths \( c_j \) at nonuniform frequency points \( (x_j, y_j) \)
   - MRI (either 2D slice, or 3D)
   - coherent diffraction/powder imaging (X-ray)
   - very long baseline interferometry (VLBI)
   - cryo electron microscopy

2) Numerical forward solvers, simulation
   - electrostatics or fluid problems in periodic box
     spectral Ewald method: Poisson solve trivial in Fourier space
   - numerical PDE eg, interpolating between overlapping grids
   - eval Fourier transform by numerical quadrature (type 1)
   - Fresnel diffraction (optics)
Who uses such transforms?

1) Fourier image reconstruction: \( f_{k,\ell} \) is unknown pixel intensities
   apparatus measures strengths \( c_j \) at nonuniform frequency points \((x_j, y_j)\)
   • MRI (either 2D slice, or 3D)
   • coherent diffraction/powder imaging (X-ray)
   • very long baseline interferometry (VLBI)
   • cryo electron microscopy

2) Numerical forward solvers, simulation
   • electrostatics or fluid problems in periodic box
     spectral Ewald method: Poisson solve trivial in Fourier space
   • numerical PDE eg, interpolating between overlapping grids
   • eval Fourier transform by numerical quadrature (type 1)
   • Fresnel diffraction (optics)

3) Spatial/temporal statistics
   • power-spectrum of NU time-series, or point-masses galaxies
   • fast kernel apply in Gaussian process regression
Fourier imaging example: black hole by VLBI

Probably the most famous astro image of 2019: $10^{-10}$ radian resolution!

2D type 2 $\rightarrow$ uniform image grid $f_{k,\ell}$

predicted signals $\{g_j\}$ at NU pts
Fourier imaging example: black hole by VLBI

Probably the most famous astro image of 2019:

10\(^{-10}\) radian resolution!

uniform image grid \( f_{k,\ell} \)

predicted signals \( \{g_j\} \) at NU pts

\(
\text{our library (FINUFFT) used by SMILI code for this image (K. Akiyama, '19)}
\)
Fourier imaging example: black hole by VLBI

Probably the most famous astro image of 2019: $10^{-10}$ radian resolution!

2D type 2 $\rightarrow$ uniform image grid $f_{k,\ell}$ predicted signals $\{g_j\}$ at NU pts

- our library (FINUFFT) used by SMILI code for this image (K. Akiyama, '19)

How? the above is a linear forward model: $\mathbf{g} = A\mathbf{f}$ big dense matrix $A$
Fourier imaging example: black hole by VLBI

Probably the most famous astro image of 2019:  

\[10^{-10} \text{ radian resolution!}\]

2D type 2  \[\rightarrow\]\  uniform image grid \(f_{k,\ell}\)  

predicted signals \(\{g_j\}\) at NU pts

- our library (FINUFFT) used by SMILI code for this image (K. Akiyama, '19)

How? the above is a linear \textit{forward model}:  

\[ g = Af \]

Iterative optimization of \(f\) until it best fits the detected signal:

\[ f_{\text{recon}} = \arg \min_f \|A f - g_{\text{detected}}\|_2^2 + \lambda_1 \|f\|_1 + \lambda_{TV} \|f\|_{TV} \]

- each iteration, \(A\) and \(A^*\) applied fast by NUFFTs
- same idea in other 2D or 3D Fourier imaging (MRI, cryo-EM, etc)
The most common NUFFT algorithm—and ours

(Dutt–Rokhlin ’93, Steidl ’98, Greengard–Lee ’04, Potts et al, . . .

Eg, 1D Type 1: set up new grid on $[0, 2\pi)$ with $n = 2N$ points, say

Design a kernel $\psi(x)$ fun math: small width ($w$ grid-points) YET $\epsilon$-small Fourier tails

1) Spread each spike $c_j$ onto grid $b_\ell = \sum_{j=1}^{M} c_j \psi(\ell h - x_j)$

detail: $2\pi$-periodize $O(wdM)$ flops $O(\ldots)$ kernel evals

often dominates cost

2) Let $\{\hat{b}_k\} = \text{size-n FFT of } \{b_\ell\}$

3) Correct for spreading: $\tilde{f}_k = \frac{1}{\hat{\psi}(k)} \hat{b}_k$ keep only low modes $-N/2 \leq k < N/2$ • Type 2 reverses the steps; Type 3 is "Type 2 wrapped inside a Type 1"
The most common NUFFT algorithm—and ours

(Dutt–Rokhlin ’93, Steidl ’98, Greengard–Lee ’04, Potts et al, . . .)

