Final Report

Pierce Hunter, Nick Kuckuck, Haoran Wang

7 June 2020

1 1D Approach

We solved the 1D equation

$$\frac{\partial^2 u}{\partial z^2} = -1; \quad 0 \leq z \leq 1 \tag{1}$$

with boundary conditions

$$\frac{\partial u}{\partial z}(1) = 0; \quad u(0) = 0 \tag{2}$$

in the four following ways:

- Direct solve on the CPU
- CG on the CPU
- Direct solve on the GPU
- CG on the GPU.

For the one-dimensional problem we can utilize the straight-forward centered-difference discretization from class, namely

$$\frac{u_{j-1} - 2u_j + u_{j+1}}{\Delta z^2} = -1 \quad \text{for} \quad 1 \leq j \leq N. \tag{3}$$

We discretize on the boundaries as well, giving the final system of equations

$$\begin{cases} \dfrac{-2u_2 + u_3}{\Delta z^2} = -1 \\ \dfrac{u_{j-1} - 2u_j + u_{j+1}}{\Delta z^2} = -1; \quad 3 \leq j \leq N-1 \\ \dfrac{u_{N-1} - u_N}{\Delta z^2} = -1. \end{cases} \tag{4}$$

We can represent this system of equations as a matrix (A) of the form

$$A = \frac{1}{\Delta z^2} \begin{bmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 1 & -2 & 1 \\ 0 & \cdots & 0 & 1 & -1 \end{bmatrix} \tag{5}$$

with the u-column vector and b-solution vector as

$$u = \begin{bmatrix} u_2 \\ \vdots \\ u_N \end{bmatrix} \qquad b = \begin{bmatrix} -1 \\ \vdots \\ -1 \end{bmatrix} \tag{6}$$

such that $Au = b$.

2 2D Approach

We then expanded the problem to two-dimensions utilizing the same techniques. In 2D eq. (1) becomes

$$\frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = -1; \quad \begin{cases} 0 \leq y \leq 1 \\ 0 \leq z \leq 1 \end{cases} \tag{7}$$

and we expand the boundary conditions in (2) as

$$\frac{\partial u}{\partial y} = 0 \text{ at } y = 1 \tag{8}$$

$$\frac{\partial u}{\partial z} = 0 \text{ at } z = 1 \tag{9}$$

$$u = 0 \text{ at } y = 0 \tag{10}$$

$$u = 0 \text{ at } z = 0. \tag{11}$$

2.1 Discretization

We transform (7) into a system of ODE's by discretizing in space—both y and z—using centered difference, such that

$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta y^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta z^2} = -1. \tag{12}$$

which simplifies when $\Delta y = \Delta z$ to

$$u_{i-1,j} + u_{i,j-1} - 4u_{i,j} + u_{i+1,j} + u_{i,j+1} = -\Delta y^2. \tag{13}$$

This discretization works when $2 \leq i \leq N$ and $2 \leq j \leq N$, but we need to solve on the boundaries, of which there are many. The boundary conditions discretize separately for the edges and corners, which expands our boundary conditions to eight separate cases

- when $i = 2$
  - and $j = 2$

$$\begin{aligned} u_{1,2} + u_{2,1} - 4u_{2,2} + u_{3,2} + u_{2,3} &= -\Delta y^2 \\ -4u_{2,2} + u_{3,2} + u_{2,3} &= -\Delta y^2 \end{aligned}$$

  - and $j = N$

$$u_{1,N} + u_{2,N-1} - 4u_{2,N} + u_{3,N} + u_{2,N+1} = -\Delta y^2$$
$$u_{2,N-1} - 3u_{2,N} + u_{3,N} = -\Delta y^2$$

  - otherwise

$$u_{1,j} + u_{2,j-1} - 4u_{2,j} + u_{3,j} + u_{2,j+1} = -\Delta y^2$$
$$u_{2,j-1} - 4u_{2,j} + u_{3,j} + u_{2,j+1} = -\Delta y^2,$$

