Implementing and Testing the

Interpolated Factored Green Function Method

For the Accelerated Evaluation of Potentials in Electromagnetic Simulations

Michael P. Acquaviva

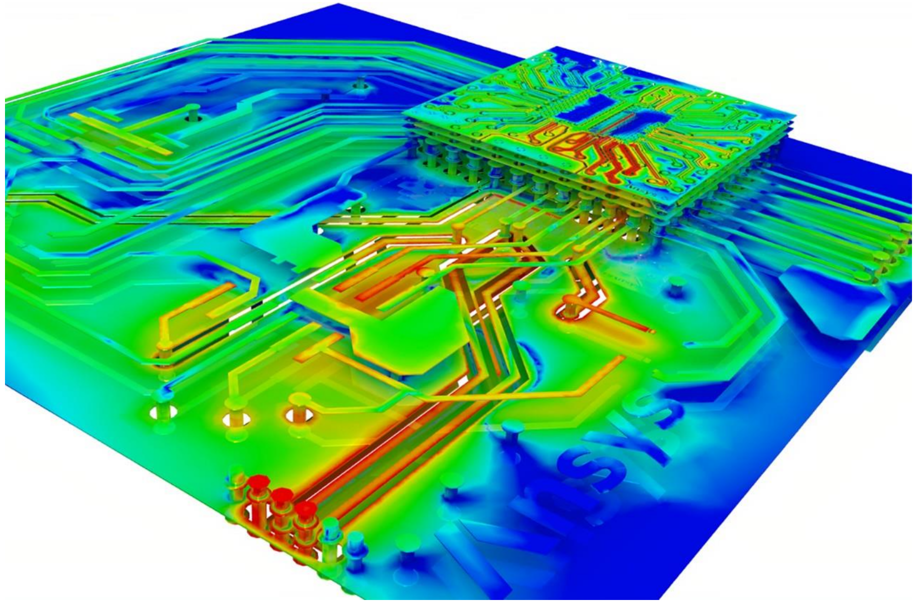
Supervised by: Piero Triverio

Engineering Science
UNIVERSITY OF TORONTO

Electromagnetic Simulation

Electromagnetic simulation is **critical** to the development of modern electronics.

As circuits become more complex, simulation and CAD tools must **evolve** to handle **increasing** electromagnetic detail and scale.



A 3D-IC interposer current density simulation
Credit: ANSYS

From Maxwell to Matrix

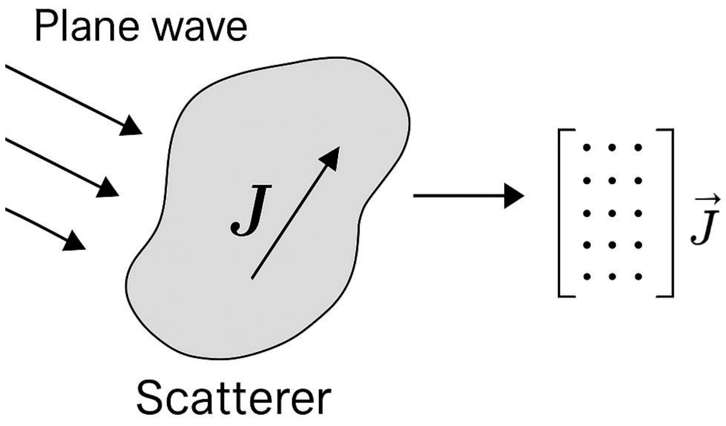
Scattering problems can be reformulated as integral equations from Maxwell's equations.

Using discrete integral formulations, the field at a **discrete** observation point is given by the convolution of the Green's function and the induced current density $\vec{J}$.

For all points, this leads to a dense linear system:

$$\boldsymbol{A}\vec{J} = \vec{b}$$

where $\boldsymbol{A}$ is the Green's function matrix and $\vec{b}$ is the known incident field.



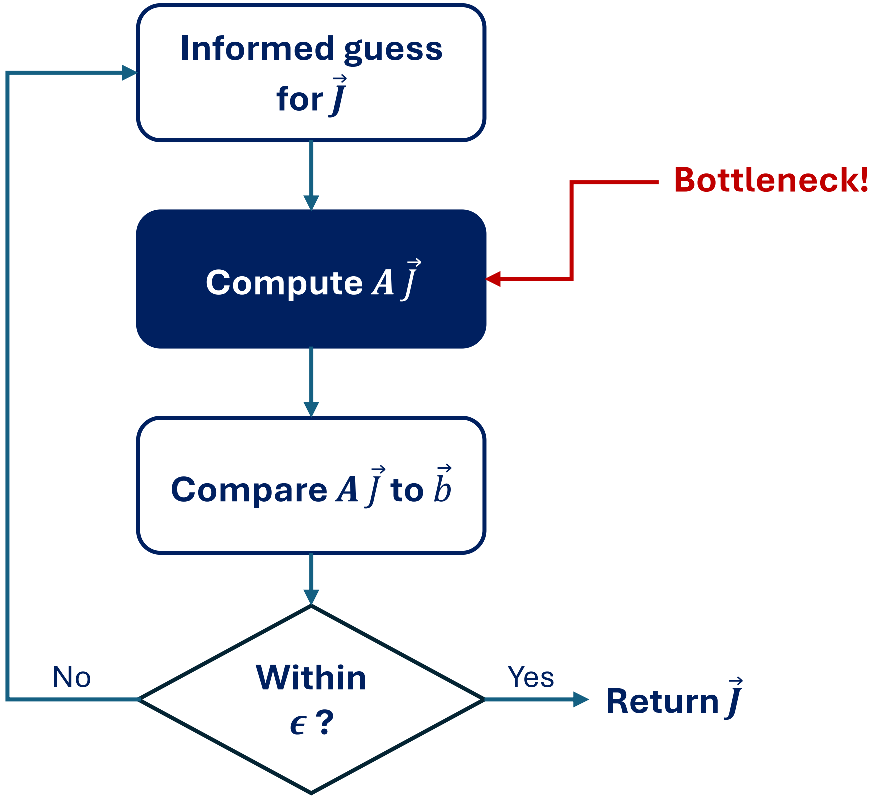
Solving the Linear System

$$\boldsymbol{A}\,\vec{J} = \vec{b}$$

Direct solutions (e.g., LU decomposition) are too expensive for large numbers of points, N.

- $\mathcal{O}(N^2)$ memory
- $\mathcal{O}(N^3)$ runtime

Instead, we use **iterative solvers** (e.g., GMRES) which only require the computation of the matrix-vector product.



