

## Nonuniform FFTs at Flatiron

—  $\sim$  \* *OR* \*  $\sim$  —

### Lessons from developing a small numerical library

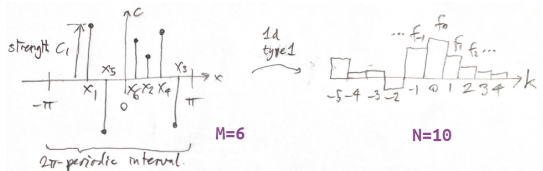
**Alex Barnett**<sup>1</sup>, with much help from: Jeremy Magland<sup>1</sup>, Ludvig af Klinteberg (Mälardalen U.), Melody Shih (NVidia), Joakim Andén (KTH), Libin Lu<sup>1</sup>, Robert Blackwell (SCC), Andrea Malleo (Bloomberg), and many others...

FWAM5    Friday, October 20, 2023

<sup>1</sup>Center for Computational Mathematics, Flatiron Institute

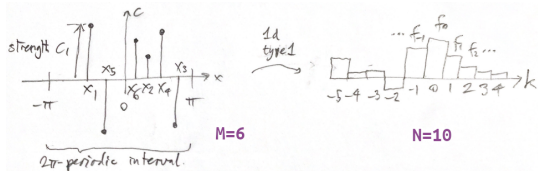
# What is this non-uniform fast Fourier transform? (NUFFT)

1D case, “type 1” transform:



# What is this non-uniform fast Fourier transform? (NUFFT)

1D case, “type 1” transform:



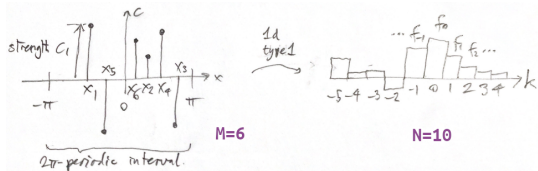
Given locations  $x_1, \dots, x_M$ , and their “strengths”  $c_1, \dots, c_M$ , return

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

- Has  $N$  outputs, each a sum of  $M$  terms: naively  $\mathcal{O}(NM)$  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:



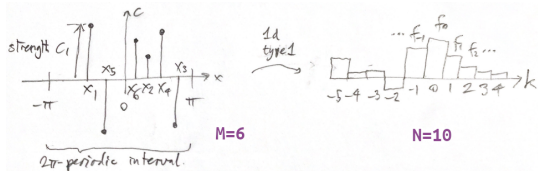
Given locations  $x_1, \dots, x_M$ , and their “strengths”  $c_1, \dots, c_M$ , return

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

- Has  $N$  outputs, each a sum of  $M$  terms: naively  $\mathcal{O}(NM)$  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  $\mathcal{O}(M + N \log N)$  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  $x_1, \dots, x_M$ , and their “strengths”  $c_1, \dots, c_M$ , return

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

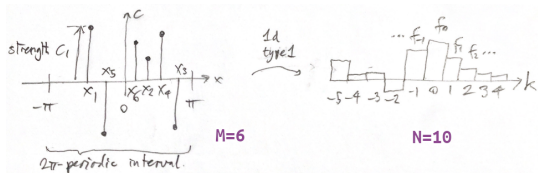
- Has  $N$  outputs, each a sum of  $M$  terms: naively  $\mathcal{O}(NM)$  cost (flops)
- Off-grid version of fast Fourier transform (FFT) there  $M = N$  and  $x_j = 2\pi j/N$
- NUFFT does this *fast* in  $\mathcal{O}(M + N \log N)$  nearly linear cost
- approximates it to a user-requested  $\varepsilon$   $\varepsilon = 10^{-1}$  (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  $x_1, \dots, x_M$ , and their “strengths”  $c_1, \dots, c_M$ , return

