Name: Hongda Li AMATH 585 WINTER 2022 HW5

The Code

The following code is the implementation for both problem 1, 2

```
   keywordstyle
"""
    The cofficient of finite difference of 2D 5 points laplacian without
    the h^2 multiplier and assume equally spaced gridpoints.
"""
function Laplacian5Stencil()
    stencil = Vector()
    push!(stencil, (0, 0, -4))
    push!(stencil, (-1, 0, 1))
    push!(stencil, (0, -1, 1))
    push!(stencil, (1, 0, 1))
    push!(stencil, (0, 1, 1))

return stencil end

"""
    The cofficient of finite difference of 2D 9 points laplacian without
    the h^2 multiplier and assume equally spaced gridpoints.
"""
function Laplacian9Stencil()
    stencil = Vector()
    push!(stencil, (-1, -1, 1/6))
    push!(stencil, (-1, 1, 1/6))
    push!(stencil, (1, -1, 1/6))
    push!(stencil, (1, 1, 1/6))
    push!(stencil, (0, -1, 4/6))
    push!(stencil, (0, 1, 4/6))
    push!(stencil, (1, 0, 4/6))
    push!(stencil, (-1, 0, 4/6))
    push!(stencil, (0, 0, -20/6))

return stencil end

"""
    Make a Linear system representin the Poisson Problem.
    Grid point:
        Equally space on both x and y direction on the domain of [0, 1] x [0, 1]
    Boundary Conditions:
        4 pices of boundary conditions are keyword paremter, should be passed in
        as a scalar function.
    Keyworld Parameters:
        * u_left, u_right, u_top, u_bottom are the function modeling the bouncary
        conditions.
        * f is the righthand side function.
        * stencil: It's a stencil generative function.

"""
function MakeLaplacianSystem(
    m::Int64;
    f=nothing,       # keyword argument
    u_left=nothing,
    u_right=nothing,
    u_top=nothing,
    u_bottom=nothing,
    stencil::Function=Laplacian5Stencil
)
    @assert m > 2 "m should be larger than 2. "
    zeroFunc(x, y) = 0
    returnsMatrixOnly = isnothing(u_top) &&
        isnothing(u_bottom) &&
        isnothing(u_left) &&
```

```
        isnothing(u_right) &&
        isnothing(f)

    # if every keyword argument are nohting, then we just need the
    # laplacian matrix

    u_left = isnothing(u_left) ? (x) -> 0 : u_left
    u_right = isnothing(u_right) ? (x) -> 0 : u_right
    u_top = isnothing(u_top) ? (x) -> 0 : u_top
    u_bottom = isnothing(u_bottom) ? (x) -> 0 : u_bottom
    f = isnothing(f) ? zeroFunc : f

    Tl(i, j) = i + (j - 1)*m
    # make the matrix
    h = 1/(m + 1)
    rhsMod = zeros(m^2)
    RowIdx = Vector{Float64}()
    ColIdx = Vector{Float64}()
    Vals   = Vector{Float64}()

    function AddCoeff(TupleIdx1, TupleIdx2, coef)
        i1, j1 = TupleIdx1
        i2, j2 = TupleIdx2
        # interior
        if i2 >= 1 && i2 <= m && j2 >= 1 && j2 <= m
            push!(RowIdx, Tl(i1, j1))
            push!(ColIdx, Tl(i2,j2))
            push!(Vals, coef)
        # boundary.
        elseif i2 == 0 || i2 == m + 1 # left or right B.C
            if i2 == 0 # left
                rhsMod[Tl(i1, j1)] -= u_left(j2*h)*coef
            else      # right
                rhsMod[Tl(i1, j1)] -= u_right(j2*h)*coef
            end
        elseif j2 == 0 || j2 == m + 1 # top or bottom
            if j2 == 0 # bottom
                rhsMod[Tl(i1, j1)] -= u_bottom(i2*h)*coef
            else       # top
                rhsMod[Tl(i1, j1)] -= u_top(i2*h)*coef
            end
        else
            error("this should not happen, I don't expect this")
        end
    return end

    # Construct using Stencil
    stencil = stencil()
    for i in 1:m, j in 1:m
        for (k, l, v) in stencil
            AddCoeff((i, j), (i + k, j + l), v/h^2)
        end
        rhsMod[Tl(i, j)] += f(i*h, j*h)
    end

    if returnsMatrixOnly
        return sparse(RowIdx, ColIdx, Vals)
    end

return sparse(RowIdx, ColIdx, Vals), rhsMod end


"""
    Return the interior points of a partitioning of the grid points
    in a vector of tuples of x, y if the parameter u is not given.
    if it's given then it will just put these tuples of values into the