Eg, 1D Type 1:

set up new grid on \([0, 2\pi]\) with \(n = 2N\) points, say

Design a *kernel* \(\psi(x)\)

fun math: small width (\(w\) grid-points) YET \(\varepsilon\)-small Fourier tails

\[
\text{Spread each spike } c_j \text{ onto grid } b_{\ell} = \sum_{j=1}^{M} c_j \psi(\ell - x_j)
\]

\(O(\frac{\pi}{2})\) flops \(O(\frac{\pi d M}{2})\) kernel evals often dominates cost

3) Correct for spreading:

\[
\tilde{f}_k = \hat{\psi}(k) \hat{b}_k \text{ keep only low modes } \quad -\frac{N}{2} \leq k < \frac{N}{2}
\]

\(\cdot\) Type 2 reverses the steps; Type 3 is “Type 2 wrapped inside a Type 1”
The most common NUFFT algorithm—and ours

(Dutt–Rokhlin ’93, Steidl ’98, Greengard–Lee ’04, Potts et al, . . .)

Eg, 1D Type 1:

Design a kernel $\psi(x)$

fun math: small width (w grid-points) YET $\varepsilon$-small Fourier tails

1) **Spread** each spike $c_j$ onto grid $b_\ell = \sum_{j=1}^{M} c_j \psi(\ell h - x_j)$

detail: $2\pi$-periodize

\[ O(wh/M) \] flops

\[ O(wdM) \] kernel evals

often dominates cost
The most common NUFFT algorithm—and ours

(Dutt–Rokhlin ’93, Steidl ’98, Greengard–Lee ’04, Potts et al, . . .)

Eg, 1D Type 1:

set up new grid on \([0, 2\pi]\) with \(n = 2N\) points, say

fun math: small width \((w\) grid-points) YET \(\varepsilon\)-small Fourier tails

1) **Spread** each spike \(c_j\) onto grid \(b_\ell = \sum_{j=1}^{M} c_j \psi(\ell h - x_j)\)

detail: \(2\pi\)-periodize

\(\mathcal{O}(w^d M)\) flops

\(\mathcal{O}(wdM)\) kernel evals

often dominates cost

2) Let \(\{\hat{b}_k\} = \text{size-}n\ \text{FFT}\) of \(\{b_\ell\}\)
The most common NUFFT algorithm—and ours

(Dutt–Rokhlin ’93, Steidl ’98, Greengard–Lee ’04, Potts et al, …)

Eg, 1D Type 1:

Design a kernel \( \psi(x) \)

1) \textbf{Spread} each spike \( c_j \) onto grid \( b_\ell = \sum_{j=1}^{M} c_j \psi(\ell h - x_j) \)

set up new grid on \([0, 2\pi)\) with \( n = 2N \) points, say

fun math: small width (\( w \) grid-points) YET \( \epsilon \)-small Fourier tails

1) Spread each spike \( c_j \) onto grid \( b_\ell = \sum_{j=1}^{M} c_j \psi(\ell h - x_j) \) detail: \( 2\pi \)-periodize

\[ c_1 \tilde{\psi}(x-x_1) \]

upsampled grid spacing \( h = 2\pi / n \)

kernel half–width \( wh / 2 \)

\( O(w^d M) \) flops

\( O(wdM) \) kernel evals

often dominates cost

2) Let \( \{ \hat{b}_k \} = \text{size-} n \ \text{FFT} \ \text{of} \ \{ b_\ell \} \)

3) \textbf{Correct} for spreading: \( \hat{f}_k = \frac{1}{\hat{\psi}(k)} \hat{b}_k \) keep only low modes \(-N/2 \leq k < N/2\)
The most common NUFFT algorithm—and ours

\( (\text{Dutt–Rokhlin '93, Steidl '98, Greengard–Lee '04, Potts et al, ...}) \)

Eg, 1D Type 1:

Design a kernel \( \psi(x) \)

fun math: small width \((w \text{ grid-points})\) YET \(\varepsilon\)-small Fourier tails

