Parallel Helmholtz Equation Wave Propagation through Waveguides

Edmund Chen

CS 315B

December 14, 2022

Outline

Problem Statement

- Consider the time-independent component rendered from the *wave equation* under separation of variables
  - Sommerfeld radiation condition for *unique radiating solution*
- Models wave propagation within waveguides

$$
\begin{aligned}
-\nabla^2 u - k^2 u &= f && \text{on } \Omega \\
\mathbf{n} \cdot \nabla u &= 0 && \text{on } \Gamma_{\text{wall}} \\
\mathbf{n} \cdot \nabla u + iku &= 0 && \text{on } \Gamma_{\text{out}} \\
\mathbf{n} \cdot \nabla u + iku &= 2ik && \text{on } \Gamma_{\text{in}}
\end{aligned}
$$

Galerkin Finite Element Method

- Looking for approximate solution in space of linear continuous functions V_h on some mesh T_h (Bubnov-Galerkin)

$$V_h = \{v \in C^0(\Omega) : v|_k \in \mathbb{P}_1(K) \forall K \in T_h\}$$

- We seek some approximate solution u_h with the aid of a test function v_h, both within V_h
  - Impose the *weighted average* with v_h over each element and solve

Galerkin Finite Element Method

- Integrate on domain, pop out boundary conditions, apply divergence theorem to get a *weak formulation* of the PDE

$$\int_\Omega -\nabla^2 u_h v d\Omega = \int_\Omega k^2 u_h v d\Omega$$

$$\int_\Omega \nabla u_h \cdot \nabla v d\Omega = \int_\Omega k^2 u_h v d\Omega + \oint_{\Gamma_{out}} (-iku_h) v ds + \oint_{\Gamma_{in}} (2ik - iku_h) v ds$$

- Hence the problem, find $u_h \in V_h$ which satisfies this

Galerkin Finite Element Method

- Replace the approximate solution with a weighted average of basis functions, and the test function with a basis function, $\varphi(\mathbf{x})$

$$u_h = \sum_{j=1}^{N} u_j \varphi_j \qquad v = \varphi_i$$

- Gives the discretization of form

$$\int_\Omega [\sum_{j=1}^{N} u_j \nabla \varphi_j] \cdot \nabla \varphi_i(x) d\Omega = \int_\Omega k^2 [\sum_{j=1}^{N} u_j \varphi_j] \varphi_i d\Omega -$$

$$\oint_{\Gamma_{out}} ik [\sum_{j=1}^{N} u_j \varphi_j] \varphi_i ds + \oint_{\Gamma_{in}} (2ik - ik [\sum_{j=1}^{N} u_j \varphi_j]) \varphi_i ds$$

Galerkin Finite Element Method

Factoring out admits a discretization of form $\mathbf{Au} = \mathbf{b}$ where

$$\mathbf{A} = K - k^2 M + ik(B_{out} + B_{in}) \qquad \mathbf{b} = 2ik\mathbf{b}_{in}$$

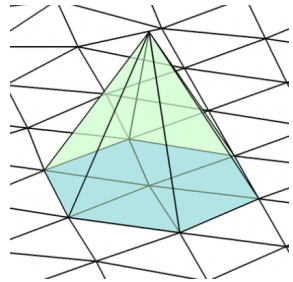
$$\sum_{j=1}^{N} u_j \left(\int_\Omega \nabla\varphi_i \cdot \nabla\varphi_j d\Omega - k^2 \int_\Omega \varphi_i\varphi_j d\Omega + ik \left(\oint_{\Gamma_{out}} \varphi_i\varphi_j ds + \oint_{\Gamma_{in}} \varphi_i\varphi_j ds \right) \right)$$
$$= 2ik \oint_{\Gamma_{in}} \varphi_i ds$$

$$K = \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_j d\Omega \quad M = \int_\Omega \varphi_i\varphi_j d\Omega \qquad B_{out} = \oint_{\Gamma_{out}} \varphi_i\varphi_j ds$$

$$B_{in} = \oint_{\Gamma_{in}} \varphi_i\varphi_j ds \qquad \mathbf{b}_{in} = \oint_{\Gamma_{in}} \varphi_i ds$$