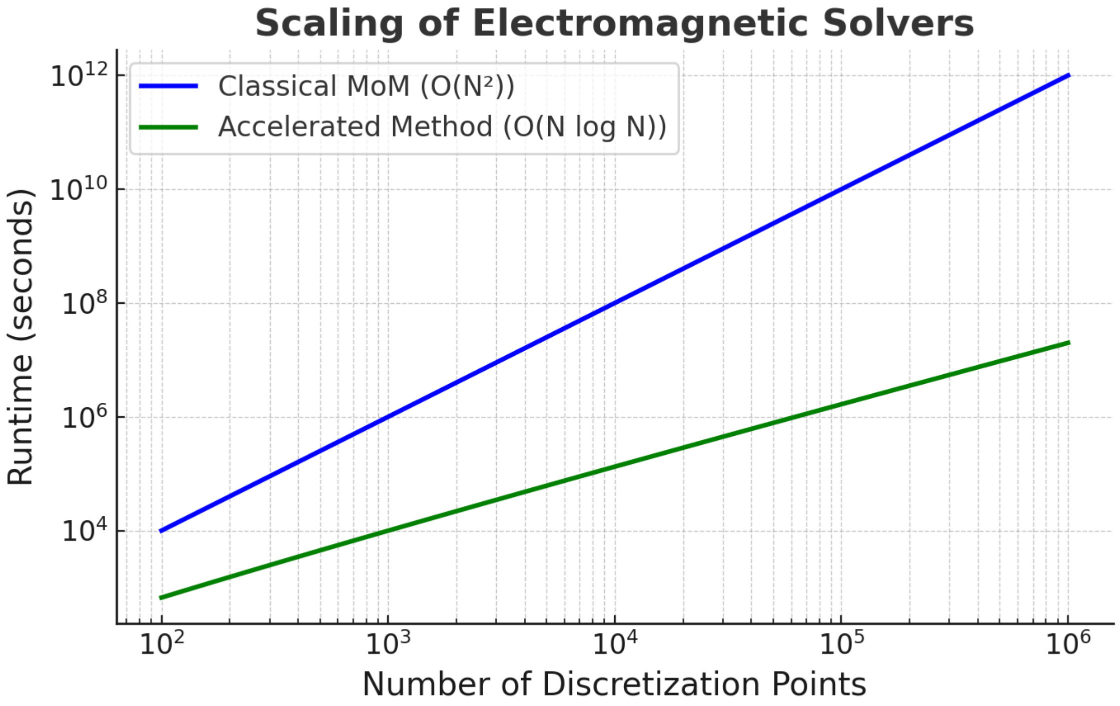
General iterative solver flowchart

The Challenge with Classical Methods

Computing the Green Function matrix and performing the matrix-vector multiplication is **expensive**:

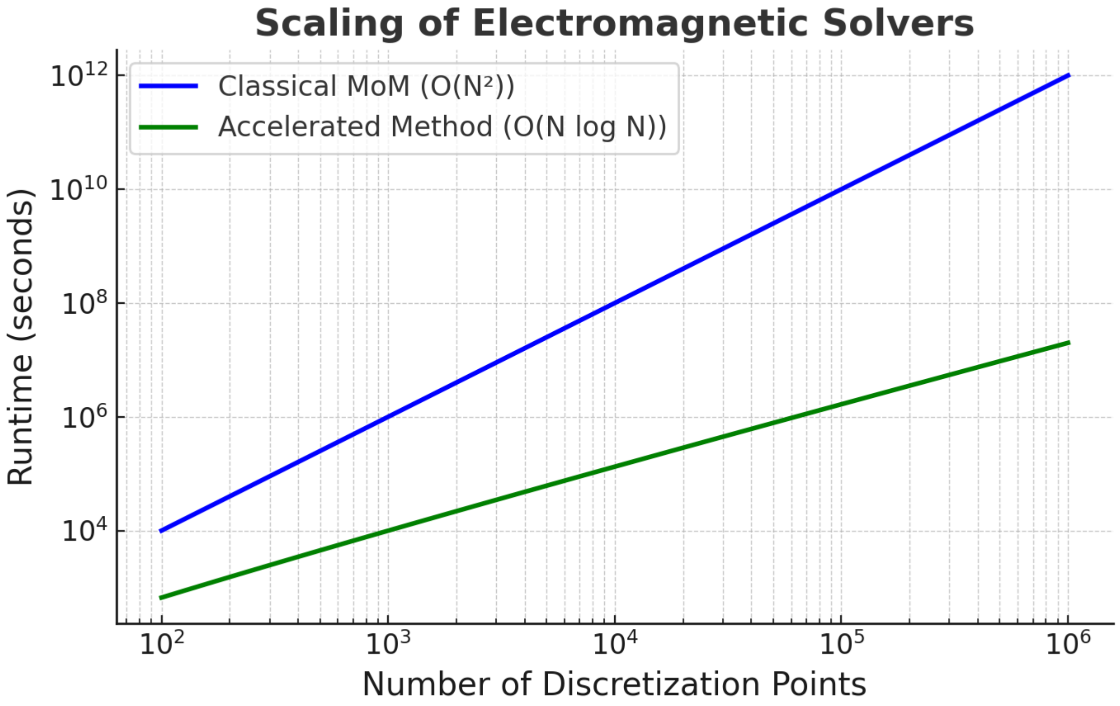
- $\mathcal{O}(N^2)$ memory
- $\mathcal{O}(N^2)$ runtime

This makes large-scale simulations **infeasible** as the number of points grows.



The Challenge with Classical Methods

To simulate modern designs, we need algorithms that scale **better than quadratic.**



Current Solutions for Fast Solvers

Fast industry solvers currently employ some variant of one of the two following algorithms. Both achieve $\mathcal{O}(N \log N)$ time and space complexity.

Fast Multipole Method (FMM)

Groups sources hierarchically; approximates far-field using multipole expansions.

❌ Requires separate integration routines for near- and far-field points.

❌ Accuracy drops as frequency increases – need to compensate with more terms in the multipole expansion

Adaptive Integral Method (AIM)

Interpolates to a uniform grid; uses FFT to accelerate convolution operations.

❌ Difficult to parallelize due to the use of the FFT

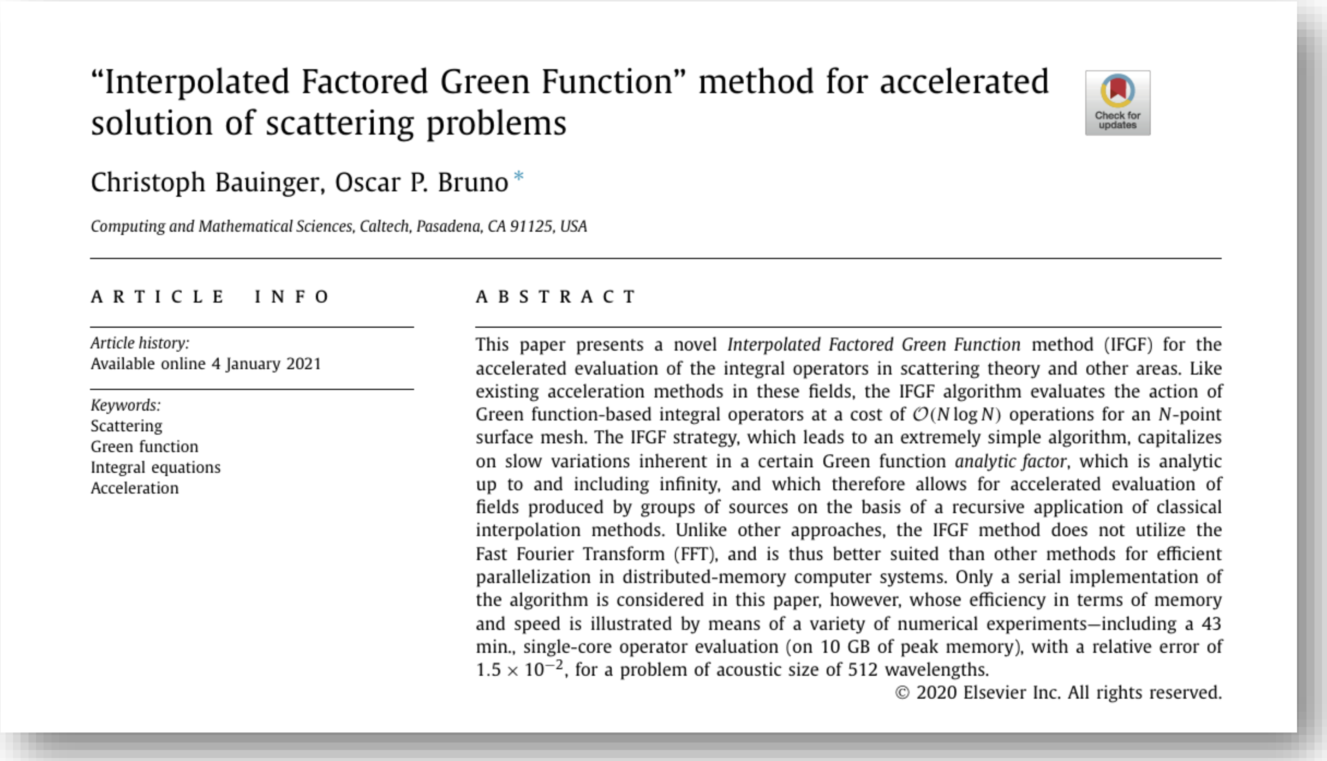
❌ Requires the points be placed on a uniform grid – this makes complex geometries difficult to simulate

