Fast NUFFT and SIMD

A journey from better algorithms to SIMD

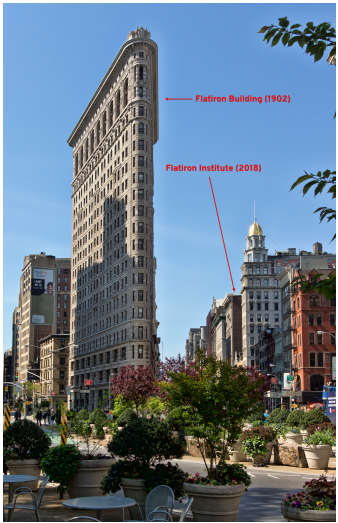
Marco Barbone
Flatiron Institute

November 5, 2025

Acknowledgments

Many of the slides are courtesy of Alex Barnett. *[Original Talk]*

Flatiron Institute and CCM



Source: Wikimedia Commons

Center for Computational Math (CCM) is one of five centers (Each $\approx$ 50 researchers)

- others: biology
- astrophysics/astronomy
- quantum physics
- neuroscience

CCM:

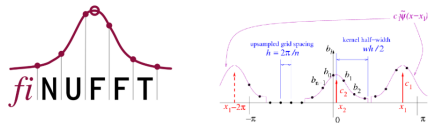
- scientific computing
- numerical analysis
- biophysics / imaging
- computational statistics
- machine learning

Supports both basic research
& software development / support

finufft 2.4.1 documentation » Flatiron Institute Nonuniform Fast Fourier Transform

Flatiron Institute Nonuniform Fast Fourier Transform

Table of Contents



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++, OpenMP, and calls FFTW. It has been developed since 2017 at the Center for Computational Mathematics at the Flatiron Institute, a division of the Simons Foundation, by Alex Barnett and others, and is released under an Apache v2 license. Here is a project overview from 2023.

Source: github.com/flatironinstitute/finufft

Recap the discrete Fourier transform (DFT)

FLATIRON INSTITUTE

Task: $f_k \;=\; \sum_{j=0}^{N-1} e^{ik\frac{2\pi}{N}j}\, c_j, \qquad -\frac{N}{2} \le k < \frac{N}{2}$ FFT is $\mathcal{O}(N \log N)$

Uses:

1. Discrete convolution (filtering, PDE solve, regular-sampled data)
2. Given uniform (U) samples of a smooth 2π-periodic function f, estimate its Fourier *series* coefficients $\hat{f}_n$; i.e.
$$f(x) \;=\; \sum_{n\in\mathbb{Z}} \hat{f}_n\, e^{-inx}$$
3. Estimate power spectrum of non-periodic signal on U grid

Contrast: what does NUFFT compute?

FLATIRON INSTITUTE

Inputs: $\{x_j\}_{j=1}^M$ NU (non-uniform) points in $[0, 2\pi)$
$\{c_j\}_{j=1}^M$ complex strengths, N = number of modes

Three flavors of task:

- **Type-1:** NU $\to$ U $\quad f_k = \sum_{j=1}^{M} e^{ikx_j} c_j, \quad -\frac{N}{2} \le k < \frac{N}{2}$

  Fourier series coeffs of sum of point masses. Generalizes the DFT ($x_j = 2\pi j/N$).

- **Type-2:** U $\to$ NU $\quad c_j = \sum_{-N/2 \le k < N/2} e^{ikx_j} f_k, \quad j = 1, \ldots, M$

  Evaluate Fourier series at arbitrary targets. Adjoint (not inverse!) of type-1.

- **Type-3:** NU $\to$ NU $\quad f_k = \sum_{j=1}^{M} e^{is_k x_j} c_j, \quad k = 1, \ldots, N$

  General exponential sum.

- For $d = 2, 3, \ldots$, replace kx_j by $\mathrm{k}\cdot\mathrm{x}_j = k_1 x_j + k_2 y_j$, etc.

Impact of the FINUFFT Library

FLATIRON INSTITUTE

Black hole imaging: The SMILI code used 2D FINUFFT (Fortran wrapper) to reconstruct the famous M87 image (EHT, MIT).
Medical imaging: Lecoeur et al. use GPU FINUFFT for 17× faster 4D-MRI. Neurospin (Paris) notes: "finufft and cufinufft are the most stable implementations we can find out there."
HPC plasma physics: GPU FINUFFT runs on 6144 Nvidia A100s at NERSC-9 Perlmutter (S. Muralikrishnan).
Cryo-EM: FINUFFT enables fast alignment/processing of 2D/3D volumes, e.g. ASPIRE and EM-Align.
Exoplanets: Used in power-spectrum estimation of nonequispaced time series: NWelch (Python) and Multitaper.jl (Julia).
Optics: Harness et al. use 2D FINUFFT for Fresnel diffraction, $10^4\times$ faster than pre-2021 methods (paper).
X-ray FEL: Multi-GPU FINUFFT accelerates single-particle reconstruction at ExaFEL (LBNL) by 5–12× (see IPDSPW paper).
Crystallography: DISCUS (manual p. 161) shows 3D type-1 FINUFFT (Fortran) gives 100× faster X-ray diffraction modeling.
Machine learning: Differentiable wrappers exist for TensorFlow, JAX, and PyTorch.