Basis functions

- For some triangle element on the mesh T^k bounded by x_1^k, x_2^k, x_3^k
- Corresponding basis functions satisfy $\varphi_i^k(x_j^k) = \delta_{ij}$



- Say we have linear basis functions of the form $\varphi_i^k = c_0 + c_1 x + c_2 y$, we need to solve the following system for each triangular element on the mesh

$$\begin{pmatrix} 1 & x_1^k & y_1^k \\ . & & \\ . & & \end{pmatrix} \begin{pmatrix} c_0 & . & . \\ c_1 & & \\ c_2 & & \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Numerical Quadrature

- Use Gaussian quadrature to evaluate integrals which cannot be solved analytically with x_i the roots of the nth Legendre polynomial

$$\int_1^{-1} f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

- For instance, K becomes (for mesh element T^k)

$$K^k = \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_j d\Omega$$
$$= \int_{T_k} \partial_x\varphi_i \partial_x\varphi_j + \partial_y\varphi_i \partial_y\varphi_j d\Omega = \text{Area}(T^k)(c_{1,i}c_{1,j} + c_{2,i}c_{2,j})$$

Process

So basically, the process is as follows

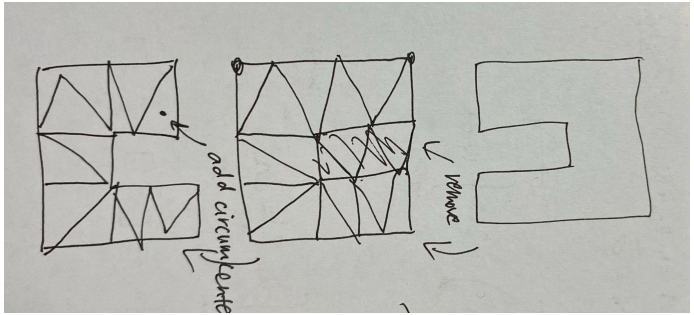
- generate a mesh
- for every triangular element
  - calculate area
  - calculate basis functions (linear system)
  - calculate resulting mass, stiffness, etc. matrices w/ quadrature
  - stamp it into a global matrix
- assemble into **Au=b**
- solve for u, solution is real part of u

Naturally, many loops and systems that can be parallelized. Note the resulting system is very sparse.

Outline

Meshing

- Well formed triangular mesh
- Use delaunay refinement for meshing, then iteratively approve upon it
- See “mesh.mov” in submission



Meshing Parallelization

- Parallelize the parts where we are identifying the set of poor quality triangles

- Rest is hard to negotiate since triangulation is black-boxed and adding multiple circumencenters at once risks inconsistent element sizes

```
function Ruppert(points, segments, threshold) is
    T := DelaunayTriangulation(points)
    Q := the set of encroached segments and poor quality triangles

    while Q is not empty:                    // The main loop
        if Q contains a segment s:
            insert the midpoint of s into T
        else Q contains poor quality triangle t:
            if the circumcenter of t encroaches a segment s:
                add s to Q;
            else:
                insert the circumcenter of t into T
            end if
        end if
        update Q
    end while

    return T
end Ruppert.
```

Element Matrices Construction

- Use numerical quadrature for each
- Main loop index on numpy matrix dimensions
  - Internally use lambda functions to calculate appropriate values and “stamp” into global matrix
- Elements need to be sufficiently removed from their neighbors to avoid having two threads fighting over same parts of global matrix