1) **Spread** each spike \( c_j \) onto grid \( b_\ell = \sum_{j=1}^{M} c_j \psi(\ell h - x_j) \) detail: \(2\pi\)-periodize

\[ O\left(\frac{w d M}{M} \right) \text{ flops} \]
\[ O\left(w d M \right) \text{ kernel evals} \]
\[ \text{often dominates cost} \]

2) Let \( \{\hat{b}_k\} = \text{size-}n \text{ FFT of } \{b_\ell\} \)

3) **Correct** for spreading: \( \hat{f}_k = \frac{1}{\hat{\psi}(k)} \hat{b}_k \) keep only low modes \(-N/2 \leq k < N/2\)

- Type 2 reverses the steps; Type 3 is “Type 2 wrapped inside a Type 1”
Story of our code: FINUFFT


- NUFFT needed for imaging (cryo-EM, etc)

- had: NYU single-threaded Fortran, Gaussian kernel $\psi(x)$ too wide (Greengard–Lee '09)

- also: NFFT3 C++, multithreaded, hard to use, user chooses $\psi(x)$ (Keiner et al '06)

Wanted faster multithreaded code, easy-to-use from many languages

2015: Jeremy: a C++ multithreaded spreader, nufft comparison

2016: Flatiron Institute founded

2017: I got excited, wrote FINUFFT building on J’s ideas/code. Me: C/Fort/Matlab

I write C++ like “C plus pass-by-reference”, simple, no STL, no classes, no namespacing. . .

Fix nearly-optimal $\psi(x) = e^{\beta \sqrt{1-x^2}}$, think it’s faster to eval. than others (I was wrong)

2017-2020: Fortran, MATLAB/Octave, Python, Julia wrappers; GPU code

Now: 212 GitHub stars, dozens of known users

Lesson: keep a CHANGELOG in your repo commit messages not enough
Story of our code: FINUFFT


NUFFT needed for imaging (cryo-EM, etc)

had: NYU single-threaded Fortran, Gaussian kernel $\psi(x)$ too wide (Greengard–Lee ’09)

also: NFFT3 C++, multithreaded, hard to use, user chooses $\psi(x)$ (Keiner et al ’06)

Wanted faster multithreaded code, easy-to-use from many languages

2015: Jeremy: a C++ multithreaded spreader, nufft_comparison
Story of our code: FINUFFT


NUFFT needed for imaging (cryo-EM, etc)

had: NYU single-threaded Fortran, Gaussian kernel $\psi(x)$ too wide (Greengard–Lee ’09)

also: NFFT3 C++, multithreaded, hard to use, user chooses $\psi(x)$ (Keiner et al ’06)

Wanted faster multithreaded code, easy-to-use from many languages

2015: Jeremy: a C++ multithreaded spreader, nufft_comparison

[2016: Flatiron Institute founded]

Lesson: keep a CHangelog in your repo

commit messages not enough,

“Small” (lib = 3k lines of C++)

Dive into some aspects...
Story of our code: FINUFFT


NUFFT needed for imaging (cryo-EM, etc)

had: NYU single-threaded Fortran, Gaussian kernel $\psi(x)$ too wide (Greengard–Lee ’09)

also: NFFT3 C++, multithreaded, hard to use, user chooses $\psi(x)$ (Keiner et al ’06)

Wanted faster multithreaded code, easy-to-use from many languages

2015: Jeremy: a C++ multithreaded spreader, nufft_comparison

[2016: Flatiron Institute founded]

2017: I got excited, wrote FINUFFT building on J’s ideas/code. Me: C/Fort/Matlab

I write C++ like “C plus pass-by-reference”, simple, no STL, no classes, no namespaces...

Fix nearly-optimal $\psi(x) = e^{\beta \sqrt{1-x^2}}$, think it’s faster to eval. than others (I was wrong)
Story of our code: FINUFFT


NUFFT needed for imaging (cryo-EM, etc)

had: NYU single-threaded Fortran, Gaussian kernel $\psi(x)$ too wide (Greengard–Lee '09)
also: NFFT3 C++, multithreaded, hard to use, user chooses $\psi(x)$ (Keiner et al '06)

Wanted faster multithreaded code, easy-to-use from many languages

2015: Jeremy: a C++ multithreaded spreader, nufft_comparison

[2016: Flatiron Institute founded]

2017: I got excited, wrote FINUFFT building on J’s ideas/code. Me: C/Fort/Matlab

I write C++ like “C plus pass-by-reference”, simple, no STL, no classes, no namespacing...