Interpolated Factored Green Function Method

A method presented in 2021, promising to compute the matrix-vector product in:

- $\mathcal{O}(N \log N)$ memory
- $\mathcal{O}(N \log N)$ runtime

It does so by approximating the Green's function using **interpolation**



"Interpolated Factored Green Function" method for accelerated solution of scattering problems

Christoph Bauinger, Oscar P. Bruno*

Computing and Mathematical Sciences, Caltech, Pasadena, CA 91125, USA

ARTICLE INFO

Article history:
Available online 4 January 2021

Keywords:
Scattering
Green function
Integral equations
Acceleration

ABSTRACT

This paper presents a novel *Interpolated Factored Green Function* method (IFGF) for the accelerated evaluation of the integral operators in scattering theory and other areas. Like existing acceleration methods in these fields, the IFGF algorithm evaluates the action of Green function-based integral operators at a cost of $\mathcal{O}(N \log N)$ operations for an N-point surface mesh. The IFGF strategy, which leads to an extremely simple algorithm, capitalizes on slow variations inherent in a certain Green function *analytic factor*, which is analytic up to and including infinity, and which therefore allows for accelerated evaluation of fields produced by groups of sources on the basis of a recursive application of classical interpolation methods. Unlike other approaches, the IFGF method does not utilize the Fast Fourier Transform (FFT), and is thus better suited than other methods for efficient parallelization in distributed-memory computer systems. Only a serial implementation of the algorithm is considered in this paper, however, whose efficiency in terms of memory and speed is illustrated by means of a variety of numerical experiments—including a 43 min., single-core operator evaluation (on 10 GB of peak memory), with a relative error of 1.5×10^{-2}, for a problem of acoustic size of 512 wavelengths.

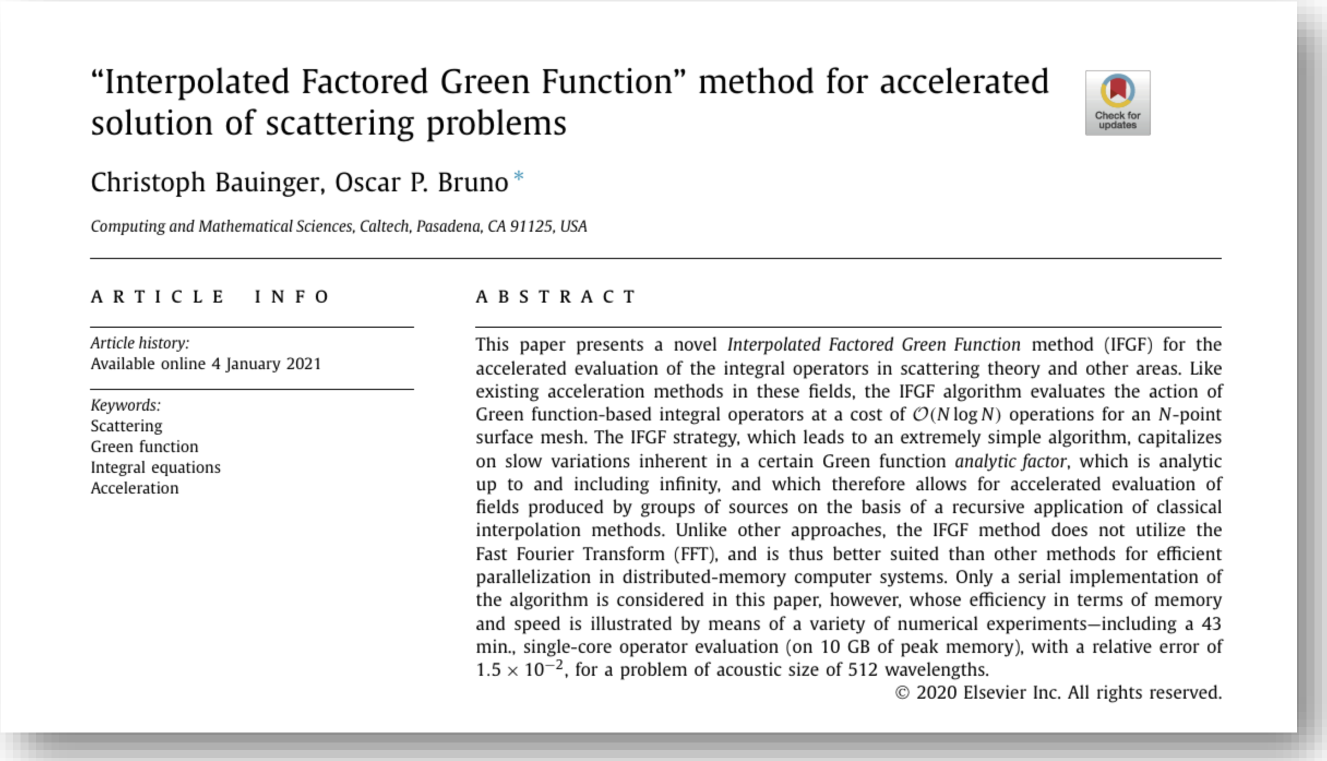
Interpolated Factored Green Function Method

The authors claim some key advantages over AIM and FMM:

✅ Parallelizable due to spatial partitioning in a tree structure

✅ Does not require separate near- and far-field integration routines (this is inherent in the algorithm)

✅ Error remains bounded with increasing wavelength



“Interpolated Factored Green Function” method for accelerated solution of scattering problems

Check for updates

Christoph Bauinger, Oscar P. Bruno *

Computing and Mathematical Sciences, Caltech, Pasadena, CA 91125, USA

ARTICLE INFO

Article history:
Available online 4 January 2021

Keywords:
Scattering
Green function
Integral equations
Acceleration

ABSTRACT

This paper presents a novel *Interpolated Factored Green Function* method (IFGF) for the accelerated evaluation of the integral operators in scattering theory and other areas. Like existing acceleration methods in these fields, the IFGF algorithm evaluates the action of Green function-based integral operators at a cost of $\mathcal{O}(N \log N)$ operations for an N-point surface mesh. The IFGF strategy, which leads to an extremely simple algorithm, capitalizes on slow variations inherent in a certain Green function *analytic factor*, which is analytic up to and including infinity, and which therefore allows for accelerated evaluation of fields produced by groups of sources on the basis of a recursive application of classical interpolation methods. Unlike other approaches, the IFGF method does not utilize the Fast Fourier Transform (FFT), and is thus better suited than other methods for efficient parallelization in distributed-memory computer systems. Only a serial implementation of the algorithm is considered in this paper, however, whose efficiency in terms of memory and speed is illustrated by means of a variety of numerical experiments—including a 43 min., single-core operator evaluation (on 10 GB of peak memory), with a relative error of 1.5×10^{-2}, for a problem of acoustic size of 512 wavelengths.