    given u function, values are returned as a vector of natural ordering.
```

```
    parameter:
        m:: An integer larger than 2.
"""
function Grid2Vec(m::Int64, u=nothing)
    v = Vector{Float64}()
    h = 1/(1 + m)
    if isnothing(u)
        for i in 1:m, j in 1:m
            push!(v, (i*h, j*h))
        end
        return v
    end

    if typeof(u) <: Function
        for i in 1:m, j in 1:m
            push!(v, u(i*h, j*h))
        end
        return v
    end

    if typeof(u) <: AbstractMatrix
        for i in 1:m, j in 1:m
            push!(v, u[j, i])
        end
        return v
    end

return v end


"""
    convert the naturally ordered vector into a grid, including the
    boundary conditions.

    * Returns a matrix that is literally the function values over the grid points.
    the (i,j) element is (j*h, i*h), so it's the 2d quadrant rotated to the right
    by 180

    * Plotting of the matrix using heatmap will recover the function's plot
    over [0, 1] x [0, 1] like, exactly as what you expected because
    julia plots it upside down.

"""
function Vec2Grid(
    m, v;
    bc_left=nothing,
    bc_right=nothing,
    bc_top=nothing,
    bc_bottom=nothing
)
    g = zeros(m, m)
    for i in 1:m, j in 1:m
        g[j, i] = v[(j - 1)*m + i]
    end
    if isnothing(bc_left) && isnothing(bc_right) && isnothing(bc_top) && isnothing(bc_bottom)
       return g
    end
    @error("Not yet implemented.")
    gWithBC = zeros(m + 2, m + 2)
    gWithBC[2:end - 1, 2:end - 1] = g

return gWithBC end


"""
    Performs conjugate gradient to solve the sparse linear effectively. The implementation of
        this method
    is faster than writting out an FFT based Poisson solver.
```

```
"""
function SimpleConjugateGradient(A::AbstractMatrix, b::AbstractVector; epsilon=1e-10)
    error("Not Yet implemented yet. ")
return end


### Hands on testing part ------------------------------------------------------

using SparseArrays, Plots, LinearAlgebra, Statistics

"""
    Basic test.
"""
function BasicTests()
    f = (x, y) -> x^2 + y^2
    m = 3
    M, b = MakeLaplacianSystem(
        3,
        u_right=(x)->1, u_top=(x) -> 1, u_bottom =(x)-> 1, u_left=(x) ->1,
        stencil=Laplacian9Stencil
    )
    @info "Carrying out some basic testing: "
    (M*6)/(m + 1)^2 |> display
    b*6/(m + 1)^2 |> display
    Vec2Grid(m, Grid2Vec(m, (x, y)-> x + y)) |> heatmap |> display

    @info "Testing whether the 9 points Laplacian works"
    m = 63
    A, b = MakeLaplacianSystem(
        m, f=f,
        u_right=(x)->1, u_top=(x) -> 1, u_bottom =(x)-> 1, u_left=(x) ->1,
        stencil=Laplacian9Stencil
    )
    println("M matrix: ")
    Vec2Grid(
        m,
        A\b,
    ) |> heatmap |> display

return end



function Problem1()
    # super fine grid point solution come first.
    M = 2^10 - 1
    f(x, y) = x^2 + y^2
    u_bc(x) = 1
    A, b = MakeLaplacianSystem(
        M,
        f=f,
        u_left=u_bc,
        u_right=u_bc,
        u_top=u_bc,
        u_bottom=u_bc
    )

    @info "The system we are solving is: "
    A |> display
    b |> display
    uVeryFine = Vec2Grid(M, A\b)
    uVeryFine |> heatmap |> display

    # estimating the error using super fine grid as a reference
    Errors = Vector{Float64}()       # L2 error over the grid
    gridWidth = Vector{Float64}()
    for m in 2 .^ (collect(2:8)) .- 1
        h = 1/(m + 1)
```

```
        A, b = MakeLaplacianSystem(
            m,
            f=f,
            u_left=u_bc, u_right=u_bc, u_top=u_bc, u_bottom=u_bc
        )
        uCoarse = Vec2Grid(m, A\b)
        skip = convert(Int64, (M + 1)/(m + 1) |> floor)
        push!(Errors,
            h*norm(
                (uVeryFine[skip:skip:end, skip:skip:end] - uCoarse)[:]
            )
        )
        push!(gridWidth, h)
    end
    # Printing things out:
    @info "These are a print out for the error vectors and gird width. "