What it took to capture a black hole

FLATIRON INSTITUTE

The light from superheated plasma bends around the supermassive black hole in the galaxy M87.
Credit: Event Horizon Telescope collaboration and Andrew Grant, Physics Today

Type 1 NUFFT

For $x_j \in [0, 2\pi)$ and weights c_j, compute

$$f_k = \sum_{j=1}^{M} c_j \, e^{ikx_j}, \qquad -\frac{N}{2} \le k < \frac{N}{2}.$$

Type 1 NUFFT

For $x_j \in [0, 2\pi)$ and weights c_j, compute

$$f_k = \sum_{j=1}^{M} c_j \, e^{ikx_j}, \qquad -\frac{N}{2} \le k < \frac{N}{2}.$$

Type 2 is similar, type 3 uses type 2 inside

A super-crude 1D type-1 NUFFT

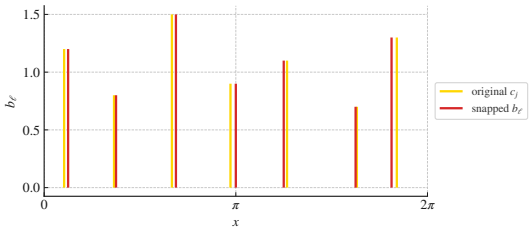
Set up fine grid on $[0, 2\pi)$, *spacing* $h = \frac{2\pi}{N_f}$, $N_f > N$

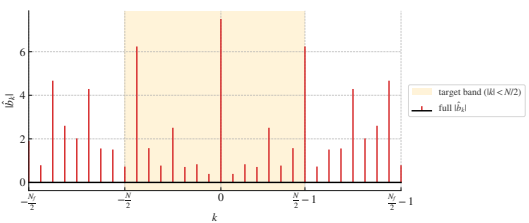
Three steps:

(a) Add each strength c_j onto the fine-grid cell with location $\tilde{x}_j$ nearest x_j.

(b) Call this vector $\{b_\ell\}_{\ell=0}^{N_f-1}$; take its size-$N_f$ FFT to get $\{\hat{b}_k\}_{k=-N_f/2}^{N_f/2-1}$.

(c) Keep just low-freq outputs: $\tilde{f}_k = \hat{b}_k$, for $-N/2 \le k < N/2$.





Error analysis of the "snap-to-grid" method

FLATIRON INSTITUTE

What is error? High freqs $|k| = N/2$ are the worst: relative error thus

$$e^{i\frac{N}{2}x_j} - e^{i\frac{N}{2}\tilde{x}_j} = \mathcal{O}(Nh) = \mathcal{O}\left(\frac{N}{N_f}\right).$$

- 1st-order convergent: e.g. error 10^{-1} needs $N_f \approx 10N$.
- in 3D needs $N_f^3 \approx 10^3 N^3$ — $1000\times$ slower than plain FFT, for 1-digit accuracy! Terrible!

And yet the idea of dumping onto a fine grid is actually good...
But need much more rapid convergence!

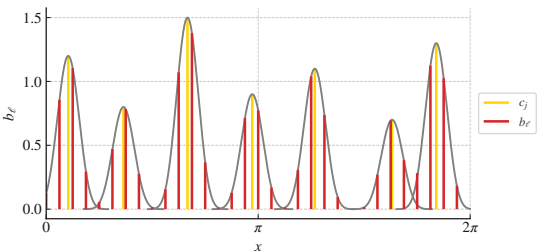
Kernel-based type-1 NUFFT (one slide)

Algorithm (with oversampling $N_f = \sigma N$):

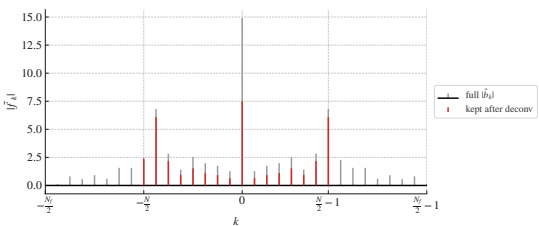
1) *Spread:* $b_\ell = \sum_j c_j\, \psi(\ell h - x_j)$ to $\approx w$ nearby fine-grid nodes.

2) *FFT:* $\widehat{b}_k = \mathcal{F}_{N_f}\{b_\ell\}$, $-N_f/2 \le k < N_f/2$.

3) *Deconvolve & truncate:* $\tilde{f}_k = \widehat{b}_k / R(k)$ for $|k| < N/2$.



spreading



deconv. & truncation

Kernel design

Design criteria:

- *Narrow support* in x (width $w\,h$) for fast spreading.
- *Tiny FT tails* beyond max band $|k| < \frac{N}{2}$ (esp. near $2\pi(1 - \frac{1}{2\sigma})$).
- smooth, and numerically stable.

Kaiser Bessel kernel:

```
def kaiser_bessel(dx, w, h, beta=None):
    if abs(dx) > (w*h)/2: return 0.0
    a = 2*dx/(w*h)
    if beta is None: beta = 2.34*w # tune
    return float(np.i0(beta*np.sqrt(1-a*a)) / np.i0(beta))
```

Notes: Choose w (e.g., 6–12) to balance accuracy vs. cost; β values are tuning knobs.