- when $i = N$
  - and $j = 2$

$$u_{N-1,2} + u_{2,1} - 4u_{N,2} + u_{N+1,2} + u_{N,3} = -\Delta y^2$$
$$u_{N-1,2} - 3u_{N,2} + u_{N,3} = -\Delta y^2$$

  - and $j = N$

$$u_{N-1,N} + u_{N,N-1} - 4u_{N,N} + u_{N+1,N} + u_{N,N+1} = -\Delta y^2$$
$$u_{N-1,N} + u_{N,N-1} - 2u_{N,N} = -\Delta y^2$$

  - otherwise

$$u_{N-1,j} + u_{N,j-1} - 4u_{N,j} + u_{N+1,j} + u_{N,j+1} = -\Delta y^2$$
$$u_{N-1,j} + u_{N,j-1} - 3u_{N,j} + u_{N,j+1} = -\Delta y^2,$$

- or, when $j = 2$ and $3 \leq i \leq N-1$

$$u_{i-1,2} + u_{i,1} - 4u_{i,2} + u_{i+1,2} + u_{i,3} = -\Delta y^2$$
$$u_{i-1,2} - 4u_{i,2} + u_{i+1,2} + u_{i,3} = -\Delta y^2,$$

- and, lastly when $j = N$ and $3 \leq i \leq N-1$

$$u_{i-1,N} + u_{i,N-1} - 4u_{i,N} + u_{i+1,N} + u_{i,N+1} = -\Delta y^2$$
$$u_{i-1,N} + u_{i,N-1} - 3u_{i,N} + u_{i+1,N} = -\Delta y^2$$

3 Convert to a Matrix

In order to convert this discretization to a matrix that can be used for a direct solve we need to define a new indexing convention. For this we calculate a global index k as

$$k = (i-2)(N-1) + (j-1). \tag{14}$$

We can then translate our discretization into this new system, giving nine total cases. Starting with the corner (2,2) we have

- $(2, 2)$

$$-4u_1 + u_N + u_2 = -\Delta y^2$$

- $(2, j)$ with $3 \leq j \leq N-1$

$$u_{j-2} - 4u_{j-1} + u_{N-2+j} + u_j = -\Delta y^2$$

- $(2, N)$

$$u_{N-2} - 3u_{N-1} + u_{2N-2} = -\Delta y^2$$

- $(i, 2)$ with $3 \leq i \leq N-1$

$$\begin{aligned} &u_{(i-3)(N-1)+1} - 4u_{(i-2)(N-1)+1} \\ &+u_{(i-1)(N-1)+1} + u_{(i-2)(N-1)+2} = -\Delta y^2 \end{aligned}$$

- (i, j) with $3 \leq i \leq N-1$ and $3 \leq j \leq N-1$

$$\begin{aligned} u_{(i-3)(N-1)+j-1} + u_{(i-2)(N-1)+j-2} - 4u_{(i-2)(N-1)+j-1} \\ +u_{(i-1)(N-1)+j-1} + u_{(i-2)(N-1)+j} = -\Delta y^2 \end{aligned}$$

- (i, N) with $3 \leq i \leq N-1$

$$\begin{aligned} u_{(i-3)(N-1)+N-1} + u_{(i-2)(N-1)+N-2} - 3u_{(i-2)(N-1)+N-1} \\ +u_{(i-1)(N-1)+N-1} = -\Delta y^2 \end{aligned}$$

- $(N, 2)$

$$u_{(N-3)(N-1)+1} - 3u_{(N-2)(N-1)+1} + u_{(N-2)(N-1)+2} = -\Delta y^2$$

- (N, j) with $3 \leq j \leq N-1$

$$\begin{aligned} u_{(N-3)(N-1)+j-1} + u_{(N-2)(N-1)+j-2} - 3u_{(N-2)(N-1)+j-1} \\ +u_{(N-2)(N-1)+j} = -\Delta y^2 \end{aligned}$$

- (N, N)

$$u_{(N-2)(N-1)} + u_{(N-2)N} - 2u_{(N-1)^2} = -\Delta y^2$$

So what we end up with is an $(N-1)^2 \times (N-1)^2$ matrix A and a solution vector b with $(N-1)^2$ entries. Moving across a row we start at $(2, 2)$, to increase j by one we move to the right 1 entry, to increase i by 1 we move the right $(N-1)$ entries, such that we hit every value of j first, then move to the next i.