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

- Has  $N$  outputs, each a sum of  $M$  terms: naively  $\mathcal{O}(NM)$  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  $\mathcal{O}(M + N \log N)$  nearly linear cost
- approximates it to a user-requested  $\varepsilon$   $\varepsilon = 10^{-1}$  (fastest) to  $\varepsilon = 10^{-14}$  (slowest)

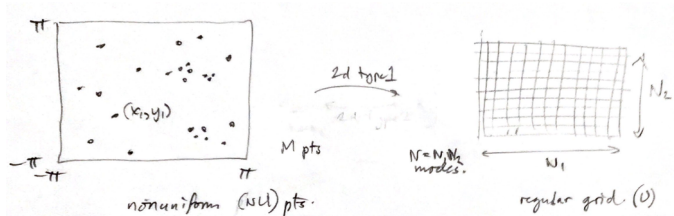
“Type 2” is its adjoint (but not inverse!)  $g_j = \sum_N f_k e^{-ikx_j}$  U  $\rightarrow$  NU

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

“Type 3” as type 1 but arbitrary target freqs.  $\{s_k\}_{k=1}^N$ :  $f_k = \sum_{j=1}^M e^{is_k x_j} c_j$   
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:  $f_{k,\ell,m} = \sum_{j=1}^M e^{i(kx_j + \ell y_j + mz_j)} c_j$   
etc

- dimensions  $\{1, 2, 3\} \times$  types  $\{1, 2, 3\} = 9$  transforms

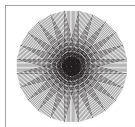
Software design: how to avoid code repetition?

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



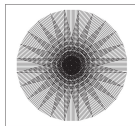
some 2D MRI points



## 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



some 2D MRI points

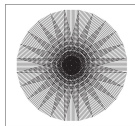
2) Numerical forward solvers, simulation

- 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



some 2D MRI points

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)

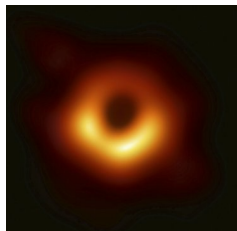
3) Spatial/temporal statistics

- power-spectrum of NU time-series, or point-masses galaxies
- fast kernel apply in Gaussian process regression

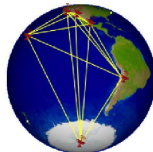
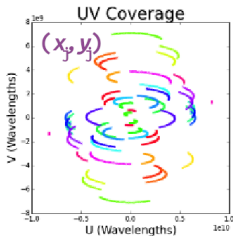
# Fourier imaging example: black hole by VLBI

Probably the most famous astro image of 2019:

$10^{-10}$  radian resolution!



2D type 2  
→



Telescope Locations

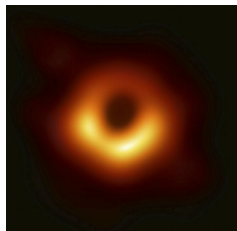
uniform image grid  $f_{k,\ell}$

predicted signals  $\{g_j\}$  at NU pts

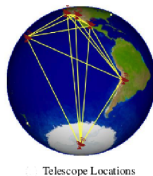
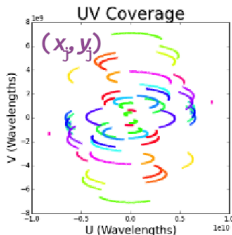
# Fourier imaging example: black hole by VLBI

Probably the most famous astro image of 2019:

$10^{-10}$  radian resolution!



2D type 2  
→



Telescope Locations

uniform image grid  $f_{k,\ell}$

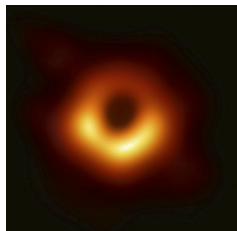
predicted signals  $\{g_j\}$  at NU pts

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

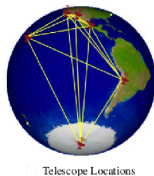
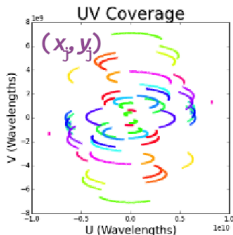
# Fourier imaging example: black hole by VLBI

Probably the most famous astro image of 2019:

$10^{-10}$  radian resolution!