Efficient kernel evaluation in FINUFFT

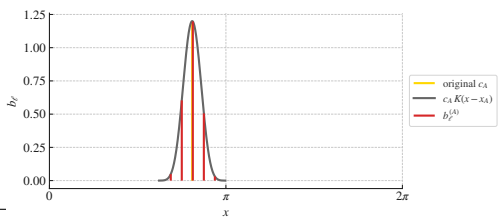
FLATIRON INSTITUTE

- In FINUFFT the cost of kernel evaluation is no longer important.
- The kernel is approximated using a piecewise polynomial
- Evaluation is done with vectorized Horner's rule.
- This makes kernel evaluation both accurate and extremely fast.
- The true bottleneck is spreading and interpolation, not the kernel itself.
- However, support still matters!
- Practical rule (FINUFFT, typically $\sigma = 2$): $w \approx 7 \Rightarrow \varepsilon \sim 10^{-6}$, $w \approx 13 \Rightarrow \varepsilon \sim 10^{-12}$.

(Barnett et al., 2020, arXiv:2001.09405)

1D spreading

```
1  T kernel_values[MAX_NSPREAD];
2  for (BIGINT i = 0; i < M; ++i) { // loop over all points
3    const T re0 = dd[2 * i];  // pts are complex
4    const T im0 = dd[2 * i + 1];
5    BIGINT i1 = (BIGINT)std::ceil(kx[i] - ns2);
6    T x1      = T(i1) - kx[i];
7    ker_eval<T>(kernel_values, ns, opts, x1);
8    BIGINT j = i1 - off1;
9    for (int dx = 0; dx < int(ns); ++dx) { // critical loop
10     const T k = kernel_values[dx];
11     du[2 * j]     += re0 * k;
12     du[2 * j + 1] += im0 * k;
13     ++j;
14   }
15 }
```



Performance issues

1. *NS* is runtime, impacts compiler vectorization
2. auto vectorization is unreliable

Performance issues

1. *NS* is runtime, impacts compiler vectorization
2. auto vectorization is unreliable
3. $NS \in [2, 16] \rightarrow$ kernel generation is possible!

Compile-time integer ranges

FLATIRON INSTITUTE

```
// helper to generate the integer sequence in range [Start, End]
template<int Offset, typename Seq> struct offset_seq;

template<int Offset, int... I>
struct offset_seq<Offset, std::integer_sequence<int, I...>> {
  using type = std::integer_sequence<int, (Offset + I)...>;
};

template<int Start, int End>
using make_range =
    typename offset_seq<Start,
      std::make_integer_sequence<int, End - Start + 1>>::type;
```

- ‘make_range<2,4>‘ $\rightarrow$ ‘integer_sequence<int,2,3,4>‘.
- Lets us encode integer intervals at compile-time.

DispatchParam

```
template<typename Seq>
struct DispatchParam {
  int runtime_val;        // value known only at runtime
  using seq_type = Seq;   // admissible values (compile-time)
};
```

- Couples a runtime integer with its allowed range of template parameters.
- Example:

  ```
  DispatchParam<make_range<2,5>> dim{runtime_dim};
  ```

Cartesian product over sequences

FLATIRON INSTITUTE

```
template<typename F, typename... Seq> struct Product;

// Recursive case
template<typename F, int... I1, typename Seq2, typename... Rest>
struct Product<F, std::integer_sequence<int, I1...>, Seq2, Rest...> {
  template<int... Prefix> static void apply(F &f) {
    (Product<F, Seq2, Rest...>::template apply<Prefix..., I1>(f), ...);
  }
};

// Base case
template<typename F, int... I1>
struct Product<F, std::integer_sequence<int, I1...>> {
  template<int... Prefix> static void apply(F &f) {
    (f.template operator()<Prefix..., I1>(), ...);
  }
};
```

- Expands all possible combinations of integer template parameters.
- Calls ‘f.template operator()<...>()‘ for each.

DispatcherCaller

FLATIRON INSTITUTE

```
template<typename Func, std::size_t N,
         typename ArgTuple, typename ResultType>
struct DispatcherCaller {
  Func &func;
  const std::array<int, N> &vals;
  ArgTuple &args;
  std::conditional_t<std::is_void_v<ResultType>, char, ResultType> result{};
  template<int... Params> void operator()() {
    static constexpr std::array<int, sizeof...(Params)> p{Params...};
    if (p == vals) { // match runtime ints
      if constexpr (std::is_void_v<ResultType>)
        std::apply([&](auto&&... a){
          func.template operator()<Params...>(std::forward<decltype(a)>(a)...);
        }, args);
      else
        result = std::apply([&](auto&&... a){
          return func.template operator()<Params...>(std::forward<decltype(a)>(a)...);
        }, args);
    }
  }
};
```

dispatch() function

FLATIRON INSTITUTE

```
template<typename Func, typename ParamTuple, typename... Args>
decltype(auto) dispatch(Func &&func, ParamTuple &&params, Args &&...args) {
  constexpr std::size_t N = std::tuple_size_v<std::remove_reference_t<ParamTuple>>;
  auto vals     = detail::extract_vals(params);   // runtime ints
  auto seqs     = detail::extract_seqs(params);   // compile-time ranges
  auto argtuple = std::forward_as_tuple(std::forward<Args>(args)...);

  using result_t = detail::dispatch_result_t<Func,
                     decltype(argtuple), decltype(seqs)>;

  detail::DispatcherCaller<Func, N, decltype(argtuple), result_t> caller{
      func, vals, argtuple};

  std::apply([&](auto &&...s){ detail::product(caller, s...); }, seqs);

  if constexpr (!std::is_void_v<result_t>)
    return caller.result;
}
```

- 'dispatch' takes runtime integers + argument pack.
- Matches against compile-time ranges.
- Calls functor with correct template parameters.