Engineering Science
UNIVERSITY OF TORONTO

Factoring the Green's Function

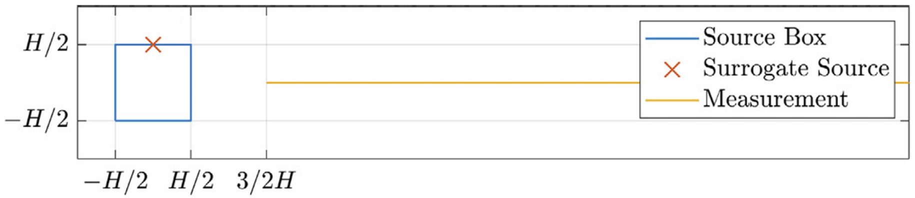
Consider the Helmholtz Green's function:

$$G(\vec{r},\vec{r}') = \frac{e^{jk|\vec{r}-\vec{r}'|}}{4\pi|\vec{r}-\vec{r}'|}$$

Now, say ∃ a square box with side-length S and centered at $\vec{r_s}$ which contains the source, $\vec{r}'$. We can re-write the Green's function for the observation point, $\vec{r}$, as:

$$G(\vec{r},\vec{r}') = \left(\frac{e^{jk|\vec{r}-\vec{r_s}|}}{4\pi|\vec{r}-\vec{r_s}|}\right)\left(\frac{|\vec{r}-\vec{r_s}|}{|\vec{r}-\vec{r}'|}e^{jk(|\vec{r}-\vec{r}'|-|\vec{r}-\vec{r_s}|)}\right)$$

We can call the left factor $G(\vec{r},\vec{r_s})$, which depends **only** on the target point and the box location. The right factor is $g_s(\vec{r},\vec{r}',\vec{r_s})$.



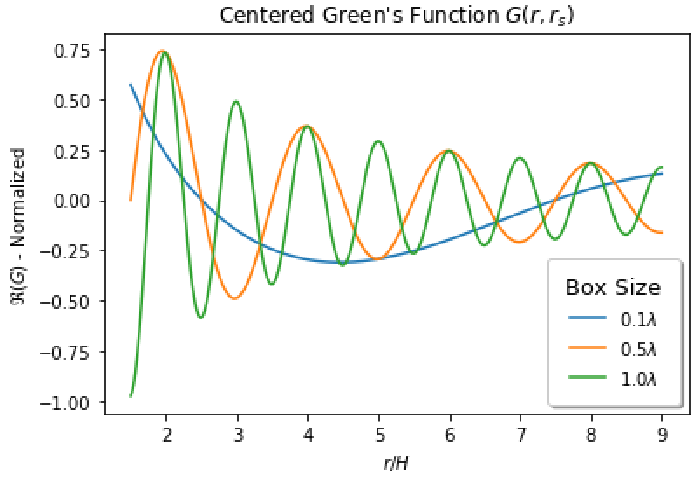
Test setup, assuming the box is centered at the origin

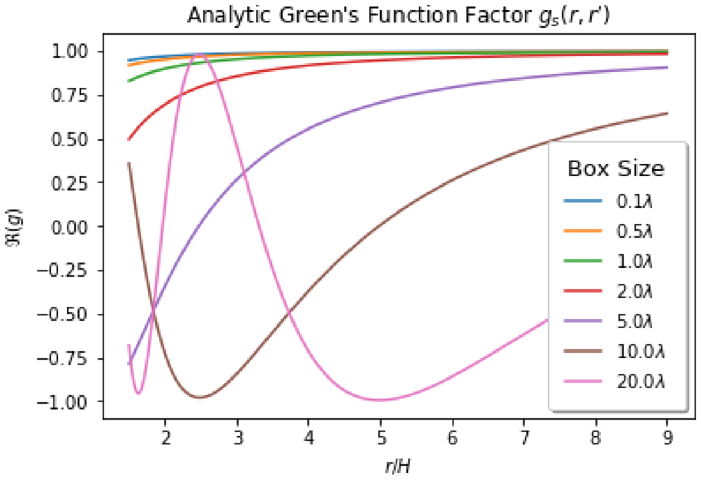
Factoring the Green's Function

We will now assume $\vec{r_s} = 0$ (for simplicity):

Centered factor: $G\left(\vec{r}, \vec{0}\right) = \left(\frac{e^{jk|\vec{r}|}}{4\pi|\vec{r}|}\right)$

Analytic factor: $g_s(\vec{r}, \vec{r}') = \left(\frac{|\vec{r}|}{|\vec{r}-\vec{r}'|} e^{jk(|\vec{r}-\vec{r}'|-|\vec{r}|)}\right)$


Centered Green's Function $G(r, r_s)$


Analytic Green's Function Factor $g_s(r, r')$

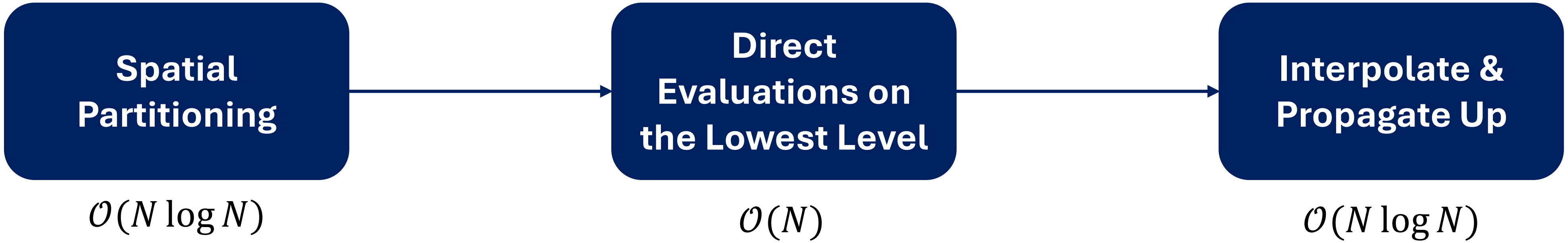
Notice the bottom plot. Even for electrically-large box sizes, the analytic factor experiences **slow** oscillations (over the box)

⇒ Can be **interpolated** using polynomials!

Now, we only need to compute G directly for a source at the center of each box and sample a few other *interpolation points*.