Along the diagonals of the matrix A we have -4 except in the following locations:

- rows $\alpha(N-1)$ the diagonal entry is -3 for $1 \leq \alpha \leq N-2$
- rows $(N-2)(N-1) + \beta$ the diagonal is -3 for $1 \leq \beta \leq N-2$
- row $(N-1)^2$ the diagonal is -2

We also have four other sub-diagonals that will all contain ones except where noted. These represent the following location:

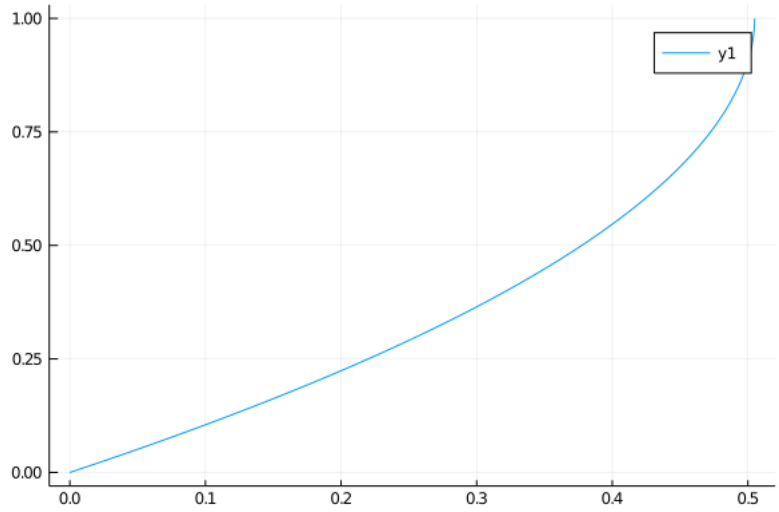
- $j-1$ which is directly below the diagonal
  In Julia these are the locations: `[2:(N-1)^2, 1:(N-1)^2-1]`
  Exception: the locations $(\alpha(N-1)+1, \alpha(N-1))$ should be zero for $1 \leq \alpha \leq N-2$
- $j+1$ which is directly above the diagonal
  In Julia these are the locations: `[1:(N-1)^2-1, 2:(N-1)^2]`
  Exception: the locations $(\alpha(N-1), \alpha(N-1)+1)$ should be zero for $1 \leq \alpha \leq N-2$
- $i-1$ which are exactly $N-1$ below the diagonal
  In Julia these are the locations: `[N:(N-1)^2, 1:(N-1)^2-(N-1)]`
- $i+1$ which are exactly $N+1$ above the diagonal
  In Julia these are the locations: `[1:(N-1)^2-(N-1), N:(N-1)^2]`

Once A is created it is probably easier to create a column vector of length $(N-1)^2$ in which every location contains $-\Delta y^2$. Once a solution is found via $u = A\backslash b$ or CG, then u can be reshaped to the correct dimensions either manually—`i = floor((k-1)/(N-1)) + 2; j = mod(k-1,N-1) + 2`—or via the reshape function transposed—`U = reshape(u,N-1,N-1)'`. Using reshape without the transpose puts the solution in meshgrid format (with y as the columns and z as the rows) similar to looking at a cross-section.

4 Results

4.1 One-Dimensional Problem

Our model solves for the velocity u as a function of depth z for a semi-random nondimensional problem. We plot the solution vector with u on the x-axis and z as the y-axis below.



It is noted that velocity increases as a function of z until reaching a maximum such that the upper boundary condition is satisfied.

