Assignment 6 : Multigrid and FFT

due : April 12, 2022 – 11:59 PM

GOAL: MG (multigrid) and FFT (fast Fourier transform) are recursive divide and conquer methods that are the bedrock of fast algorithms in both computer science tasks and numerical methods for high performance computing. They work magically but to appreciate them you just need to code them. **Their magic is easier to see in practice than to explain: Learn by doing!**

Test Problem: Laplace Solver with Point Charge

Let's start again with the simple iterative solvers Jacobi, Gauss Seidel and Red/Black (or even/odd) iterations in 1D. We want to compare them in 1D and 2D with multigrid and direct solve be Fourier Transforms! Remember all these methods must give the same solution when they converge, so you can use one method to debug the next. Also in 1D the **exact** solution is known even on a grid with spacing h, so there is no confusion of what the answer is. See the figure in HW6code. (Using a simple exact solution to debug a code is perhaps the most important debugging tool used by experts but too often not taught in class rooms.) The 1D Laplace equation for a function $\phi(x)$ of a real variable $x \in [a, b]$ and put it on a grid with spacing h is:

$$-\frac{d^2\phi(x)}{dx^2} = b(x) \rightarrow -\frac{2\phi[i] - \phi[i-1] - \phi[i+1]}{h^2} = b[i] \tag{1}$$

Let's take this to model temperature in the cold night with a heater in the middle of the 1D room. The walls are set to temperatures T_1 and T_2. For simplicity let's take $a = -1, b = 1$ with a grid from $i = -N, \cdots N$ with $h = 1/N$. Let's set the walls to absolute zero $T_1 = 0$ and $T_2 = 0$ with a single heater in the middle of the room! (I guess you are in outer space in a linear room. The matrix `A T = b` equation is

$$T[i] - \frac{1}{2}(T[i-1] + T[i+1]]) = h(\alpha/2)\delta_{x,0} \tag{2}$$

with $T[N] = T[-N] = 0$. Actually we know the solution **even on a grid with spacing** $h = 1/N$!

$$T[i] = (N - |i|)(h\alpha/2) \quad \text{for} \quad x = ih \tag{3}$$

Since this is exact, we can even look at for $h \rightarrow 0$.

$$-\frac{d^2T(x)}{dx^2} = \alpha\delta(x) \tag{4}$$

whatever "$\delta(x)$" means. The rule is the integral over $\delta(x)$ is 1. Let's use this as the definition. (Physicists and engineers cheat this way all the time. Actually it is not cheating – it is smart. [1])

[1] See comment in in the very interesting book *The history of Pi* by Petr Beckman that "What especially outranged the mathematicians was not so much that electrical engineers continued to use it (e.g. delta functions) but that it would almost always supply the correct result"

Now with $h = 1/N$ our solution is $T = h(N - |i|)(\alpha/2) \to (1 - |x|)(\alpha/2)$ is independent of h so it is also the continuum solution when we take $h \to 0$ or $N \to \infty$. It has zero second derivative for all $x \neq 0$. What happens at $x = 0$? Lets check the solution by comparing the LHS and RHS of Eq. 4 near $x = 0$,

$$\begin{aligned} \text{LHS} &= -\int_{-\epsilon}^{\epsilon} \frac{d^2T(x)}{dx^2} = -\frac{dT(x)}{dx}|_{-\epsilon}^{\epsilon} = (1+1)\alpha/2 \\ \text{RHS} &= \int_{-\epsilon}^{\epsilon} \alpha\delta(x) = \alpha \end{aligned} \tag{5}$$

for $\epsilon > 0$.

0.1 Doubling Trick

The code on GitHUB has periodic boundary conditions, stabilized by a small mass term instead of fixed boundaries or a source. As a convenience we introduce a **doubling trick** to replace the boundary condition with $T = 0$ by an antisymmetric periodic problem. This avoids using boundary conditions with a doubled periodic lattice with images charges. Helpful to simplify our MG and FFT exercise. The exact solution we gave above strictly is the limit to zero mass. Try it and you will see.

The problem we described above has fixed boundary conditions so the value of $T[i]$ at $i = 0$ and $i = 2N$ are not variables. So there are really are $2N - 1$ free variables. We would like that to have 2^n variables to do the Multigrid or FFT so let's be creative but with $2N - 1 = 2^n$ this is impossible.