2D type 2  
→



uniform image grid  $f_{k,\ell}$

predicted signals  $\{g_j\}$  at NU pts

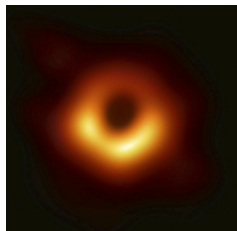
- our library (FINUFFT) used by SMILI code for this image (K. Akiyama, '19)

How? the above is a linear *forward model*:  $\mathbf{g} = \mathbf{A}\mathbf{f}$  big dense matrix  $\mathbf{A}$

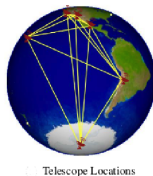
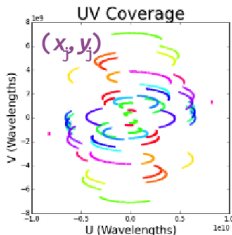
# Fourier imaging example: black hole by VLBI

Probably the most famous astro image of 2019:

$10^{-10}$  radian resolution!



2D type 2  
→



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*:  $\mathbf{g} = \mathbf{A}\mathbf{f}$  big dense matrix  $\mathbf{A}$

Iterative optimization of  $\mathbf{f}$  until it best fits the detected signal:

$$\mathbf{f}_{\text{recon}} = \arg \min_{\mathbf{f}} \|\mathbf{A}\mathbf{f} - \mathbf{g}_{\text{detected}}\|_2^2 + \lambda_1 \|\mathbf{f}\|_1 + \lambda_{\text{TV}} \|\mathbf{f}\|_{\text{TV}}$$

- each iteration,  $\mathbf{A}$  and  $\mathbf{A}^*$  applied fast by NUFFTs dominant cost, I think
- 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

# 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



# The most common NUFFT algorithm—and ours

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

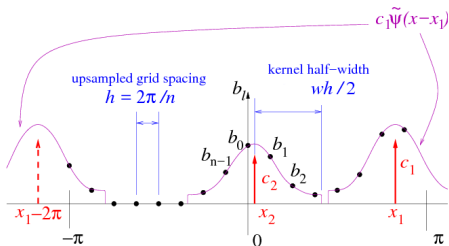
Eg, 1D Type 1:

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

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

# The most common NUFFT algorithm—and ours

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

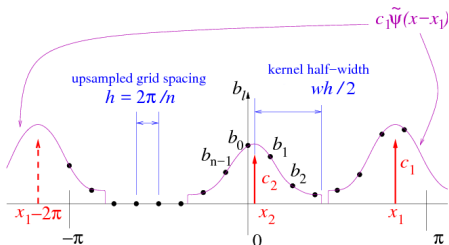
Eg, 1D Type 1:

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

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  $\{\hat{b}_k\} = \text{size-}n \text{ FFT of } \{b_l\}$

# The most common NUFFT algorithm—and ours

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

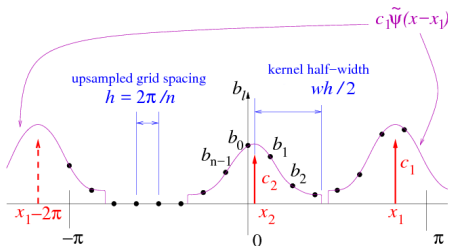
Eg, 1D Type 1:

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

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  $\{\hat{b}_k\}$  = size- $n$  **FFT** of  $\{b_l\}$

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

# The most common NUFFT algorithm—and ours

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

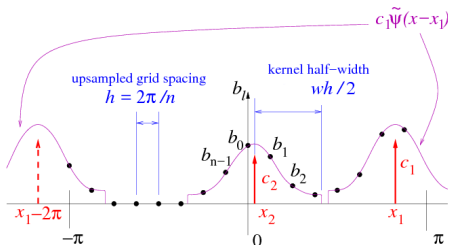
Eg, 1D Type 1:

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



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

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

often dominates cost

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

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”

# Story of our code: FINUFFT



Prehistory (2015): Leslie Greengard, Jeremy Magland,  
Marina Spivak, myself, at SCDA

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

# Story of our code: FINUFFT



Prehistory (2015): Leslie Greengard, Jeremy Magland,  
Marina Spivak, myself, at SCDA

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



Prehistory (2015): Leslie Greengard, Jeremy Magland,  
Marina Spivak, myself, at SCDA

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]

# Story of our code: FINUFFT



Prehistory (2015): Leslie Greengard, Jeremy Magland,  
Marina Spivak, myself, at SCDA

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)



# Story of our code: FINUFFT



Prehistory (2015): Leslie Greengard, Jeremy Magland,  
Marina Spivak, myself, at SCDA

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

# Story of our code: FINUFFT



Prehistory (2015): Leslie Greengard, Jeremy Magland,  
Marina Spivak, myself, at SCDA

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



Prehistory (2015): Leslie Greengard, Jeremy Magland,  
Marina Spivak, myself, at SCDA

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!

# 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!

```
// Code generated by gen_all_horner_C_code.m in finufft/devel
// Authors: Alex Barnett & Ludvig af Klinteberg.
// (C) The Simons Foundation, Inc.
if (w==2) {
    FLT c0[] = {4.514704324215315E+01, 4.514704324215300E+01, 0.000000000000000E+00, 0.000000000000000E+00};
    FLT c1[] = {5.7408070938221300E+01, -5.7408070938221293E+01, 0.000000000000000E+00, 0.000000000000000E+00};
    FLT c2[] = {-1.839511792004648E+00, -1.839511792004656E+00, 0.000000000000000E+00, 0.000000000000000E+00};
    FLT c3[] = {-2.038242625318208E+01, 2.038242625318208E+01, 0.000000000000000E+00, 0.000000000000000E+00};
    FLT c4[] = {-2.094080443357742E+00, -2.094080443357738E+00, 0.000000000000000E+00, 0.000000000000000E+00};
    for (int i=0; i<4; i++) ker[i] = c0[i] + z*(c1[i] + z*(c2[i] + z*(c3[i] + z*(c4[i]))));
} else if (w==3) {
    FLT c0[] = {1.565399118931511E+02, 8.8006872410780295E+02, 1.5653991189967152E+02, 0.000000000000000E+00};
    FLT c1[] = {3.1653018869611077E+02, 7.4325702843759617E-14, -3.1653018868907071E+02, 0.000000000000000E+00};
    FLT c2[] = {1.774269279045448E+02, -3.3149255274727801E+02, 1.7742692791117119E+02, 0.000000000000000E+00};
    FLT c3[] = {-1.5357716116473156E+01, 9.5071486252033243E-15, 1.5357716122720193E+01, 0.000000000000000E+00};
    FLT c4[] = {-3.7757583061523668E+01, 5.3222970968867315E+01, -3.7757583054647384E+01, 0.000000000000000E+00};
    FLT c5[] = {-3.965401107608804E+00, 1.8062124448285358E-13, 3.9654011139270540E+00, 0.000000000000000E+00};
    for (int i=0; i<4; i++) ker[i] = c0[i] + z*(c1[i] + z*(c2[i] + z*(c3[i] + z*(c4[i] + z*(c5[i])))));
} else if (w==4) {
    FLT c0[] = {5.4284366850213200E+02, 1.0073871433088398E+04, 1.0073871433088396E+04, 5.4284366850213223E+02};
    FLT c1[] = {1.4650917259256939E+03, 6.1905285583602863E+03, -6.1905285583602881E+03, -1.4650917259256937E+03};
    FLT c2[] = {1.4186910680718345E+03, -1.3995339862725591E+03, -1.3995339862725598E+03, 1.4186910680718347E+03};
    FLT c3[] = {5.1133995502497419E+02, -1.4191608683682996E+03, 1.4191608683682998E+03, -5.1133995502497424E+02};
    FLT c4[] = {-4.8293622641174039E+01, 3.9393732546135226E+01, 3.9393732546135816E+01, -4.8293622641174061E+01};
    FLT c5[] = {-7.8386867802392288E+01, 1.4918904800408930E+02, -1.4918904800408751E+02, 7.8386867802392359E+01};
    FLT c6[] = {-1.0039212571700894E+01, 5.0626747735616746E+00, 5.062674773562512E+00, -1.0039212571700640E+01};
    for (int i=0; i<4; i++) ker[i] = c0[i] + z*(c1[i] + z*(c2[i] + z*(c3[i] + z*(c4[i] + z*(c5[i] + z*(c6[i]))))));
} else if (w==5) {
    FLT c0[] = {9.9223677575398392E+02, 3.7794697666613320E+04, 9.8715771010760494E+04, 3.7794697666613283E+04, 9.9223677575398403E+02, 0.000000000000000E+00, 0.000000000000000E+00, 0.000000000000000E+00};
    for (int i=0; i<4; i++) ker[i] = c0[i] + z*(c1[i] + z*(c2[i] + z*(c3[i] + z*(c4[i] + z*(c5[i] + z*(c6[i] + z*(c7[i]))))));
}
}
ker_horner_allw_loop.c Top L1 Git-master (C/*L WS Abbrev)
```