Fix nearly-optimal $\psi(x) = e^{\beta \sqrt{1-x^2}}$, think it’s faster to eval. than others (I was wrong)

2017-2020: Fortran, MATLAB/Octave, Python, Julia wrappers; GPU code

Lesson: keep a CHANGELOG in your repo

commit messages not enough

Small (lib = 3k lines of C++)
Dive into some aspects...
Story of our code: FINUFFT


NUFFT needed for imaging (cryo-EM, etc)

had: NYU single-threaded Fortran, Gaussian kernel $\psi(x)$ too wide (Greengard–Lee ’09)

also: NFFT3 C++, multithreaded, hard to use, user chooses $\psi(x)$ (Keiner et al ’06)

Wanted faster multithreaded code, easy-to-use from many languages

2015: Jeremy: a C++ multithreaded spreader, nufft_comparison

[2016: Flatiron Institute founded]

2017: I got excited, wrote FINUFFT building on J’s ideas/code. Me: C/Fort/Matlab

I write C++ like “C plus pass-by-reference”, simple, no STL, no classes, no namespacing. . .

Fix nearly-optimal $\psi(x) = e^{\beta \sqrt{1-x^2}}$, think it’s faster to eval. than others (I was wrong)

2017-2020: Fortran, MATLAB/Octave, Python, Julia wrappers; GPU code

now: 212 GitHub stars, dozens of known users national labs, on 6000 GPUs, etc
Story of our code: FINUFFT


NUFFT needed for imaging (cryo-EM, etc)

had: NYU single-threaded Fortran, Gaussian kernel $\psi(x)$ too wide (Greengard–Lee '09)

also: NFFT3 C++, multithreaded, hard to use, user chooses $\psi(x)$ (Keiner et al '06)

Wanted faster multithreaded code, easy-to-use from many languages

2015: Jeremy: a C++ multithreaded spreader, nufft_comparison

[2016: Flatiron Institute founded]

2017: I got excited, wrote FINUFFT building on J’s ideas/code. Me: C/Fort/Matlab

I write C++ like “C plus pass-by-reference”, simple, no STL, no classes, no namespacing. . .

Fix nearly-optimal $\psi(x) = e^{\beta \sqrt{1-x^2}}$, think it’s faster to eval. than others (I was wrong)

2017-2020: Fortran, MATLAB/Octave, Python, Julia wrappers; GPU code

now: 212 GitHub stars, dozens of known users national labs, on 6000 GPUs, etc

Lesson: keep a CHANGELOG in your repo commit messages not enough 😊

“Small” (lib = 3k lines of C++) Dive into some aspects...
Performance: kernel $\psi(x)$ evaluation

Problem: for some chips & compilers, $\exp(x)$ slow (40 M evals/sec/core)
Fix: piecewise polynomial approx + Horner’s rule

C gen. by MATLAB!

• GCC/ICC compilers SIMD-vectorize this; get 400-700 M evals/sec/core
• think hard re SIMD, but avoid maintaining intrinsics immintrin.h
• are exploring custom AVX512 (Wenda Zhou + R. Blackwell)
Performance: kernel $\psi(x)$ evaluation

Problem: for some chips & compilers, $\exp(x)$ slow (40 M evals/sec/core)
Fix: piecewise polynomial approx + Horner’s rule

C gen. by MATLAB!

• GCC/ICC compilers SIMD-vectorize this; get 400-700 M evals/sec/core
• think hard re SIMD, but avoid maintaining intrinsics immintrin.h
• are exploring custom AVX512 (Wenda Zhou + R. Blackwell)
Performance: spreading

The *order* in which NU points spread to grid has big effect on speed!

- bin-sort NU pts (into $16 \times 4 \times 4$ cuboids of grid)
- process all pts in bin 1, then bin 2, ... *good for keeping grid in cache* (J. Magland)
- multithreaded the bin-sort  

(2023: M. Reinecke speeds it up!)
Performance: spreading

The *order* in which NU points spread to grid has big effect on speed!
- bin-sort NU pts (into $16 \times 4 \times 4$ cuboids of grid)
- process all pts in bin 1, then bin 2, \ldots \text{good for keeping grid in cache (J. Magland)}
- multithreaded the bin-sort \text{(2023: M. Reinecke speeds it up!)}

Load-balanced multithreading
- Type-2 easy: OpenMP parallel over NU pts
- Type-1 not so: writes collide!