Some more boilerplate

FLATIRON INSTITUTE

```
template<typename T> struct SpreadSubproblem1dCaller {
  BIGINT off1;
  UBIGINT size1;
  T *du;
  UBIGINT M;
  const T *kx;
  const T *dd;
  const finufft_spread_opts &opts;
  template<int NS, int KM> int operator()() const {
    if (opts.simd == 1)
      spread_subproblem_1d_scalar_kernel<T, NS, KM>(off1, size1, du, M, kx, dd, opts);
    else
      spread_subproblem_1d_kernel<T, NS, KM>(off1, size1, du, M, kx, dd, opts);
    return 0;
  }
};
```

NS is now compile time

FLATIRON INSTITUTE

```
SpreadSubproblem1dCaller<T> caller{off1, size1, du, M, kx, dd, opts};
using NsSeq  = make_range<MIN_NSPREAD, MAX_NSPREAD>;
using KerSeq = std::make_integer_sequence<int, KEREVAL_METHODS>;
auto params  = std::make_tuple(DispatchParam<NsSeq>{opts.nspread},
                               DispatchParam<KerSeq>{opts.kerevalmeth});
dispatch(caller, params);
```

NS is now compile time

```
SpreadSubproblem1dCaller<T> caller{off1, size1, du, M, kx, dd, opts};
using NsSeq  = make_range<MIN_NSPREAD, MAX_NSPREAD>;
using KerSeq = std::make_integer_sequence<int, KEREVAL_METHODS>;
auto params  = std::make_tuple(DispatchParam<NsSeq>{opts.nspread},
                               DispatchParam<KerSeq>{opts.kerevalmeth});
dispatch(caller, params);
```

C++ metaprogramming is a lot of pain... BUT, now there is a nice API to allow compile time dispatch of runtime parameters!

NS is now compile time

FLATIRON INSTITUTE

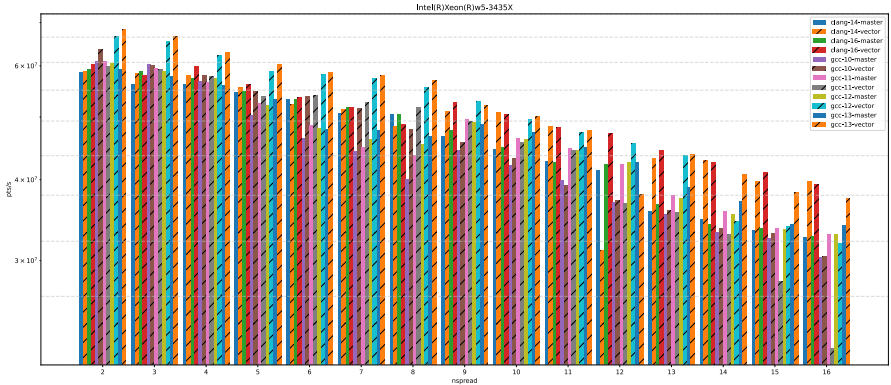
```
SpreadSubproblem1dCaller<T> caller{off1, size1, du, M, kx, dd, opts};
using NsSeq  = make_range<MIN_NSPREAD, MAX_NSPREAD>;
using KerSeq = std::make_integer_sequence<int, KEREVAL_METHODS>;
auto params  = std::make_tuple(DispatchParam<NsSeq>{opts.nspread},
                               DispatchParam<KerSeq>{opts.kerevalmeth});
dispatch(caller, params);
```

C++ metaprogramming is a lot of pain...

BUT, now there is a nice (enough) API to allow compile time dispatch of runtime parameters!

Is it worth it?

Performance



Main idea

The idea is to vectorize the “spreading” aka “critical loop”

1D spreading

FLATIRON INSTITUTE

```
1  T kernel_values[MAX_NSPREAD];
2  for (BIGINT i = 0; i < M; ++i) { // loop over all points
3    const T re0 = dd[2 * i];  // pts are complex
4    const T im0 = dd[2 * i + 1];
5    BIGINT i1 = (BIGINT)std::ceil(kx[i] - ns2);
6    T x1      = T(i1) - kx[i];
7    ker_eval<ns, kerevalmeth, T>(kernel_values, opts, x1);
8    BIGINT j = i1 - off1;
9    for (int dx = 0; dx < int(ns); ++dx) { // critical loop
10     const T k = kernel_values[dx];
11     du[2 * j]     += re0 * k;
12     du[2 * j + 1] += im0 * k;
13     ++j;
14   }
15 }
```

Main idea

FLATIRON INSTITUTE

The idea is to vectorize the "spreading" aka "critical loop"

```
1 for (int dx = 0; dx < int(ns); ++dx) { // critical loop
2     const T k = kernel_values[dx];
3     du[2 * j]     += re0 * k;
4     du[2 * j + 1] += im0 * k;
5     ++j;
6 }
```