Algorithm Flowchart

There are 3 main steps in the IFGF algorithm:



Spatial Partitioning

$\mathcal{O}(N \log N)$

Direct Evaluations on the Lowest Level

$\mathcal{O}(N)$

Interpolate & Propagate Up

$\mathcal{O}(N \log N)$

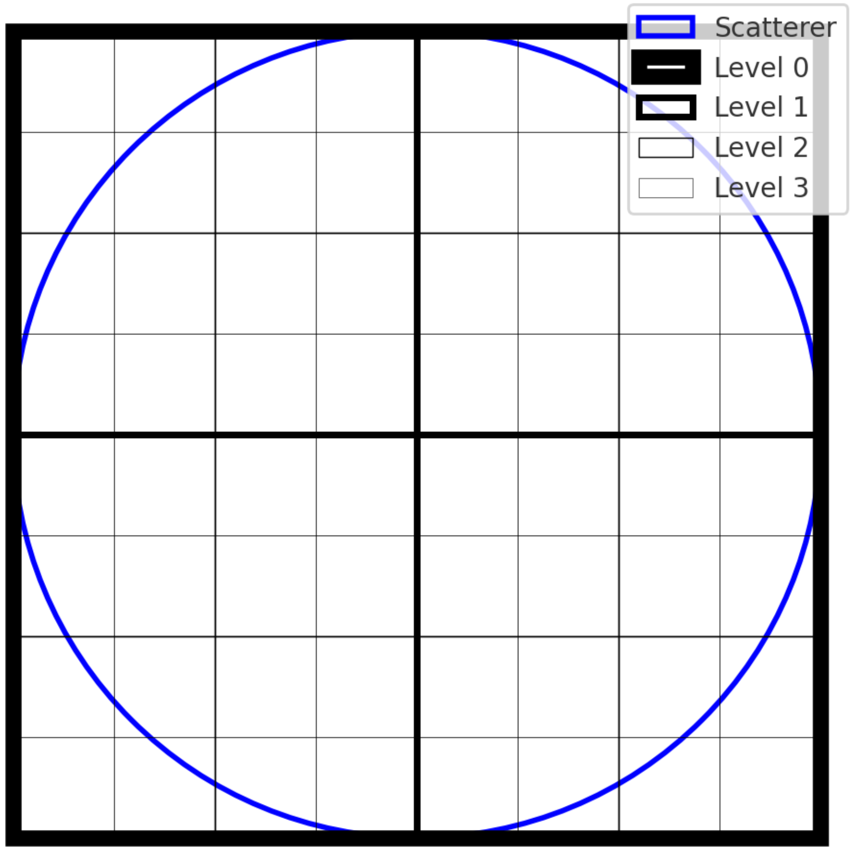
1. Spatial Partitioning

IFGF begins by subdividing the space which bounds the scatterer into an *octree* of *boxes*.

When moving from level d $\rightarrow (d+1)$, each box spawns 8 new boxes (4 in 2D).

This is repeated until the side-length, H_D, at the leaf depth, D, is $\frac{\lambda}{4}$. This ensures that the Green function does not vary much over the lowest boxes.

IFGF Spatial Partitioning



2D circular scatterer with $r = \lambda$

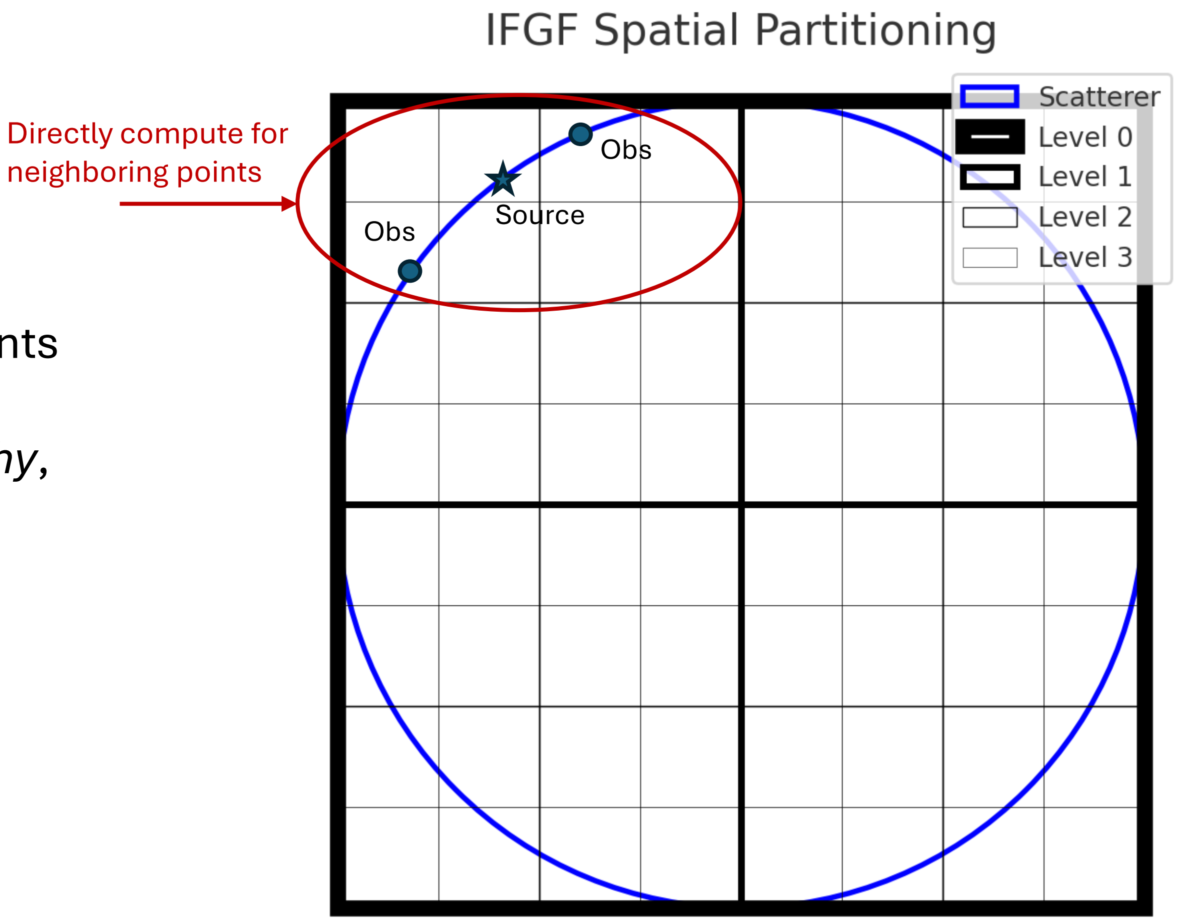
Engineering Science
UNIVERSITY OF TORONTO

2. Direct Evaluations on Lowest Level

On the leaf level of the octree, compute the interactions **only** between points in **adjacent** boxes.

Also, choose a fixed number of interpolation points per box and treat those as observation points

- Selecting these follows the *cone hierarchy*, explained later


IFGF Spatial Partitioning
Directly compute for neighboring points
Obs
Source
Obs
Scatterer
Level 0
Level 1
Level 2
Level 3