Load-balanced multithreading (also collab with J. Magland)

2D case, type−1, spread to fine grid:
- 1D kernel evals
- outer prod
- copy over
- spread

subproblems: each own thread

NU pts x j

$N_f$
Interface—how it evolved

2017: I wanted simple, familiar to users in C/Fortran and match NYU code

C-compatible: pass pointers, explicit array sizes, return value is error code...

```c
int M = 1e8; // number of nonuniform points
vector<double> x(M); // NU pts
vector<complex<double>> c(M); // NU strengths
// (here user fills x and c as they like)
int N = 1e7; // number of output modes
vector<complex<double>> f(N); // allocate output array
int status = finufft1d1(M, &x[0], &c[0], +1, 1e-9, N, &f[0], NULL); // do it
```

2018: Batch transforms sharing NU pts:

`finufft1d1many(ntrans, M, ...)`

batch FFTW often faster, sorting only done once

2020: maintaining 18 × 2 functions too much pain → "guru" interface

4-function pattern: Create plan, Set the NU pts, Execute transform(s), Destroy plan.

`finufft plan`: "opaque" pointer to (private) C++ struct. (as in Brian's talk)

Lesson: learn about public vs private headers, namespacing (as I had to)

Lesson: think hard about interface, break it VERY rarely!

help users: preserve all simple and batch interfaces (they call guru)
Interface—how it evolved

2017: I wanted simple, familiar to users in C/Fortran and match NYU code

C-compatible: pass pointers, explicit array sizes, return value is error code...

```c
int M = 1e8; // number of nonuniform points
vector<double> x(M); // NU pts
vector<complex<double> > c(M); // NU strengths
// (here user fills x and c as they like)
int N = 1e7; // number of output modes
vector<complex<double> > f(N); // allocate output array
int status = finufft1d1(M, &x[0], &c[0], +1, 1e-9, N, &f[0], NULL); // do it
```

- Why last arg NULL? it accepts ptr to options C-struct eg, opts.debug=2

2018: Batch transforms sharing NU pts: `finufft1d1many(ntrans, M, ...)`

Batch FFTW often faster, sorting only done once

2020: maintaining 18 × 2 functions too much pain → “guru” interface

4-function pattern: Create plan, Set the NU pts, Execute transform(s), Destroy plan.

`finufftplan`: “opaque” pointer to (private) C++ struct. (as in Brian’s talk)

Lesson: learn about public vs private headers, namespacing (as I had to)

Lesson: think hard about interface, break it VERY rarely! help users: preserve all simple and batch interfaces (they call guru)
Interface—how it evolved

2017: I wanted simple, familiar to users in C/Fortran and match NYU code

C-compatible: pass pointers, explicit array sizes, return value is error code...

```c
int M = 1e8; // number of nonuniform points
vector<double> x(M); // NU pts
vector<complex<double> > c(M); // NU strengths
// (here user fills x and c as they like)
int N = 1e7; // number of output modes
vector<complex<double> > f(N); // allocate output array
int status = finufft1d1(M, &x[0], &c[0], +1, 1e-9, N, &f[0], NULL); // do it
```

- Why last arg NULL? it accepts ptr to options C-struct eg, opts.debug=2

2018: Batch transforms sharing NU pts:

```c
finufft1d1many(ntrans, M, ...)
```

- batch FFTW often faster, sorting only done once
Interface—how it evolved

2017: I wanted simple, familiar to users in C/Fortran and match NYU code

C-compatible: pass pointers, explicit array sizes, return value is error code...

```c
int M = 1e8; // number of nonuniform points
vector<double> x(M); // NU pts
vector<complex<double> > c(M); // NU strengths
// (here user fills x and c as they like)
int N = 1e7; // number of output modes
vector<complex<double> > f(N); // allocate output array
int status = finufft1d1(M, &x[0], &c[0], +1, 1e-9, N, &f[0], NULL); // do it
```