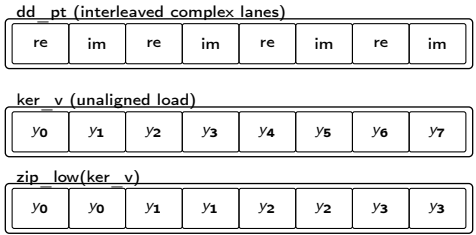
Each element of the kernel (k) is used twice: One for the real part one for the imaginary

SIMD spreading

Assuming that trg and ker are padded accordingly

```
const auto dd_pt = complex_register<simd_type>(dd[i*2], dd[i*2+1]);
for (int dx = 0; dx < int(ns); dx+=simd) { // critical loop
    auto ker_v = simd_type::load_unaligned(ker.data()+ns/2)
    ker_v = xsimd::swizzle(ker_v, zip_low_index<arch_t,T>);
    const auto du_pt = simd_type::load_unaligned(trg + dx);
    const auto res = xsimd::fma(ker_v, dd_pt, du_pt0);
    res.store_unaligned(trg + dx);
}
```

SIMD spreading



FMA and store:

```
1  const auto res = xsimd::fma(ker_v, dd_pt, du_pt0);
2  res.store_unaligned(trg + dx);
```

SIMD spreading

```
const auto dd_pt = complex_register<simd_type>(dd[i*2], dd[i*2+1]);
for (int dx = 0; dx < int(ns); dx+=simd) { // critical loop
    auto ker_v = simd_type::load_unaligned(ker.data()+ns/2)
    ker_v = xsimd::swizzle(ker_v, zip_low_index<arch_t,T>);
    const auto du_pt = simd_type::load_unaligned(trg + dx);
    const auto res = xsimd::fma(ker_v, dd_pt, du_pt0);
    res.store_unaligned(trg + dx);
}
```

The problem is that this code is memory bound!
2 loads + 1 store + one scalar load VS 1 swizzle + 1 FMA
It is not too bad because the kernel is <= 16 elements and stored on the stack. It is hot memory, likely in L1.

SIMD spreading

FLATIRON INSTITUTE

Can we do better?

```
1  const auto dd_pt = complex_register<simd_type>(dd[i*2], dd[i*2+1]);
2  for (int dx = 0; dx < int(ns); dx+=simd) { // critical loop
3      auto ker_v = simd_type::load_unaligned(ker.data()+ns/2)
4      ker_v = xsimd::swizzle(ker_v, zip_low_index<arch_t,T>);
5      const auto du_pt = simd_type::load_unaligned(trg + dx);
6      const auto res = xsimd::fma(ker_v, dd_pt, du_pt0);
7      res.store_unaligned(trg + dx);
8  }
```

SIMD spreading

FLATIRON INSTITUTE

We are loading the kernel and throwing away half of it.

```
const auto dd_pt = complex_register<simd_type>(dd[i*2], dd[i*2+1]);
for (int dx = 0; dx < int(ns); dx+=simd) { // critical loop
    auto ker_v = simd_type::load_unaligned(ker.data()+ns/2)
    ker_v = xsimd::swizzle(ker_v, zip_low_index<arch_t,T>);
    const auto du_pt = simd_type::load_unaligned(trg + dx);
    const auto res = xsimd::fma(ker_v, dd_pt, du_pt0);
    res.store_unaligned(trg + dx);
}
```

SIMD spreading

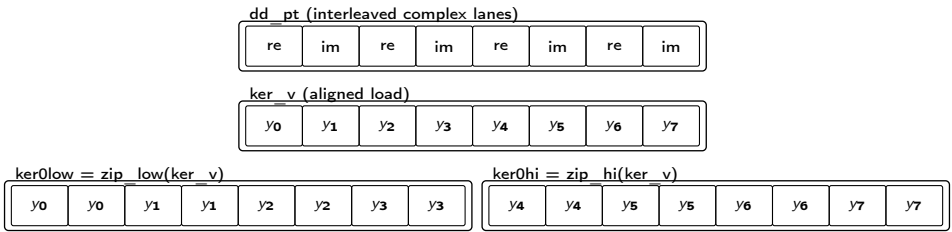
FLATIRON INSTITUTE

```
1  static constexpr auto regular_part = (2*ns + padding) &
   ↪  (-(2*simd_size));
2  // two iterations at once
3  for (uint8_t dx = 0; dx < regular_part; dx += 2*simd_size) {
4    const auto ker_v  = simd_type::load_aligned(ker.data() + dx/2);
5    const auto du_pt0 = simd_type::load_unaligned(trg + dx);
6    const auto du_pt1 = simd_type::load_unaligned(trg + dx + simd_size);
7    const auto ker0low = xsimd::swizzle(ker_v, zip_low_index<arch_t,T>);
8    const auto ker0hi  = xsimd::swizzle(ker_v, zip_hi_index<arch_t,T>);
9    const auto res0 = xsimd::fma(ker0low, dd_pt, du_pt0);
10   const auto res1 = xsimd::fma(ker0hi , dd_pt, du_pt1);
11   res0.store_unaligned(trg + dx);
12   res1.store_unaligned(trg + dx + simd_size);
13 }
```

SIMD spreading

- One `ker_v` load feeds two FMAs via `zip_low`/`zip_hi`.
- Stride by $2 \times$ `simd_size` lanes per iteration.