2D circular scatterer with $r = \lambda$

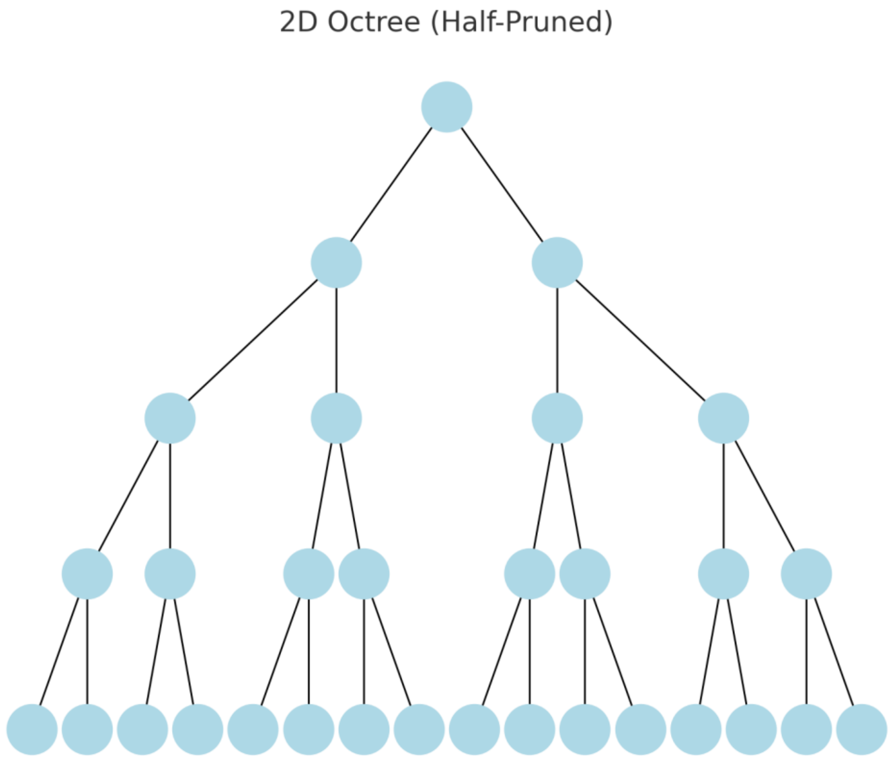
3. Interpolate & Propagate Up

For each box, we interpolate the field using the analytic factor as $F_k^d(\vec{r}) = \frac{I_d^k(\vec{r})}{G\left(\vec{r}, \overrightarrow{r_k^d}\right)}$, where $I_d^k(\vec{r})$ is computed by interpolation.

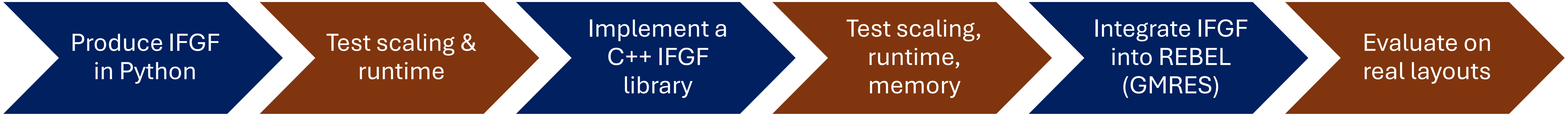
Moving up the tree, the field at each point is re-centered using:

$$F_j^{d-1} = \sum_{children\ of\ k} \frac{G\left(\vec{r}, \overrightarrow{r_k^d}\right)}{G\left(\vec{r}, r_j^{d-1}\right)} F_k^d(\vec{r})$$

At the root, we are left with the approximated full field.



Project Roadmap



Produce IFGF in Python

Test scaling & runtime

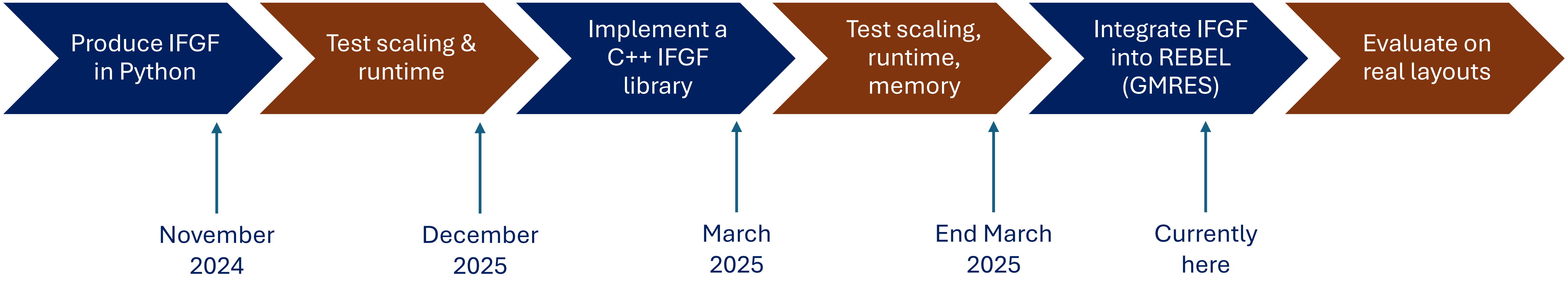
Implement a C++ IFGF library

Test scaling, runtime, memory

Integrate IFGF into REBEL (GMRES)

Evaluate on real layouts

Project Timeline



Produce IFGF in Python

Test scaling & runtime

Implement a C++ IFGF library

Test scaling, runtime, memory

Integrate IFGF into REBEL (GMRES)

Evaluate on real layouts

November 2024

December 2025

March 2025

End March 2025

Currently here

File Structure

ifgf.py

- Top-level. Implements the evaluation of the three-step algorithmic process

interpolation.py

- Implements the Chebyshev interpolation routine

octree.py

- Implements the tree data structure and splitting

cone.py

- Implements the interpolation cone classes

boundingbox.py

- Implements the boxes for the octree class

kernels.py

- Implements the Helmholtz and Laplace kernels. Also performs direct evaluations

utils.py

- Misc. utility functions

Data Structures

Class IFGF: the top-level class for implementing IFGF

- `IFGF(source_points, target_points, kernel)`
    - Prepares spatial partitioning (downward pass) – Octree and Cones
- `IFGF.evaluate(weights)`
    - Performs direct evaluation and interpolation (upward pass)

Class Octree: the core data structure responsible for spatial partitioning. Each Octree object is also a node.

- `Octree(source_points, target_points, level, parent, children)`
    - Builds an Octree node
- `Octree.split(criterion)`
    - Recursively builds the tree
- `Octree.compute_interaction_list:`
    - Computes the relevant boxes on which to interpolate onto
- `Octree.generate_cones()`
    - Computes the cone domains and spawns Cone objects

Data Structures

Class BoundingBox: The object which contains the information on points in the box

- `BoundingBox(source_points, target_points, r_center, side_length)`
  - Builds a box containing the points
- `BoundingBox.split()`
  - Spawns children boxes (not recursive)

Class Cone: The main object for interpolation

- `Cone(source_points, interpolation_points)`
  - Builds the arrangement of interpolation points within a cone domain
- `Cone.refine(criterion)`
  - Determines how the cones will split when moving up a level (depends on the electrical length of the underlying box)

Data Structures

Class Kernel: Defines the underlying nature of the scattering problem

- `Kernel(wavenumber)`
  - Constructs the appropriate Green Function kernel (Helmholtz or Laplace). The wavenumber can be complex
- `Kernel.evaluate(source_points, target_points, weights)`
  - Directly solves the matrix-vector multiplication for a subset of points
  - Can call this function on all points to get the inefficient $\mathcal{O}(N^2)$ solution
  - Used in validation and on the lowest-level of the Octree

Translation to C++

Goal is to provide a serial implementation in the form of a library which can be called by any iterative solver.

```
kernel = Kernel(wavenumber)
ifgf = IFGF(sources, targets, kernel)
ifgf.evaluate(weights)
```

File structure looks the same as the Python implementation, with HPP headers.

Used the MinGW64 compiler.

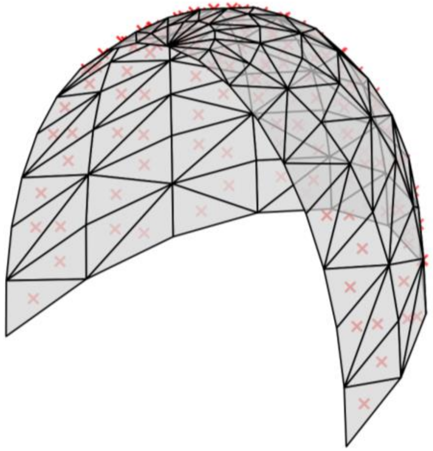
Used C++ standard library and Eigen for linear algebra.