We then ensured our solution was converging as expected by decreasing Δz by orders of 2, and checking to make sure the error was decreasing at a constant rate.

Δz	$\varepsilon_{\Delta z} = \sqrt{z}\|u - e\|$	$\Delta\varepsilon = \frac{\varepsilon_{2\Delta z}}{\varepsilon_{\Delta z}}$	$r = \log_2(\Delta\varepsilon)$
0.1	3.102×10^{-2}	Empty Entry	Empty Entry
0.05	1.497×10^{-2}	2.072	1.051
0.025	7.352×10^{-3}	2.036	1.026
0.0125	3.642×10^{-3}	2.019	1.014

We do see the error decreasing linearly with decreases in Δz, which leads us to believe we do not have instability in our system.

We timed the one-dimensional problem using the four methods listed above and depict the results in the table below, keeping Δz constant at 5×10^{-5} throughout.

Device	**Method**	**Time [s]**	**Error** $\sqrt{z}\|u - e\|$
CPU	Direct	0.0426	1.44×10^{-5}
	CG	15.1	1.44×10^{-5}
GPU	Direct	1.13	1.44×10^{-5}
	CG	3.19	1.44×10^{-5}

We note the direct solve on the CPU is the fastest by far. We expect this to be the case as we do not have any time dependence in our model. We do see, when using CG, that the GPU is significantly faster than the CPU, which would be helpful if this problem were altered to include time dependence.

4.2 Two-Dimensional Problem

We ensured our solution was converging as expected by decreasing $\Delta z = \Delta y$ by orders of 2, and checking to make sure the error was still within the machine precision limit. We did this by solving for u and then checking to ensure $Au - b \approx 0$. We show the results for four values of Δz in the table below.

Δz	$Au - b$
0.1	8.02×10^{-16}
0.05	1.8×10^{-15}
0.025	4.6×10^{-15}
0.0125	1.02×10^{-14}

So, we find that for all values of Δz our method does solve the problem within $N\varepsilon$ where N is the length of z and ε is machine precision ($\sim 10^{-16}$). We do expect the error in our solution vector to incease with Δz since we do not have an exact solution on which to compare.

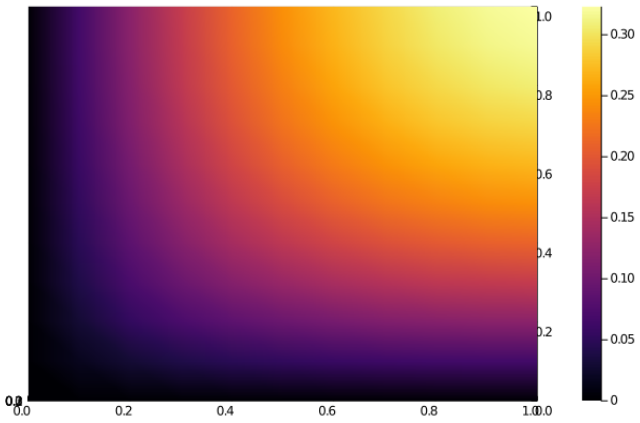
As with the 1D case, we timed the two-dimensional problem using three of the four methods listed above and depict the results in the table below for z = 0.01. Because of difficulties with the CuArrays package in Julia we were not able to solve for the solution directly on the GPU, though we would expect this to be much slower than the direct solve on the CPU, as it was with the 1D case. The results of our timings are depicted in the table below.