SIMD spreading




```
1  const auto res0 = xsimd::fma(ker0low, dd_pt, du_pt0);
2  const auto res1 = xsimd::fma(ker0hi , dd_pt, du_pt1);
3  res0.store_unaligned(trg + dx);
4  res1.store_unaligned(trg + dx + simd_size);
```

SIMD spreading

The tail needs to be handled separately, otherwise we would need to double the kernel padding

```
1 if constexpr (regular_part < 2 * ns) {
2   const auto ker0    = simd_type::load_unaligned(ker.data() +
    ↪  (regular_part / 2));
3   const auto du_pt   = simd_type::load_unaligned(trg + regular_part);
4   const auto ker0low = xsimd::swizzle(ker0, zip_low_index<arch_t, T>);
5   const auto res     = xsimd::fma(ker0low, dd_pt, du_pt);
6   res.store_unaligned(trg + regular_part);
7 }
```

SIMD spreading

The tail needs to be handled separately, otherwise we would need to double the kernel padding

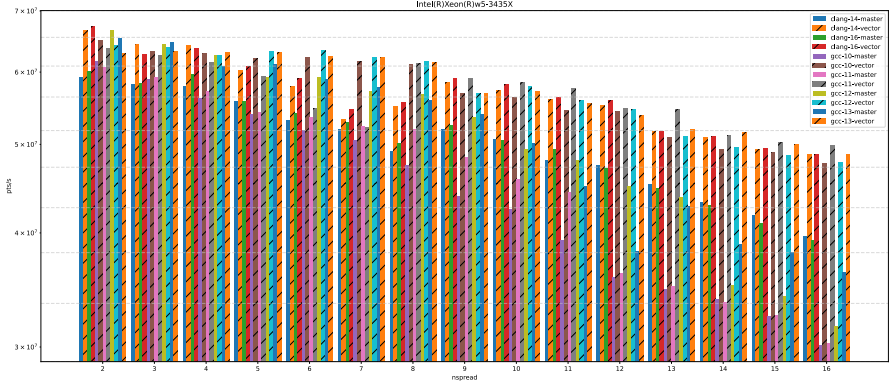
```
if constexpr (regular_part < 2 * ns) {
  const auto ker0    = simd_type::load_unaligned(ker.data() + (regular_part / 2));
  const auto du_pt   = simd_type::load_unaligned(trg + regular_part);
  const auto ker0low = xsimd::swizzle(ker0, zip_low_index<arch_t, T>);
  const auto res     = xsimd::fma(ker0low, dd_pt, du_pt);
  res.store_unaligned(trg + regular_part);
}
```

No ker0hi! We save 1 swizzle and 1 FMA

c++ constexpr allows this case to happen only when the kernel size is not a multiple of simd size

Performance

Was it all worth it?



Conclusions

- In 1D speeding is 10-100% faster than auto-vectorization
- Performance across compilers (and CPUs) is consistent
- This results in a 30-100% performance improvement for entire transforms: link
- Code is still (sort of) maintainable

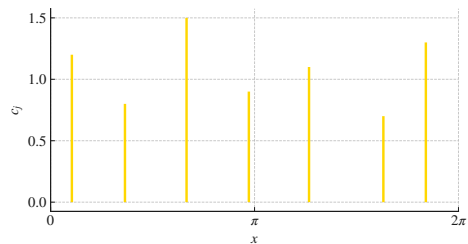
In the future: performance improvements due to better swizzle; runtime kernel fitting,

BACKUP

"snap-to-grid" algorithm Step 1 — Setup

Choose a fine grid: $N_f > N$, $h = 2\pi/N_f$.

```
import numpy as np
N = 16; sigma = 2
Nf = sigma * N
h = 2*np.pi / Nf
xj = np.array
    ([0.30,1.05,1.92,2.80,3.65,4.70,5.30])
xj = (xj/xj.max()) * (2*np.pi*0.92)
cj = np.array
    ([1.2,0.8,1.5,0.9,1.1,0.7,1.3])
```

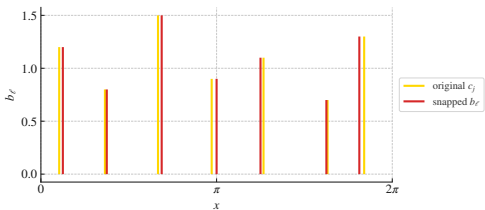


"snap-to-grid" algorithm Step 2 — Snap to nearest

Nearest-neighbor spreading:

$$b_\ell \;=\; \sum_{j=1}^{M} c_j \; 1\Big(\ell = \text{round}\frac{x_j}{h}\Big), \quad \ell = 0, \ldots, N_f - 1.$$

```
idx = np.round(xj/h).astype(int) % Nf
b = np.zeros(Nf)
for i,c in zip(idx, cj):
    b[i] += c
```