- 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, ... *good for keeping grid in cache* (J. Magland)
- multithreaded the bin-sort (2023: M. Reinecke speeds it up!)

## Load-balanced multithreading

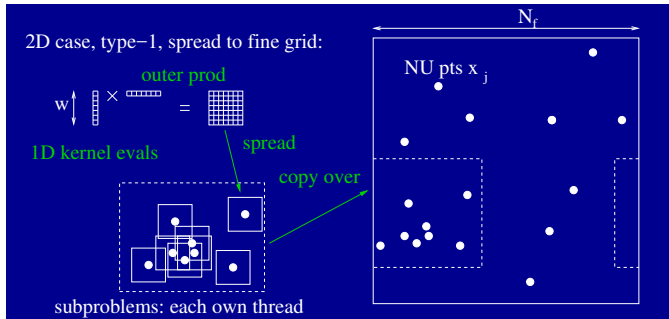
(also collab with J. Magland)

Type-2 easy: OpenMP parallel over NU pts

*no collisions reading from U blocks*

Type-1 not so: writes collide!

*load-balance via “subproblems” each of  $10^4$  NU pts*



## 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...

```
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
```



## 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...

```
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`

## 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...

```
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

## 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...

```
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)

## 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...

```
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)

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)

## 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

(Libin Lu)

simple for users `f = finufft1d1(x,c,+1,1e-9,N);`

- still recommend MWrap to auto-gen MEX code horror

(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

(Libin Lu)

simple for users `f = finufft1d1(x,c,+1,1e-9,N);`

- still recommend MWrap to auto-gen MEX code horror

(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

## 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

(Libin Lu)

simple for users `f = finufft1d1(x,c,+1,1e-9,N);`

- still recommend MWrap to auto-gen MEX code horror

(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  $\geq 10\epsilon$  ?

```
> 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  $\geq 10\epsilon$  ?