There are lots of methods to deal with boundary condition BUT to make life easy (always a good idea at first) let's turn this into a periodic problem. This is easy. Just double the size of the problem and make it odd with reflection around the boundaries. To do this take double the $2N + 1$ points to $N_0 = 4N + 2 - 2 = 4N$ points and make the system anti-periodic. (We have - 2 because when you joint the two parts the $T = 0$ ends are identified.) Namely put a positive source at $i = N$ again and a negative source at $i = -N$ which is equivalent mod N_0 to a negative source at $i = 3N$ (i.e. $-NmodN_0 = 3N$). Now the solution automatically will vanish at $i = 0$ and $i = 2N$ by symmetry so we can solve this problem with multigrid and FFTs on a periodic grid of size $N_0 = 4N$. The periodic problem has, for $doubleT[N_0]$,

$$T[i] - \frac{1}{2}[T[i-1] + T[i+1]] + h^2\frac{m^2}{2}T[i] = \alpha h^2\delta_{i,N} - \alpha h^2\delta_{i,3N+2} \tag{6}$$

defining wrap around conditions: $T[i+1] \equiv T[(i+1)\%N_0]$ and $T[i-1] \equiv T[(i-1+N_0)\%N_0]$ for $i = 0, \cdots, N_0 - 1$. We assume powers of 2 ($N_0 = 2^n$). m^2 is a small number that will not change the solution very much. In your code you can set $h = 1$. Sept we have changed the parameter N for convenience in the next part.

Part I Recursive Multigrid Solver

The exercise is to use a point source into the Laplace problem in both 1D and 2D and solve for the solution using a simple iteration and a Multigrid program. First solve this problem with a simple Jacobi, Gauss Seidel and Red/Black iteration with fixed boundary condition as a test program. Actually you already have this code form HW5. To give you an easy

The problem is to program this with multigrid and compare the iteration number for large N and scaling with respect to N relative to the single level algorithms. To implement Multigrid start on GitHub there are both 1D and 2D multigrid codes with periodic boundary condition but now point sources in `HW6code/MG` and the doubling trick to mimic the fixed boundary conditions. If the Multigrid is really working you can take it almost to zero. (Note in 2D you double twice. Once on each axes. e.g a "quadrupling trick" ?)

.The code is recursive and has 3 basic routines, described in class and the lecture notes. The top level has $h = 1$ and $N_0 = 2^n$ grid points. The problem is to program this with multigrid and compare the iteration number for large N and scaling with respect to N relative to the single level algorithms aboveThis is called level 0. Below it are levels with spacing $2^{level}h$ and $N_0/2^{level}$ grid points. Note that there 3 crucial function:

```
Proj(double *rHat, double r, int level);        // Project level to level +1
Inter(double *error,double *errorHat,int level);// Interpolate level to level -1
Iterate(double *phi_new, *phi, *b , level);     // Iterate Once  on level
```

Coding Exercise #1: Point source with Images

Now for the problem you should slightly modify the MG code provided in both 1D and 2D, In 2D you may simplify the exercise still further with just one point source in the middle and periodic boundary conditions in both x and y directions. (For further reference possibly for you project see Sec 10.2 in class notes for fixed boundary conditions.) The program should stop iterating when each method has reached single precision convergence:

$$\frac{\sqrt{\sum_{i=1}^{2N-1} r[i] * r[i]}}{\sqrt{\sum_{i=1}^{2N-1} b[i] * b[i]}} < 10^{-6} \tag{7}$$

The deliverables for this exercise are:

- At least one source file, `multigrid.cpp`, as described above. Don't forget a Makefile!
- Plots that shows the number of iterations in both your 1D and 2D code for as a function of m^2 for $m^2 = 1, 0.1, 0.01, 0.001$and 0.0001 for Jacobi iterations both with and without multigrid to a small residual like 10^{-6}. The number of grid points should be large so the linear dimension is at least 1024. Vary this until you get some nice plots.
- For the smallest mass for both Jacobi and multigrid plot the residual as function of iteration and comment.

Part II: Slow vs Fast Fourier Transform

The Fast Fourier Transform (FFT) is a classic algorithm very important to analyzing signals of all kind [2]. It was invented by Gauss (who else!) but credited to Cooley and Turkey who rediscovered without knowing Gauss' earlier application: https://en.wikipedia.org/wiki/Fast_Fourier_transform

```
Gauss, Carl Friedrich,"Theoria interpolationis methodo nova tractata",
Werke, Band 3,265327 (Knigliche Gesellschaft der Wissenschaften, Gttingen, 1866)

Cooley, James W.; Tukey, John W. (1965). "An algorithm for the machine
calculation of complex Fourier series". Math. Comput. 19:
297301. doi:10.2307/2003354.
```

This is divide and conquer algorithms doing a specific matrix vector operation in $O(NlogN)$ instead of the straight forward $O(N^2)$. See the class lecture notes for details.