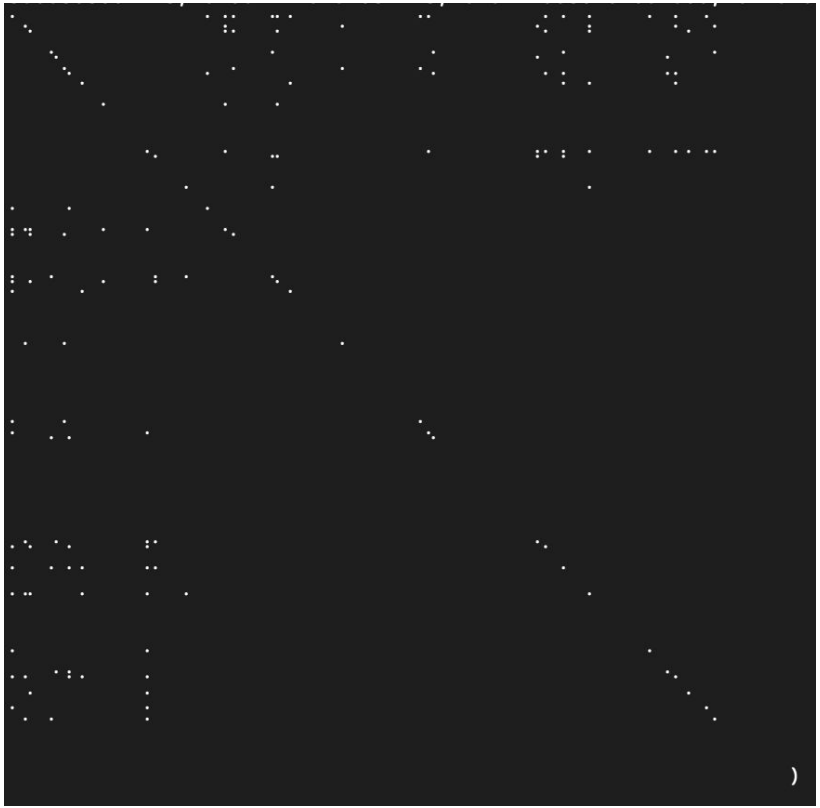
```
def loadvec(p, t, e):
    k, i, j = find_bnd(e, t)
    cmx = coeffmx(np.identity(3), p, t, k)
    f = lambda x,y:((cmx[0,i] + cmx[1,i]*x)*y + (cmx[2,i]*y**2)/2.0)
    g = lambda x,y:((cmx[0,j] + cmx[1,j]*x)*y + (cmx[2,j]*y**2)/2.0)

    if p[t[k,i],1] > p[t[k,j],1]:
        id = f(p[t[k,i],0],p[t[k,i],1]) - f(p[t[k,i],0],p[t[k,j],1])
        jd = g(p[t[k,i],0],p[t[k,i],1]) - g(p[t[k,i],0],p[t[k,j],1])
    else:
        id = f(p[t[k,i],0],p[t[k,j],1]) - f(p[t[k,i],0],p[t[k,i],1])
        jd = g(p[t[k,i],0],p[t[k,j],1]) - g(p[t[k,i],0],p[t[k,i],1])

    return np.array([id, jd]) #review
```

Basis functions and Numerical quadrature calculation

- Cunumeric and other parallel libraries can further parallelize certain processes within this
- Prefer to use operations with numpy arrays as opposed to naive loops

Overarching Linear System

- Very sparse linear system to be solved
- Symmetric, as it is a nxn matrix with n elements on each side
- Hard to exactly transform to strictly diagonal matrix due to unstructured mesh making elements spatially nearby each other far apart index-wise



Parallelization Comments

- Cunumeric does not parallelize most linalg operations
  - Solving the actual linear system incurs a heavy penalty due to this
- System is very sparse, but since numpy does not natively implement sparse matrices, converted to scipy for final linear system solve
  - Sparse matrices implemented from SciPy