- Why last arg NULL? it accepts ptr to options C-struct eg, opts.debug=2

2018: Batch transforms sharing NU pts: finufft1d1many(ntrans, M, ...)

batch FFTW often faster, sorting only done once

2020: maintaining $18 \times 2$ functions too much pain $\rightarrow$ “guru” interface

4-function pattern: Create plan, Set the NU pts, Execute transform(s), Destroy plan.

finufft_plan: “opaque” pointer to (private) C++ struct. (as in Brian’s talk)
Interface—how it evolved

2017: I wanted simple, familiar to users in C/Fortran and match NYU code

C-compatible: pass pointers, explicit array sizes, return value is error code...

```c
int M = 1e8; // number of nonuniform points
vector<double> x(M); // NU pts
vector<complex<double> > c(M); // NU strengths
// (here user fills x and c as they like)
int N = 1e7; // number of output modes
vector<complex<double> > f(N); // allocate output array
int status = finufft1d1(M, &x[0], &c[0], +1, 1e-9, N, &f[0], NULL); // do it
```

- Why last arg NULL? it accepts ptr to options C-struct eg, opts.debug=2

2018: Batch transforms sharing NU pts: `finufft1d1many(ntrans, M, ...)`

batch FFTW often faster, sorting only done once

2020: maintaining $18 \times 2$ functions too much pain → “guru” interface

4-function pattern: Create plan, Set the NU pts, Execute transform(s), Destroy plan.

`finufft_plan`: “opaque” pointer to (private) C++ struct. (as in Brian’s talk)

Lesson: learn about public vs private headers, namespaces (as I had to)

Lesson: think hard about interface, break it VERY rarely!

help users: preserve all simple and batch interfaces (they call guru)
Wrappers to other languages: expands user base

Guru interface made wrapping easier: just wrap 4 funcs, pass opaque ptr
write the simple and batched functions via a few lines in each high-level language

MATLAB/Octave

simple for users \( f = \text{finufft1d1}(x,c,+1,1e-9,N); \)
• still recommend MWrap to auto-gen MEX code horror

(Libin Lu)

(Bindel '09)
Wrappers to other languages: expands user base

Guru interface made wrapping easier: just wrap 4 funcs, pass opaque ptr
write the simple and batched functions via a few lines in each high-level language

MATLAB/Octave

simple for users f = finufft1d1(x,c,+1,1e-9,N);
• still recommend MWrap to auto-gen MEX code horror

Libin Lu

Python

2018: Dan Foreman-Mackey wrote a pybind11 wrapper to FINUFFT
• served well for a while. But eg forced recompile of libfinufft.so

2020: Joakim Andén switched us to ctypes

nice, 1000 lines incl. auto-doc-gen

Others have wrapped (cu)FINUFFT in autodiff frameworks:
tensorflow-nufft, jax-finufft, pytorch-finufft
Wrappers to other languages: expands user base

Guru interface made wrapping easier: just wrap 4 funcs, pass opaque ptr
write the simple and batched functions via a few lines in each high-level language

MATLAB/Octave

simple for users \( f = \text{finufft1d1}(x,c,+1,1e-9,N); \)
• still recommend MWrap to auto-gen MEX code horror (Libin Lu, Bindel ‘09)

Python

2018: Dan Foreman-Mackey wrote a pybind11 wrapper to FINUFFT
• served well for a while. But eg forced recompile of libfinufft.so

2020: Joakim Andén switched us to ctypes nice, 1000 lines incl. auto-doc-gen

Others have wrapped (cu)FINUFFT in autodiff frameworks:
tensorflow-nufft, jax-finufft, pytorch-finufft

Julia wrapper is separate repo: helps separate concerns (L. af Klinteberg)

Lessons: each new language brings installation troubles linux/OSX/Windows
Testing

Need tests that check accuracy for all transforms fail if measured error $\geq 10\varepsilon$?

> make test [includes bunch of edge cases M=0, N=0, eps=0.0, etc...]
0 segfaults out of 8 tests done
0 fails out of 8 tests done
Testing

Need tests that check accuracy for all transforms fail if measured error $\geq 10\varepsilon$?