Coding Exercise #2: Program the Complex FFT

The discrete Fourier Transform on N points and its inverse are

$$f(k) \;\; = \sum_{n=0}^{N-1} c_n e^{2\pi i k n/N} \quad , \quad c_n \;\; = \frac{1}{N} \sum_{k=0}^{N-1} f(k) e^{-2\pi i k n/N} \tag{8}$$

respectively. On GitHUB at `!HW6code/FFT!`there is already code for the slow FT. This exercise is simply to convert it into a FFF.

[2]By the way a very similar FFT on Quantum Computers was found that is $O(]logN)$, which is the driving discovery leading to the excitement of faster Quantum Algorithms and the angst about breaking classical encryption methods based on finding prime factors!

The deliverables for this exercise are:

- Run the code `MainFFT.ccp` and verify that the slow (N^2) transforms and its inverse does give the correct result. You can use synthetic data.
- Add to `MainFFT.ccp` the FFT routines and test them against FT.
- Now run speed test of both FT and FFT for large range of sizes $N = 2^n$ with $n = 1, 2, 3 \cdot 10$ at least and plot result to *prove* empirically the scaling of $O(N^2)$ and $O(NlogN)$ respectively

Coding Exercise #3 :Solution to test problem by Fourier Transform

Solve this problem using an Fourier Transform and compare with the previous part. How do we do this? Ok it is convenient now to call the index x as an integer and set h = 1. This is common trick in code. Later you can put back h with $x \to xh$, when you want to revert to the original problem. Ok let's pretend we don't know the solution (this will be true when we go to 2D next). We could try a series in $cos(xn\pi/(2N))$ which satisfy the boundary condition:

$$T[x] = \sum_{k=1}^{N} a_k cos(xk\pi/(2N)) = a_1 \cos(x\pi/(2N)) + a_2 \cos(2x\pi/(2N)) + \cdots \tag{9}$$

An interesting feature of the solution is that the Fourier modes don't like this discontinuous derivative. See Gibbs phenomena and discussion in class: https://en.wikipedia.org/wiki/Gibbs_phenomenon Still the slow modes are the low k- terms as explained in the Lecture.

First we re-write our equation in k-space:

$$(1 - \cos(2\pi k/N_0))\widetilde{T}[k] = \alpha h^2[e^{i2\pi kN/N_0} - e^{-i2\pi kN/N_0}] = 2i\alpha h^2 \sin(2\pi kN/N_0) \tag{10}$$

for for the transformed solution: $\widetilde{T}[k] = \sum_x e^{i2\pi kx/N_0} T[x])$ Then we solve for $T[k]$ and calculate the Fourier transform.

$$T[x] = \frac{1}{N_0} \sum_{k \neq 0} e^{-i2\pi kx/N_0} \widetilde{T}[k] \tag{11}$$

(Why can I drop $k = 0$ although I have no m^2 term? Food for thought?) Do this with a regular"slow" FT and a super FFT. Compare multigrid vs FTT speeds for a range of a values of N_0.

In the lecture notes **Sec. 9.3 Complex Fourier Test Code** , there is a complete slow simple test code for the regular (slow) FT in detail mostly to show how to use complex variables in C. **Just copy it**. Here you only need to turn the FT into a FFT by introducing a recursive call to a function. Also you will want to implement `void makePhase(Complex *omega, int N)` to the setup table of $\omega's$. For debugging you will want to do the inverse too. So add to the test code the functions

```
void FFT(Complex * Ftilde, Complex * F, Complex * omega, int N);
void FFTinv(Complex * F, Complex * Ftilde, Complex * omega, int N);
```

and

```
void FFTrecursive(Complex * Ftilde, Complex * F, Complex * omega, int N);
void FFTrecursive_inv(Complex * F, Complex * Ftilde, Complex * omega, int N);
```

then apply them as set routine to this exercise. Not the template for these routines are in `fft.h`

Next generalize this problem to solve by FFTs the for the 2D example in the multigrid examples with periodic boundary condition in both axes. This is not difficult because you can take Fourier transform on one axis and another one after the other.