- Eigen is an easy translation from NumPy

Integration into REBEL: Geometry

REBEL converts .gds files into a triangular mesh. IFGF needs a point-cloud to work.

To solve this, we consider one point per triangle, at the centroid.



Converting a spherical surface mesh to a point-cloud

Integration into Rebel: Integration

Rebel explicitly solves three types of integrals in its current formulation:

1. Singular integrals: occurs when $\vec{r_i} \approx \vec{r_j}'$. In IFGF we do **not** compute these – instead we leave Rebel to solve this using singularity extraction.
2. Far-field integrals
3. Near-field integrals

IFGF handles these **internally** – we just pass all non-singular integral points to the IFGF solver without making the distinction.

Testing Protocol: Measuring Error

When reporting the error of the algorithm, we compared the results produced by IFGF to those produced by naïvely applying the kernel function to all source and target points (i.e., the direct evaluation method)

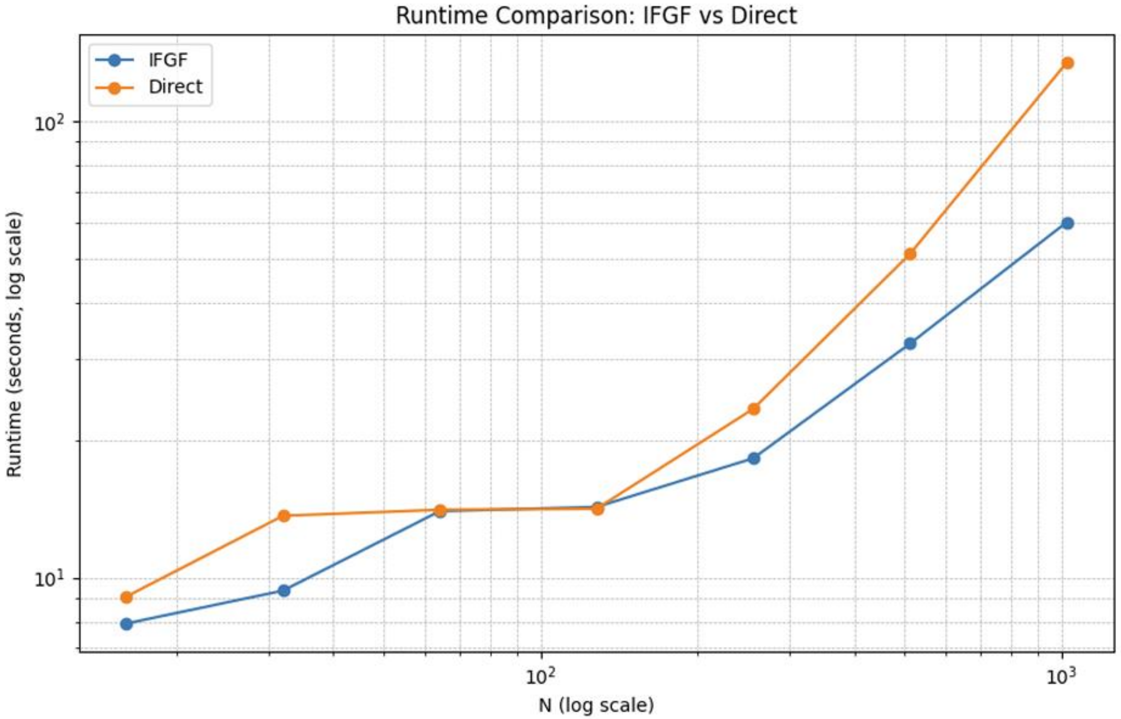
The error metric used was the root-mean-square-error (RMSE):

$$\epsilon = \sqrt{\frac{\sum_N \left| I_{IFGF}^{(i)} - I_{Direct}^{(i)} \right|^2}{\sum_N \left| I_{Direct}^{(i)} \right|^2}}$$

where $I_{IFGF}^{(i)}$ is the result at the i^{th}observation point computed with IFGF and
$I_{Direct}^{(i)}$ is the result at the i^{th}observation point computed with the kernel directly

Preliminary Python Runtime Analysis

The test was conducted on a sphere with radius 1m and wavenumber 8π rad/m. The point density was increased



Engineering Science
UNIVERSITY OF TORONTO

Tests Conducted for C++ Evaluation

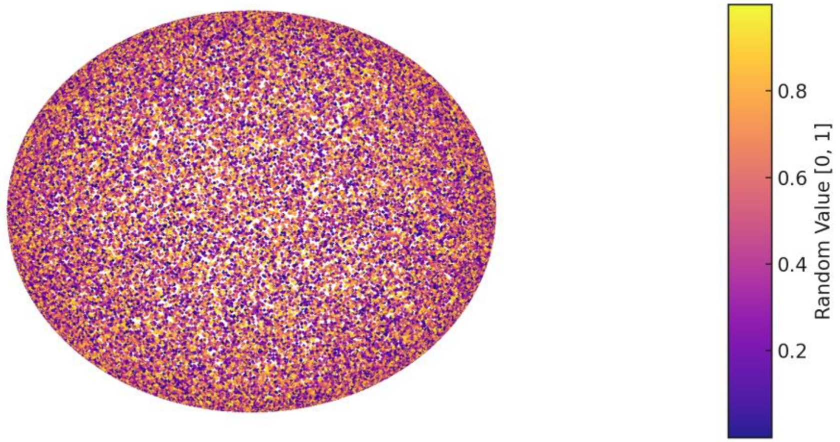
All source coefficients were initialized to a value $\in [0,1]$