> make test [includes bunch of edge cases M=0, N=0, eps=0.0, etc...]
0 segfaults out of 8 tests done
0 fails out of 8 tests done

A story: I wrote finufft1d_test, etc, writes to stdout for humans...

test 1d type 1:
1000000 NU pts to 1000000 modes in 0.0829 s 1.21e+07 NU pts/s
one mode: rel err in F[370000] is 6.59e-08
[...] Such speed and accuracy testers are crucial for progress
Testing

Need tests that check accuracy for all transforms fail if measured error $\geq 10\varepsilon$?

> make test  
[includes bunch of edge cases $M=0$, $N=0$, $\varepsilon=0.0$, etc...]

0 segfaults out of 8 tests done
0 fails out of 8 tests done

A story: I wrote finufft1d_test, etc, writes to stdout for humans...

test 1d type 1:
$1000000$ NU pts to $1000000$ modes in $0.0829$ s $1.21e+07$ NU pts/s
one mode: rel err in F[370000] is $6.59e-08$

[[...]]  
Such speed and accuracy testers are crucial for progress

To make a pass-fail test, wrote bash to pipe stdout to numdiff (linux tool)

3 years later: OSX, Windows users cannot find numdiff, ugh!

thus: each C++ test driver now uses exit code $0$ for success, $1$ for fail

Lesson: remove all nonessential dependencies. Use exit code as test result.
Testing

Need tests that check accuracy for all transforms fail if measured error $\geq 10\varepsilon$?

> make test [includes bunch of edge cases M=0, N=0, eps=0.0, etc...]

0 segfaults out of 8 tests done
0 fails out of 8 tests done

A story: I wrote finufft1d_test, etc, writes to stdout for humans...

test 1d type 1:
1000000 NU pts to 1000000 modes in 0.0829 s 1.21e+07 NU pts/s
one mode: rel err in $F[370000]$ is 6.59e-08

Such speed and accuracy testers are crucial for progress

To make a pass-fail test, wrote bash to pipe stdout to numdiff (linux tool)

3 years later: OSX, Windows users cannot find numdiff, ugh!

thus: each C++ test driver now uses exit code 0 for success, 1 for fail

Lesson: remove all nonessential dependencies. Use exit code as test result.

Lesson: make tests (and examples) for each wrapped language
Testing

Need tests that check accuracy for all transforms fail if measured error ≥ 10ε?

> make test [includes bunch of edge cases M=0, N=0, eps=0.0, etc...]
0 segfaults out of 8 tests done
0 fails out of 8 tests done

A story: I wrote finufft1d_test, etc, writes to stdout for humans...

test 1d type 1:
1000000 NU pts to 1000000 modes in 0.0829 s 1.21e+07 NU pts/s
one mode: rel err in F[370000] is 6.59e-08

Such speed and accuracy testers are crucial for progress

To make a pass-fail test, wrote bash to pipe stdout to numdiff (linux tool)

3 years later: OSX, Windows users cannot find numdiff, ugh!

thus: each C++ test driver now uses exit code 0 for success, 1 for fail

Lesson: remove all nonessential dependencies. Use exit code as test result.

Lesson: make tests (and examples) for each wrapped language

Continuous Integration (CI)

Jenkins (was using makefile, now uses CMake)
**Documentation**

README.md, or a .tex file to PDF, fine for small project, but:

---

**Web-facing essential for professionalism**

- math definitions
- installation
- examples
- tutorial cases
- troubleshooting
- devnotes

...
Documentation: lessons

- Decent solution I recommend: sphinx, host at readthedocs.io
  docs/* .rst ReStructuredText, in master branch (unlike GHpages)
  \LaTeX\ math pretty good via mathjax, texext
Documentation: lessons

- Decent solution I recommend: sphinx, host at readthedocs.io
docs/* .rst ReStructuredText, in master branch (unlike GHpages)
\LaTeX\ math pretty good via mathjax, texext

- SSOT: avoid repetition, makes maintenance/changes easier
  Eg: C++ source has no doc, refers user to docs/ (ie web)
  But don’t get carried away with obscure doc build automation
  eg: my bash script auto-generates C/C++ docs, now hard to understand
Documentation: lessons

- Decent solution I recommend: sphinx, host at readthedocs.io
docs/*.rst ReStructuredText, in master branch (unlike GHpages)
\LaTeX\ math pretty good via mathjax, texext

- SSOT: avoid repetition, makes maintenance/changes easier
  Eg: C++ source has no doc, refers user to docs/ (ie web)
But don’t get carried away with obscure doc build automation
  eg: my bash script auto-generates C/C++ docs, now hard to understand