The deliverables for this exercise are:

- At least one source file, `FFTsolve.cpp`, that calls another `FFT.cpp` as described above. Don't forget a Makefile!
- For both 1D and 2D make a 2d plot of the solution for both the multigrid solution and the Fourier solution picking a convenient (not too large grid size). The Fourier solutions should be "exact" up to round off. Take the difference of multigrid and Fourier solutions so see if the agree to round off.
- Extra credit make two FFT codes: `FFTiterative .cpp` and `FFTiterative_inv.cpp` and compare time them to compare them. The iterative routines is actually better!

Comments

Why did I do FFT and MG together?

They are both similar recursive divide by 2 methods and therefore data structure map N sites to N/2 is essential the same for both. You can write them once and use them for both FFT and MG. For example in MG: Input double T[N] for $x = 0, 1, 2...N-1$ with $N = 2^p$

$$x = n_0 + 2n_1 + 2^2 n_2 + \cdots + 2^{p-1} n_{p-1} \tag{12}$$

where $p = \log_2(N)$ is the number of bits. (call it "x" or "n" who cares!)

How do we do bit divide an conquer? Project on to N/2 values:

$$\hat{x} = x/2 \quad , \quad \hat{x} = n_1 + 2n_2 + \cdots + 2^{p-1} n_{p-1} \tag{13}$$

Interpolate back to N values:

$$\text{even:} \quad x = 2\hat{x} \quad , \quad \text{odd:} \quad x = 2\hat{x} + 1 \tag{14}$$

So for example in MG the *Projection* routines above you project using

$$\hat{r}[\hat{x}] = (1/2)(r[2\hat{x}] + r[2\hat{x}+1]) \tag{15}$$

and *Interpolate* routine uses,

$$e[2\hat{x}] = e[\hat{x}] \quad , \quad e[2\hat{x}+1] = e[\hat{x}] \tag{16}$$

Same algebra here for FFT needed for

$$y_k = \mathcal{FT}_{N/2}[a_{2n} + \omega_N^k a_{2n+1}] \tag{17}$$

$$y_{k+N/2} = \mathcal{FT}_{N/2}[a_{2n} - \omega_N^k a_{2n+1}] \tag{18}$$

The basic difference is the FFT keep two copies for even/odd as it recurses so it losses no information and in one pass gets the exact transform in $O(NlogN)$ time. The recursive discrete FFT is very much like multigrid. In fact for this case at fixed h, it is **exact** (with infinite precession arithmetic which is impossible of course), but it is not as efficient to a reasonable accuracy. More important the FFT, it can not be generalized to complex geometries and variable conductance. MG converges in $O(N)$ to fixed accuracy. Faster, more stable and more generally applicable.

Possible Extensions for Solver Projects

There are many linear solvers – indeed a vast landscape. Some of these might make ideal examples as an element of your project. One example that you can try as a quick extra credit in this Homework is the venerable *Conjugate Gradient Method*. This is a class of method s that use so called Krylov space. Look at Conjugate Gradient reference below. It easy to program and as extra credit compare its performance with red/black and Multigrid for the 1D Laplacian. If you want an extensive (but readable) set of notes on Conjugate Gradient see on GitHub the file: *painless-conjugate-gradient.pdf*.

Also the Jacobi iteration of the Laplacian as explained in the Lecture notes is the time dependent flow of temperature. By adding fixed or variable temperatures at the boundary or in the interior the 2D iteration is way to study temperature profile on a computer chip. Thus the iteration forms the basis of many related projects on heat flow and scale them up at the SCC with MPI and/or openMP or put them on a GPU. Keep it simple. Pick a particular geometry, boundary conditions, heat production and heat sink and a simple solver choice and parallelization method. You could search the web for input files from actual chips or other 2D surfaces. Incidentally the same equations apply to 2D or 3D laminar (i.e. smooth) fluid flow or electro static potential. **Same equations same solution!** [3]

Some Wikipedia References

1. Wikipedia, Jacobi method
2. Wikipedia, Gauss-Seidel method
3. Wikipedia,Multigrid method
4. Wikipedia,Conjugate Gradient
5. Wikipedia, Thermal management (electronics)

[3]Richard Feynman https://quotefancy.com/quote/1185706/Richard-P-Feynman-The-same-equations-have-the-same-solutions and more delightful quotes https://quotefancy.com/richard-p-feynman-quotes