Device	**Method**	**Time [s]**	**Error** $Au - b$
CPU	Direct	0.027	1.31×10^{-14}
	CG	0.025	1.49×10^{-10}
GPU	CG	0.2	1.49×10^{-10}

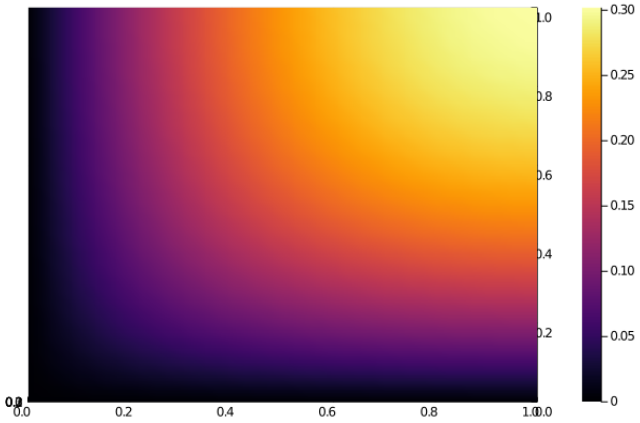
We see the CPU methods are much faster than the GPU methods, but we do expect that if we had been able to run a large enough problem (we were limited by GPU memory) the CG solver on the GPU would have overtaken both methods in terms of both efficiency and time. Unfortunately, in the time remaining we don't really have a way to evidence that claim, it is just our groups expectation that eventually the GPU code would be faster.

We also include a series of plots showing the solution matrix U at different resolutions. The figures appear to be approaching a common solution.

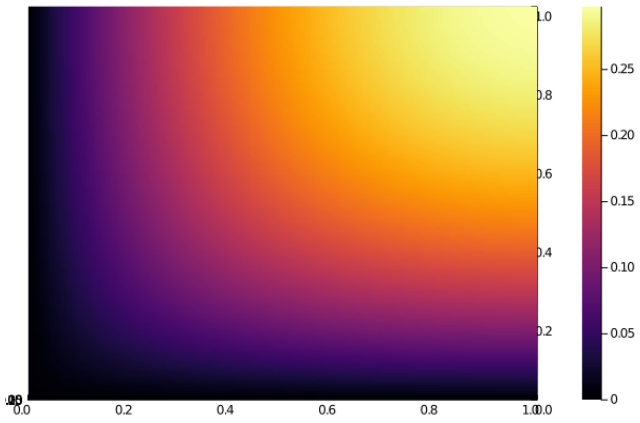
Low resolution case ($\Delta z = 0.1$)



Mid resolution case ($\Delta z = 0.025$)



High resolution case ($\Delta z = 0.01$)