```
> 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  $\geq 10\epsilon$  ?

```
> 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.

# Testing

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

```
> 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  $\geq 10\epsilon$  ?

```
> 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)

each GitHub push reruns tests:




Jenkins (was using `makefile`, now uses `CMake`)

# Documentation

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

finufft 2.2.0.dev0 documentation » Flatiron Institute Nonuniform Fast Fourier Transform

next



## Flatiron Institute Nonuniform Fast Fourier Transform

Table of Contents

- Installation
- Installation (GPU)
- Directories in this package
- Mathematical definitions of transforms
- Example usage from C++ and C
- Documentation of all C++ functions
- C interface (GPU)
- Options parameters
- Error (status) codes
- Troubleshooting
- Tutorials and application demos
- Usage from Fortran
- MATLAB/Octave interfaces
- Python interface
- Python interface (GPU)
- Julia interfaces
- Changelog
- Developer notes
- Related packages
- Dependent packages, wrappers, users, and citations
- Acknowledgments
- References

Next topic

Installation

This Page

Show Source

Quick search

Go

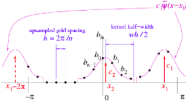



Diagram illustrating the nonuniform grid spacing  $h$  (in  $2\pi/h$ ) and the kernel half-width  $\pi h/2$ . The diagram shows a grid of points  $x_1, x_2, \dots, x_M$  and the corresponding weights  $c_1, c_2, \dots, c_M$ . The kernel half-width is indicated by a dashed line.


FINUFFT is a 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, either on a multi-core shared-memory machine, or on a GPU. It is extremely fast (typically achieving  $10^6$  to  $10^8$  points per second on a CPU, or up to  $10^9$  points per second on a GPU), 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. The CPU library is written in C++ (with limited use of ++ features), OpenMP, and calls FFTW. It has been developed since 2017 at the Center for Computational Mathematics at the Flatiron Institute, by Alex Barnett and others, and is released under an Apache v2 license.

### What does FINUFFT do?

As an example, given  $M$  real numbers  $x_j \in [0, 2\pi)$ , and complex numbers  $c_j$ , with  $j = 1, \dots, M$ , 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

$$f_k = \sum_{j=1}^M c_j e^{ikx_j}, \quad \text{for } k \in \mathbb{Z}, \quad -N/2 \leq k \leq N/2 - 1. \quad (1)$$

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) 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) is: the returned values  $f_k$  are the Fourier series coefficients of the  [v-latest](#) 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, in-

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)  
 $\text{\LaTeX}$  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)  
 $\text{\LaTeX}$  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)  
 $\text{\LaTeX}$  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  $\Rightarrow$  unusable



# Documentation: lessons

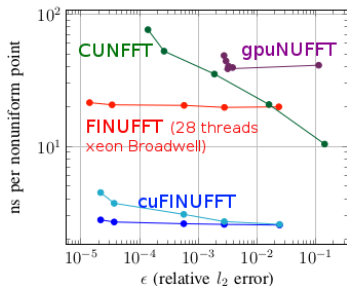
- Decent solution I recommend: sphinx, host at `readthedocs.io`  
`docs/*.rst` ReStructuredText, in master branch (unlike GHpages)  
 $\text{\LaTeX}$  math pretty good via `mathjax`, `texext`
- SSOT: avoid repetition, makes maintenance/changes easier  
Eg: C++ source has no doc, refers user to docs/ (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  $\Rightarrow$  unusable
- Always update docs *at the time you change the source*

## GPU version

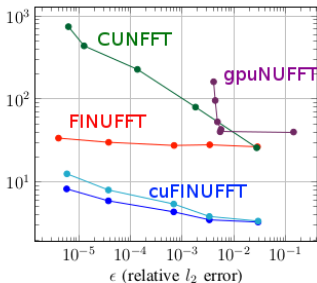
cuFINUFFT: CUDA called from C++ (Melody Shih, intern '18, '19 [NYU→NVIDIA])

Single-precision V100 GPU comparisons (including H2D & precomp, smaller is better):

2D Type 1,  $N_i = 1000$ ,  $M = 10^7$



3D Type 1,  $N_i = 100$ ,  $M = 10^7$



- 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

+ benchmark existing codes, on same task!

- can you perform task faster / more accurate?

## 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 ☺

# 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 ☹
- 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!