- Describe what your tool does in plain mathematical language
  Don’t assume jargon from one science area (“wavefunction”, “k-space” etc)
  Most MRI NUFFT codes fail to do this ⇒ unusable
Documentation: lessons

- Decent solution I recommend: sphinx, host at readthedocs.io
docs/*.rst ReStructuredText, in master branch (unlike GHpages)
\LaTeX\ math pretty good via mathjax, texext

- SSOT: avoid repetition, makes maintenance/changes easier
  Eg: C++ source has no doc, refers user to docs/ (ie web)

  But don’t get carried away with obscure doc build automation
  eg: my bash script auto-generates C/C++ docs, now hard to understand

- Describe \textit{what your tool does in plain mathematical language}
  Don’t assume jargon from one science area (“wavefunction”, “k-space” etc)
  Most MRI NUFFT codes fail to do this \Rightarrow unusable

- Always update docs \textit{at the time you change the source}
**GPU version**

**cuFINUFFT**: CUDA called from C++ *(Melody Shih, intern ’18, ’19 [NYU→NVidia])*  

- spreading done in shared memory (subproblems), not global memory *(Shih et al, PDSEC 2021, best paper prize)*

2023: cleaned up GPU lib (+tests,…) into FINUFFT repo *(R. Blackwell)*

Future: language interfaces auto-detect GPU arrays?
Conclusions: developing a small(ish) library

First identify a need
– can you perform task faster / more accurate?

+ benchmark existing codes, on same task!

– users find bugs you’d never thought of
  must respond (thanks Libin)

– users make improvement PRs, or beat your speed
  respond to some

– be conservative: one change at a time
  for your sanity (eg: no MPI)

– try to be somewhat idiomatic
  I won’t use #define FLT double
  in C++ again

Collect a great team: software engineers + interns + motivated users
– library becomes used,
  ⇒ more Issues / requests

– quarterly(?) all-team meeting
  prioritize features, drop others

Installation, dependencies & language wrappers takes much of our time!
– helps users, but needs a team (linux, OSX, Windows). Your choice.
Eg: guess which language wrapper SMILI black hole astro code uses?
– still plenty of fun with math & floating-point speed hacking!

Thank-you!
Conclusions: developing a small(ish) library

First identify a need
  – can you perform task faster / more accurate?

Start small, but professionalizing a library involves learning!
  – users find bugs you’d never thought of
  – users make improvement PRs, or beat your speed
  – be conservative: one change at a time
  – try to be somewhat idiomatic

+ benchmark existing codes, on same task!
  must respond (thanks Libin)
  respond to some
  for your sanity (eg: no MPI)

I won’t use #define FLT double in C++ again 😊
Conclusions: developing a small(ish) library

First identify a need + benchmark existing codes, on same task!
- can you perform task faster / more accurate?

Start small, but professionalizing a library involves learning!
- users find bugs you’d never thought of must respond (thanks Libin)
- users make improvement PRs, or beat your speed respond to some
- be conservative: one change at a time for your sanity (eg: no MPI)
- try to be somewhat idiomatic I won’t use #define FLT double in C++ again 😊

Collect a great team: software engineers + interns + motivated users
- library becomes used 😊 ⇒ more Issues / requests 😞
- quarterly(?) all-team meeting prioritize features, drop others
Conclusions: developing a small(ish) library

First identify a need + benchmark existing codes, on same task!
  – can you perform task faster / more accurate?

Start small, but professionalizing a library involves learning!
  – users find bugs you’d never thought of must respond (thanks Libin)
  – users make improvement PRs, or beat your speed respond to some
  – be conservative: one change at a time for your sanity (eg: no MPI)
  – try to be somewhat idiomatic I won’t use \#define FLT double in C++ again 😊

Collect a great team: software engineers + interns + motivated users
  – library becomes used 😊 ⇒ more Issues / requests 😞 prioritize features, drop others
  – quarterly(?) all-team meeting

Installation, dependencies & language wrappers takes much of our time!
  – helps users, but needs a team (linux, OSX, Windows). Your choice.

Eg: guess which language wrapper SMILI black hole astro code uses?
  – still plenty of fun with math & floating-point speed hacking!

Thank-you!