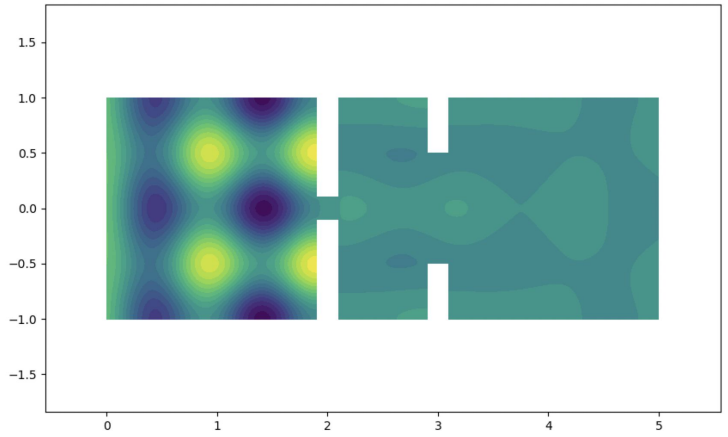
Parallelization Comments

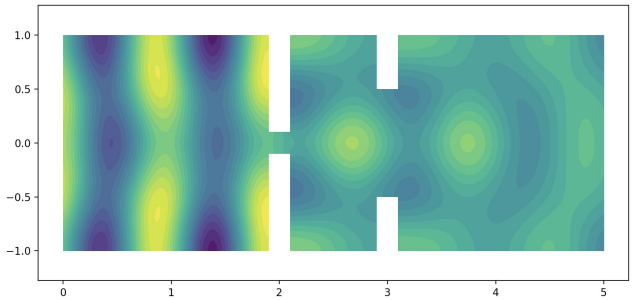
- Cunumeric does not parallelize most linalg operations
  - Solving the actual linear system incurs a heavy penalty due to this
- System is very sparse, but since numpy does not natively implement sparse matrices, need to solve a dense matrix
  - Sparse matrices implemented from SciPy

Numerical Results

- We see resonance phenomena for different wavenumbers, here for 6 - 6.5 range

- Ran for a mesh element quality of 0.2 and an uniform refinement 2x over





Results

- Ran on g4dn.xlarge on AWS (1 NVidia T4 CPU) with cunumeric, python, and julia implementations
- Roughly 3x speedup on critical section of matrix stamping on Python/cuNumeric with 1 GPU
  - Not great admittedly, cost incurred from dense matrix solve
  - Should have converted at end into sparse matrix

```
ED> done with mesh_import()
ED> done with waveguide_edges()
ED> done with femhelmholtz1()
ED> done with femhelmholtz2()
ED> done with femhelmholtz3()
ED> done with femhelmholtz4()
ED> femhelmholtz execution time: 7.039739370346069  seconds
ED> A\B execution time: 22.86920189857483  seconds
ED> Overall execution time: 32.64009618759155  seconds
[ 1.04402865 -0.49637759 -1.91578359 ...  0.90199697  1.00425585
  0.54496894]
```

Results

- Also parallelized with Numba to test performance (helmholtz-numba.py
- Fast with vectorization and compilation to c code optimizations
    - 5s elapsed baseline

Results

- Also implemented and ran on Julia language (CPU parallelization)
- 1 GPU accelerant roughly correspond to 3x CPU accelerant

```
generating mesh:
  4.704548 seconds (15.24 M allocations: 977.409 MiB, 4.65% gc time, 2.62% compilation time)

finding mesh boundaries:
  0.340401 seconds (931.75 k allocations: 119.770 MiB, 3.48% gc time, 61.24% compilation time)

generating constituent matrices:
 28.525171 seconds (18.06 M allocations: 144.674 GiB, 5.27% gc time, 4.72% compilation time)

solving linear system:
  1.181536 seconds (4.29 M allocations: 299.368 MiB, 14.62% gc time, 88.33% compilation time)
```

Outline

Local Discontinuous Galerkin Method

- The main obstacle for enabling more parallelism is the basis functions $\varphi_i(x_j) = \delta_{ij}$ needing to be continuous, hence the necessary synchronization when assembling global matrix and solving for coefficients on every element
- Discontinuous-Galerkin methods allow these to be piecewise constant functions, making the mass matrix block-diagonal since there are no inter-element basis function dependencies

$$u_h = \sum_{k=1}^{n} \sum_{i=0}^{p} u_i^k \varphi_i^k(x)$$

Local Discontinuous Galerkin Method

- Cannot be directly done on anything higher than first-order spatial derivative
- Rewrite into a system of first-order equations and choose fluxes appropriately, hence the Local DG method

$$-\nabla\sigma = k^2 u \qquad \nabla u = \sigma$$

- This becomes **embarrassingly parallel** because I can parallelize across elements in constructing elemental submatrices matrices without race conditions on stamping into the same place on the global matrix

Parallel Unstructured Mesh Generation

- Most naive way to pre-section mesh into a few blocks so at least one process can take each block
- Falls apart for more complicated unstructured meshes unfortunately