5 Code Listing

Here is the code for our 1D solver:

```
using Pkg
Pkg.add("LinearAlgebra")
Pkg.add("SparseArrays")
Pkg.add("IterativeSolvers")
Pkg.add("Printf")
Pkg.add("CuArrays")
Pkg.add("CUDAnative")
Pkg.add("CUDAdrv")
Pkg.add("Plots")

using LinearAlgebra
using SparseArrays
using IterativeSolvers
using Printf

using CuArrays
CuArrays.allowscalar(false)
using CUDAnative
using CUDAdrv
using Plots

# this shows the diff between CG on CPU and GPU
# Δz = 5e-5

Δz = 0.0125
z = 0:Δz:1
N = length(z)

function exact(z)
    return -1/2 * z.^2 + z
end

# create sparse matrix A
Il = 2:N-1
Jl = 1:N-2
Iu = 1:N-2
Ju = 2:N-1

dl = ones(N-2)
du = ones(N-2)
d = -2*ones(N-1)
A = sparse(Il,Jl,dl,N-1,N-1) + sparse(Iu,Ju,du,N-1,N-1) + sparse(1:N-1,1:N-1,d,N-1,N-1)
A[N-1, N-1] = -1
A .= A / (Δz^2)
```

```
45
46      b = -ones(N-1, 1)
47
48
49      #   Perform LU solve
50      println("Direct solve on CPU")
51      u_dummy_DSCPU = A \ b
52      @time u_int_DSCPU = A \ b
53      u_DSCPU = [0; u_int_DSCPU]
54      # @show umod
55      @printf "norm between our solution and the exact solution = \x1b[31m %e \x1b[0m\n" sqrt(Δz
56      println("-----------")
57      println()
58
59      #    Perform CG solve
60      println("Native Julia CG solve on CPU")
61      u_dummy_CGCPU = cg(-A, -b)
62      @time u_int_CGCPU = cg(-A, -b)
63      u_CGCPU = [0; u_int_CGCPU]
64      @printf "norm between our solution and the exact solution = \x1b[31m %e \x1b[0m\n" sqrt(Δz
65
66      println("-----------")
67      println()
68
69
70      # d_A = CuArrays.CUSPARSE.CuSparseMatrixCSC(A)
71      d_A = CuArray(A)
72      d_b = CuArray(b)
73
74      println("Direct solve on GPU")
75      u_dummy_DSGPU = d_A \ d_b
76      @time u_int_DSGPU = d_A \ d_b
77      u_int_DSGPU_reg = Array(u_int_DSGPU)
78      u_DSGPU = [0; u_int_DSGPU_reg]
79      @printf "norm between our solution and the exact solution = \x1b[31m %e \x1b[0m\n" sqrt(Δz
80      println("-----------")
81      println()
82
83      # CG solve on GPU
84      d_A = CuArrays.CUSPARSE.CuSparseMatrixCSC(A)
85      d_b = CuArray(b)
86      u_cg = CuArray(zeros(size(d_b)))
87      u_dummy = CuArray(zeros(size(d_b)))
88
89      println("CG solve on GPU")
90      # u_dummy_CGCPU = cg(-d_A, -d_b)
91      cg!(u_dummy, d_A, d_b)
92      # @time u_int_CGGPU = cg(-d_A, -d_b)
```

```
93      @time cg!(u_cg, d_A, d_b)
94      u_int_CGGPU_reg = Array(u_cg)
95      u_CGGPU = [0; u_int_CGGPU_reg]
96      @printf "norm between our solution and the exact solution = \x1b[31m %e \x1b[0m\n" sqrt(Δz
97      println("-----------")
98      println()
99
100
101     # plot u against z
102     # u (x axis) z (y axis)
103     # plot(u_DSCPU,z)
104     # png("1D_mid_res")
```