Wavenumber of $k = 2\pi$ [rad/m] was used for all cases

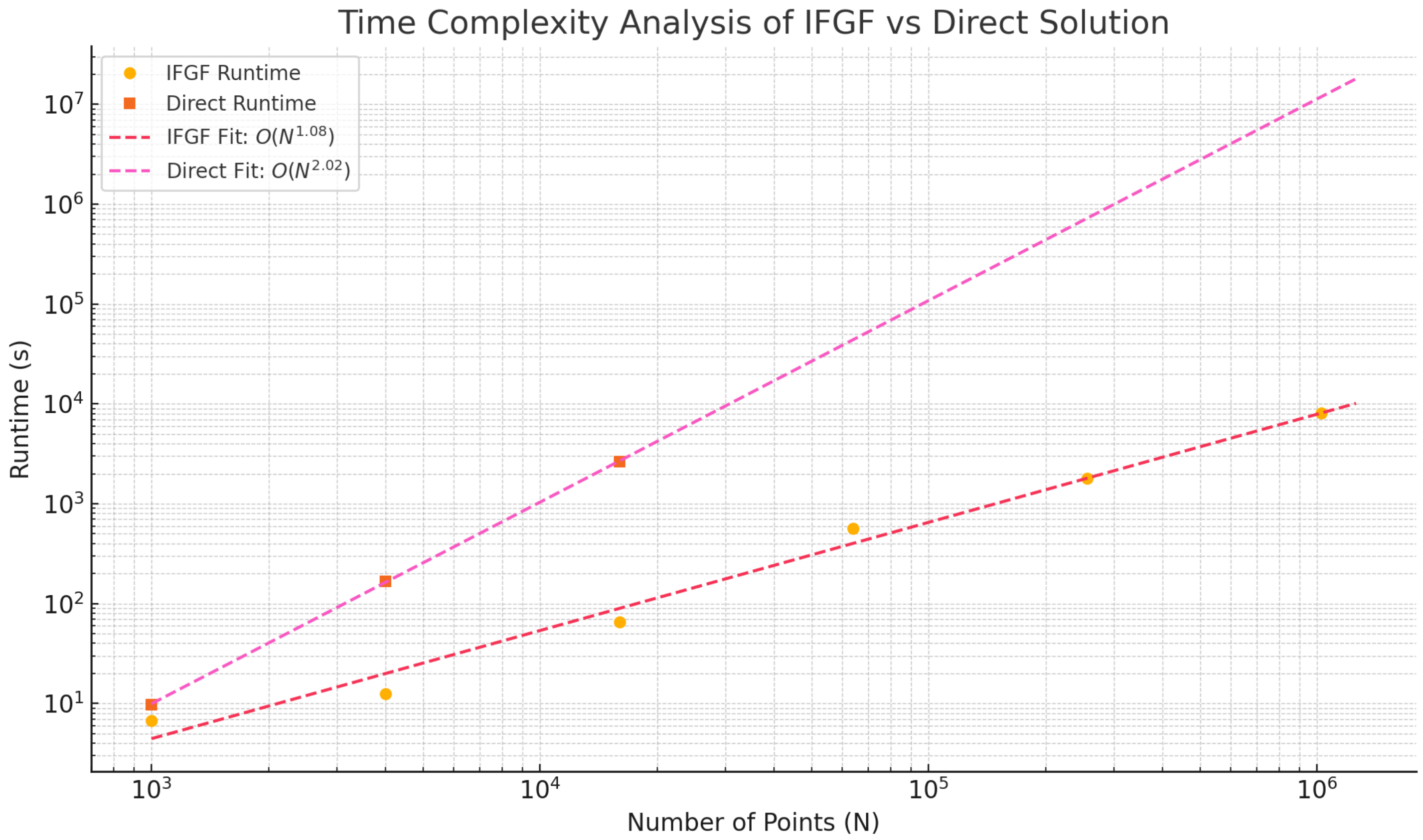
Radius of the sphere doubled each time: 1,2,4,8,16,32 [m]

Number of points varied proportionally to the area of the surface:
1, 4, 16, 64, 256, 1024 [x10^3 points]

Point-cloud used for r=8m, N=64000 evaluation



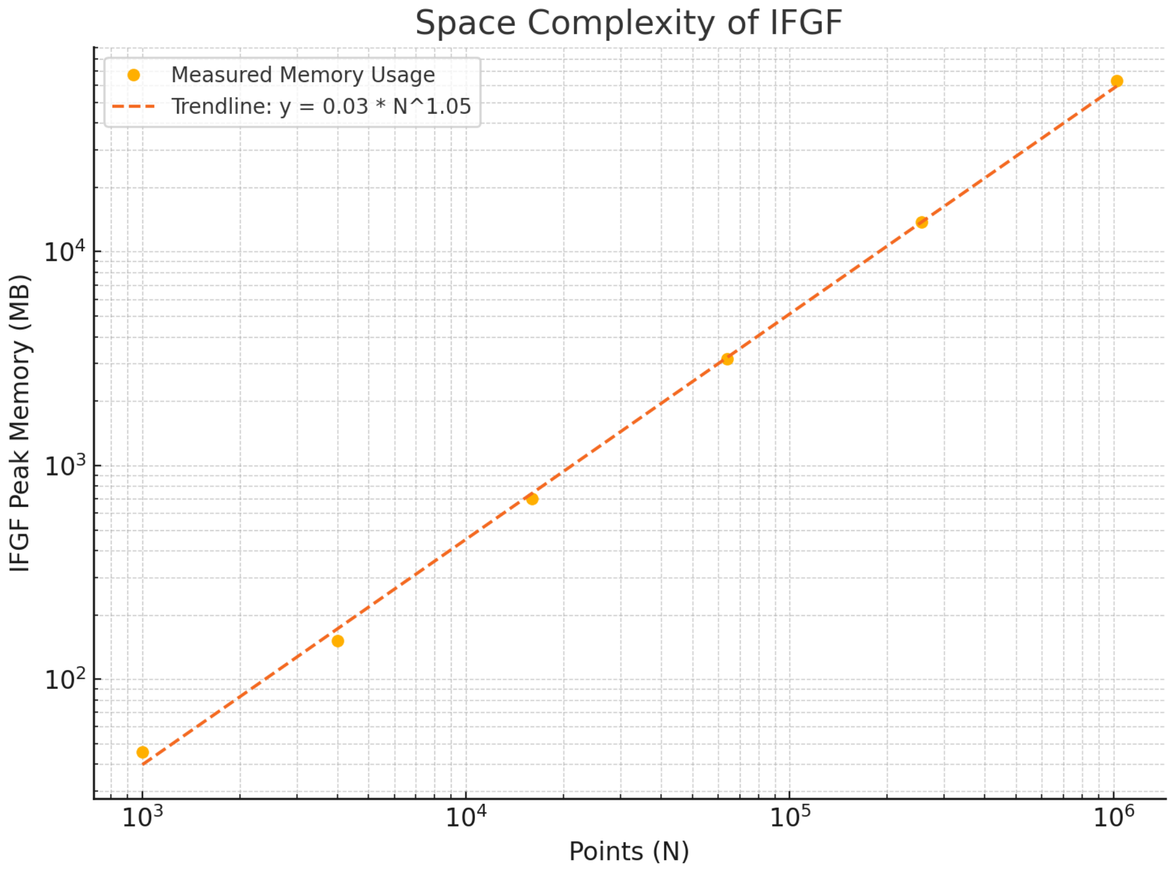
C++ Runtime Analysis



Engineering Science
UNIVERSITY OF TORONTO

C++ Peak Memory Analysis

Used the <psapi.h> library interface, linked with MinGW



Engineering Science
UNIVERSITY OF TORONTO

C++ Error Analysis

Due to the runtime limitations, the $\mathcal{O}(N^2)$ direct computation was only performed on the first 3 datapoints. The errors for these are listed below:

Wavenumber [rad/m]	Radius [m]	Points	RMSE Error (x10^{-3})
2π	1	1000	4.10
2π	2	4000	6.80
2π	4	16000	16.20

Next Steps: Finishing REBEL Integration

I have a functioning header library which compiles and can construct the IFGF operator class.

REBEL is installed, currently through the use of Docker.

I have created a wrapper header for Eigen, similar to what is done currently with LAPACK. I am able to link REBEL against Eigen

I have created a new directory named “/shared_memory/ifgf”

Started writing code for the swapping of the near- and far-field integrals. Need to change the main rebel file to include this acceleration and run some unit tests still

Instead of including Eigen as the wrapper, exploring using a SLL or DLL instead

Conclusion

It is evident that IFGF provides the much-needed acceleration for electromagnetic scattering problem solutions.

A performance and accuracy comparison against FMM and AIM is the next step towards demonstrating the scalability of this algorithm.

Implementing and Testing the

Interpolated Factored Green Function Method

For the Accelerated Evaluation of Potentials in Electromagnetic Simulations

Michael P. Acquaviva

Supervised by: Piero Triverio

Engineering Science
UNIVERSITY OF TORONTO