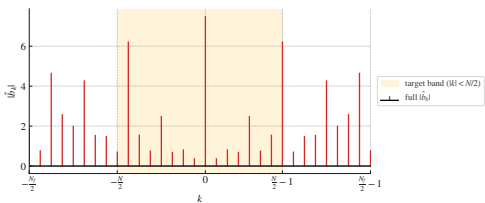


“snap-to-grid” algorithm Step 3 — FFT on fine g

Compute

$$\widehat{b}_k = \mathcal{F}_{N_f}\{b_\ell\}, \qquad -\frac{N_f}{2} \le k < \frac{N_f}{2}.$$

```
hat_b = np.fft.fftshift(np.fft.fft(b))
ks = np.arange(-Nf//2, Nf//2)
```

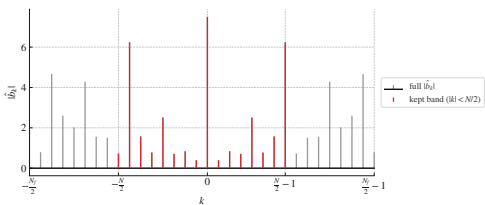


“snap-to-grid” algorithm Step 4 — Truncate to target band

Keep only $-N/2 \le k < N/2$:

$$\tilde{f}_k \;=\; \widehat{b}_k, \qquad -\frac{N}{2} \le k < \frac{N}{2}.$$

```
kmin, kmax = -N//2, N//2 - 1
mask = (ks >= kmin) & (ks <= kmax)
hat_b_trunc = np.zeros_like(hat_b)
hat_b_trunc[mask] = hat_b[mask]
```



A super-crude 1D type-1 NUFFT: "snap to fine g[illegible]"

FLATIRON INSTITUTE

Set up fine grid on $[0, 2\pi)$, *spacing* $h = \frac{2\pi}{N_f}$, $N_f > N$

Three steps:

(a) Add each strength c_j onto the fine-grid cell with location $\tilde{x}_j$ nearest x_j.

(b) Call this vector $\{b_\ell\}_{\ell=0}^{N_f-1}$; take its size-$N_f$ FFT to get $\{\hat{b}_k\}_{k=-N_f/2}^{N_f/2-1}$.

(c) Keep just low-freq outputs: $\tilde{f}_k = \hat{b}_k, \quad$ for $-N/2 \le k < N/2$.

What is error? High freqs $|k| = N/2$ are the worst: relative error thus

$$e^{i\frac{N}{2}x_j} - e^{i\frac{N}{2}\tilde{x}_j} = \mathcal{O}(Nh) = \mathcal{O}\left(\frac{N}{N_f}\right).$$

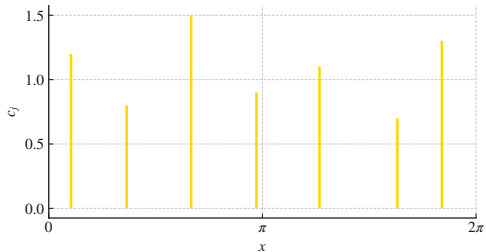
- 1st-order convergent: e.g. error 10^{-1} needs $N_f \approx 10N$.
- in 3D needs $N_f^3 \approx 10^3 N^3$ — $1000\times$ slower than plain FFT, for 1-digit accuracy! Terrible!

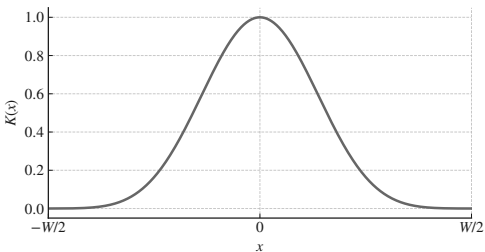
And yet the idea of dumping onto a fine grid is actually good...
But need much more rapid convergence!

Kernel Step 1 – Oversample & choose kernel

Oversample by $\sigma \approx 2$: $N_f = \sigma N$, $h = 2\pi/N_f$. Choose smooth compact ψ with width $\approx w\,h$ and $\psi(0) = 1$.

```
def kaiser_bessel(dx, w, h, beta=None):
    W = w*h; a = 2*dx/W
    if abs(a) > 1: return 0.0
    if beta is None: beta = 2.34*w
    return float(np.i0(beta*np.sqrt(1-a*a
        ))/np.i0(beta))
w = 6
```



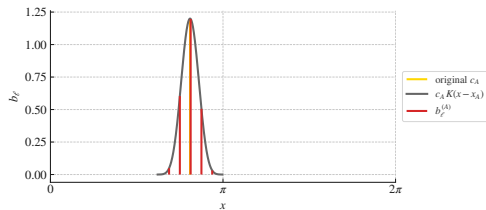


Kernel Step 2a — Single-point spreading

Spread x_A to $\approx w$ nearest grid cells:

$$b_\ell^{(A)} \;=\; c_A\,\psi(\ell h - x_A).$$

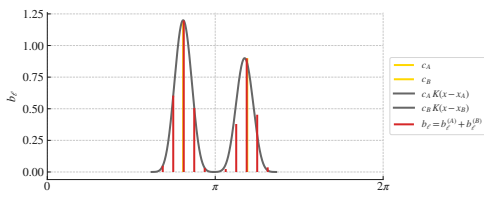
```
xA, cA = np.pi - 0.6, 1.2
bA = np.zeros(Nf)
l0 = int(np.floor((xA - (w*h)/2)/h))
l1 = int(np.ceil ((xA + (w*h)/2)/h))
for l in range(l0, l1+1):
    bA[l % Nf] += cA * kaiser_bessel(l*h
        - xA, w, h)
```