Here is the code for our 2D solver:

```
1
2       using Pkg
3       Pkg.add("LinearAlgebra")
4       Pkg.add("SparseArrays")
5       Pkg.add("IterativeSolvers")
6       Pkg.add("Printf")
7       Pkg.add("CuArrays")
8       Pkg.add("CUDAnative")
9       Pkg.add("CUDAdrv")
10      Pkg.add("Plots")
11
12      using LinearAlgebra
13      using SparseArrays
14      using IterativeSolvers
15      using Printf
16
17      using CuArrays
18      CuArrays.allowscalar(false)
19      using CUDAnative
20      using CUDAdrv
21
22      using Plots
23
24      Δz = 0.01
25      z = 0:Δz:1
26      y = 0:Δz:1
27      N = length(z)
28
29      # create sparse matrix A
30      Imain = 1:(N-1)^2
31      Jmain = 1:(N-1)^2
32
33      Il = 2:(N-1)^2
34      Jl = 1:((N-1)^2-1)
35      Iu = 1:((N-1)^2-1)
36      Ju = 2:(N-1)^2
37
38      il = N:(N-1)^2
39      jl = 1:((N-1)^2-(N-1))
40      iu = 1:((N-1)^2-(N-1))
41      ju = N:(N-1)^2
42
43      dbig = ones((N-1)^2-1)
44      dsmall = ones((N-1)^2-(N-1))
45      dmain = -4*ones((N-1)^2)
46      A = (sparse(Il,Jl,dbig,(N-1)^2,(N-1)^2)
```

```
        + sparse(Iu,Ju,dbig,(N-1)^2,(N-1)^2)
        + sparse(Imain,Jmain,dmain,(N-1)^2,(N-1)^2)
        + sparse(il,jl,dsmall,(N-1)^2,(N-1)^2)
        + sparse(iu,ju,dsmall,(N-1)^2,(N-1)^2))

    # rows α(N-1) the diagonal entry is -3
    for α=1:N-2
        A[α*(N-1),α*(N-1)] = -3
    end

    # rows (N-2)(N-1)+β the diagonal is -3
    for β=1:N-2
        A[(N-2)*(N-1)+β,(N-2)*(N-1)+β] = -3
    end

    # rows (N-1)^2 the diagonal is -2
    A[(N-1)^2,(N-1)^2] = -2

    # locations (α*(n-1)+1, α*(n-1)) should be zero
    # locations (α*(n-1), α*(n-1)+1) should be zero
    for α=1:N-2
        A[α*(N-1)+1,α*(N-1)] = 0
        A[α*(N-1),α*(N-1)+1] = 0
    end

    # print(Matrix(A))
    # print(size(A))

    b = -Δz^2 * ones((N-1)^2, 1)
    # print(b)
    # print(size(b))

    #   Perform LU solve
    println("Direct solve on CPU")
    u_dummy_DSCPU = A \ b
    @time u_int_DSCPU = A \ b
    u_DSCPU = reshape(u_int_DSCPU, (N-1, N-1))
    U_DSCPU = zeros(N,N)
    U_DSCPU[2:N,2:N] = u_DSCPU[:,:]
    #u_DSCPU = transpose(u_DSCPU)
    # print(size(u_DSCPU))
    # @show umod
    @printf "norm between AU and b = \x1b[31m %e \x1b[0m\n" norm(A*u_int_DSCPU - b)
    println("-----------")
    println()


    #    Perform CG solve
```

```
95      println("Native Julia CG solve on CPU")
96      u_dummy_CGCPU = cg(-A, -b)
97      @time u_int_CGCPU = cg(-A, -b)
98      u_CGCPU = reshape(u_int_CGCPU, (N-1, N-1))
99      U_CGCPU = zeros(N,N)
100     U_CGCPU[2:N,2:N] = u_CGCPU[:,:]
101     #u_CGCPU = transpose(u_CGCPU)
102     @printf "norm between AU and b = \x1b[31m %e \x1b[0m\n" norm(A*u_int_CGCPU - b)
103
104     println("-----------")
105     println()
106
107     #=
108     # d_A = CuArrays.CUSPARSE.CuSparseMatrixCSC(A)
109     d_A = CuArray(A)
110     d_b = CuArray(b)
111
112     println("Direct solve on GPU")
113     u_dummy_DSGPU = d_A \ d_b
114     @time u_int_DSGPU = d_A \ d_b
115     u_DSGPU = Matrix(reshape(u_int_DSGPU, (N-1, N-1)))
116     U_DSGPU = zeros(N,N)
117     U_DSGPU[2:N,2:N] = u_DSGPU[:,:]
118     @printf "norm between AU and b = \x1b[31m %e \x1b[0m\n" norm(d_A*u_int_DSGPU - d_b)
119     println("-----------")
120     println()
121     =#
122
123
124
125     # CG solve on GPU
126     d_A = CuArrays.CUSPARSE.CuSparseMatrixCSC(A)
127     d_b = CuArray(b)
128     u_cg = CuArray(zeros(size(d_b)))
129     u_dummy = CuArray(zeros(size(d_b)))
130
131     println("CG solve on GPU")
132     # u_dummy_CGCPU = cg(-d_A, -d_b)
133     cg!(u_dummy, d_A, d_b)
134     # @time u_int_CGGPU = cg(-d_A, -d_b)
135     @time cg!(u_cg, d_A, d_b)
136     u_CGGPU = Matrix(reshape(u_cg, (N-1, N-1)))
137     U_CGGPU = zeros(N,N)
138     U_CGGPU[2:N,2:N] = u_CGGPU[:,:]
139     @printf "norm between AU and b = \x1b[31m %e \x1b[0m\n" norm(d_A*u_cg - d_b)
140     println("-----------")
141     println()
142
```

```
143
144     # u for direct solve on CPU (surface)
145
146     # plot(y,z,U_DSCPU,st=:surface,camera=(0,90))
147     # png("2D_mid_res")
148
149
```