Kernel Step 2a — Single-point spreading

Spread x_B to $\approx w$ nearest grid cells:

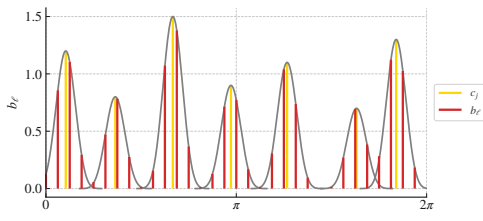
```
xB, cB = np.pi + 0.6, 0.9
bB = np.zeros(Nf)
l0 = int(np.floor((xB - (w*h)/2)/h))
l1 = int(np.ceil ((xB + (w*h)/2)/h))
for l in range(l0, l1+1):
    bB[l % Nf] += cB * kaiser_bessel(l*h
        - xB, w, h)
```



Kernel Step 2b — Superposition

Linear sum. $b_\ell = b_\ell^{(A)} + b_\ell^{(B)}$.

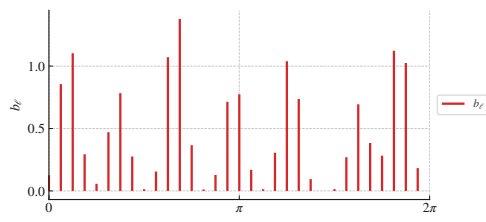
```
bSum = bA + bB # elementwise sum at grid
     nodes
```



Kernel Step 2c — Full spreading over all points

All points. In practice we loop over all M points and accumulate.

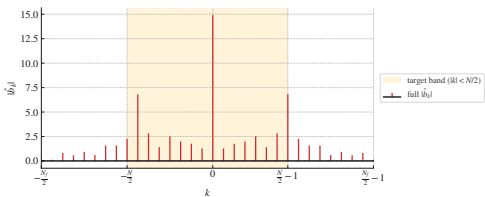
```
b2 = np.zeros(Nf)
for x, c in zip(xj, cj):
    left, right = x - (w*h)/2, x + (w*h)
        /2
    l0, l1 = int(np.floor(left/h)), int(
        np.ceil(right/h))
    for l in range(l0, l1+1):
        center = l*h
        b2[l % Nf] += c * kaiser_bessel(
            center - x, w, h)
```



Kernel Step 3 – FFT on oversampled grid

$$\widehat{b}_k = \mathcal{F}_{N_f}\{b_\ell\}, \qquad -\frac{N_f}{2} \le k < \frac{N_f}{2}.$$

```
hat_b2 = np.fft.fftshift(np.fft.fft(b2))
ks = np.arange(-Nf//2, Nf//2)
```

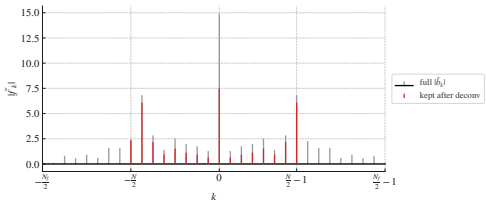


Kernel Step 4 – Deconvolution & truncation

Convolution in $x \Rightarrow$ multiplication in k.
Estimate discrete kernel response $R(k)$ by FFT of a spread unit impulse, then

$$\tilde{f}_k = \frac{\widehat{b}_k}{R(k)} \quad \text{for } -\tfrac{N}{2} \le k < \tfrac{N}{2}.$$

```
g = np.zeros(Nf)
for l in range(-w//2-1, w//2+2):
    g[l % Nf] = kaiser_bessel(l*h, w, h)
R = np.fft.fftshift(np.fft.fft(g))
kmin, kmax = -N//2, N//2 - 1
mask = (ks >= kmin) & (ks <= kmax)
hat_b2_corr = np.zeros_like(hat_b2,
    dtype=complex)
hat_b2_corr[mask] = hat_b2[mask] / (np.
    abs(R[mask]) + 1e-12)
```



FINUFFT ES ("exp of sqrt") kernel

FLATIRON INSTITUTE

$$\psi_{\rm ES}(x) \;:=\; e^{\beta\sqrt{1-z^2}}, \qquad z := \frac{2x}{wh} \in [-1,1], \quad \text{zero otherwise.}$$

(found via numerical tinkering; simplifies the Bessel I_0 route)

- Its Fourier transform $\hat{\psi}_{\rm ES}$ has *no known closed form*.
- Numerical consequence (FINUFFT): evaluate $\hat{\psi}_{\rm ES}(\xi)$ by accurate 1D quadrature and in step (c) deconvolve via division by $\hat{\psi}_{\rm ES}(\xi)$.
- Analytic consequence: error must be bounded by working with the FT integral directly.

Error analysis (upsampling $\sigma > 1$; width $w \to \infty$):

$$\varepsilon \;=\; \mathcal{O}\left(\sqrt{w}\, e^{-\pi w\sqrt{1-1/\sigma}}\right).$$

- Same exponential rate as Kaiser–Bessel and PSWF (Fuchs '64).
- Practical rule (FINUFFT, typically $\sigma = 2$): $w \approx 7 \Rightarrow \varepsilon \sim 10^{-6}$, $w \approx 13 \Rightarrow \varepsilon \sim 10^{-12}$.

(Barnett et al., 2020, arXiv:2001.09405)