    Errors |> display
    gridWidth |> display
    gridWidthLogged = log2.(gridWidth); ErrorsLogged = log2.(Errors)
    fig = plot(
        gridWidthLogged,
        ErrorsLogged,
        title="Log2 vs Log2 Error",
        label="5 p stencil",
        legend=:bottomright
    )
    plot!(fig, gridWidthLogged, gridWidth.^2 .|> log2, label="reference h^2")
    xlabel!(fig, "log2(h)")
    ylabel!(fig, "log2(E)")
    fig |> display
    savefig(fig, "p1_fig.png")

    # Estimating rate of convergence.
    println("The estimated rate of convergence for the first problem is: ")
    (ErrorsLogged[1:end - 1] - ErrorsLogged[2:end])./
    (gridWidthLogged[1:end - 1] - gridWidthLogged[2:end])|> mean |> display

return end


function Problem2()
    function SolveWithDeferredCorrection(m::Int64)
        # super fine grid point solution come first.
        f(x, y) = x^2 + y^2
        u_bc(x) = 1
        A, b = MakeLaplacianSystem(
            m,
            f=f,
            u_left=u_bc,
            u_right=u_bc, u_top=u_bc, u_bottom=u_bc,
            stencil=Laplacian9Stencil
        )
        b .+= 1/(3*(m + 1)^2)
    return Vec2Grid(m, A\b) end
    uVeryFine = SolveWithDeferredCorrection(2^10 - 1)
    @info "uVeryFine 9 points stencils solution looks like: "
    uVeryFine |> display

    # estimating the error using super fine grid as a reference
    Errors = Vector{Float64}()       # L2 error over the grid
    gridWidth = Vector{Float64}()
    M = 2^10 - 1
    for m in 2 .^ (collect(2:8)) .- 1
        h = 1/(m + 1)
        uCoarse = SolveWithDeferredCorrection(m)
        skip = convert(Int64, (M + 1)/(m + 1) |> floor)
        push!(Errors,
            h*norm(
```

```
                (uVeryFine[skip:skip:end, skip:skip:end] - uCoarse)[:]
            )
        )
        push!(gridWidth, h)
    end

    # Printing things out:
    @info "These are a print out for the error vectors and gird width. "
    Errors |> display
    gridWidth |> display
    gridWidthLogged = gridWidth .|> log2
    ErrorsLogged = Errors .|> log2
    fig = plot(
        gridWidthLogged,
        ErrorsLogged,
        title="Log2 vs Log2 Error",
        label="9 p stencil",
        legend=:bottomright
    )
    plot!(
        fig,
        gridWidthLogged,
        gridWidth.^3 .|> log2,
        label="reference h^3"
    )
    xlabel!(fig, "log2(h)")
    ylabel!(fig, "log2(E)")
    fig |> display
    savefig(fig, "p2_fig.png")
    # estimate the convergence numerically.
    println("The estimated rate of convergence for the second problem is: ")
    (ErrorsLogged[1:end - 1] - ErrorsLogged[2:end])./
    (gridWidthLogged[1:end - 1] - gridWidthLogged[2:end])|> mean |> display
end

BasicTests()
Problem1()
Problem2()
```

Problem 1

Theory that goes with the code

Natural ordering of the element refers to the mapping of tuple of indices from the grid to the actual point in space, given as $u_{i,j} = u(ih, jh)$. Here, we use equally space point in both xy direction on the unit square: $[0,1] \times [0,1]$. And we use m number of interior points, and including the boundary we will have $m+2$ boundary point.

The implementation focuses on building up the system of matrices using the stencil defined on each point indexed grid point (i,j). For each point, we convert the coordinate indices (i,j) to linear indices for row and column location in the sparse matrix, the coefficient on the stencil is then added to the matrix or the RHS of the system. This is done for $(i,j) \in \{0,1,\cdots,m\}^2$. If any of the point are laying outside, we move the coefficient to the RHS vector.

Alternatively, one can also use Kronecker Product to construct the structural matrice, but it will lose the elegence of on the code and less generic. Algebraically, one can handle the boundary conditions by thinking about the stencils on the boundary. I only did it for the 5 point stencil.

Let $\mathcal{I}(i,j) = (j-1)m + i \quad \forall (i,j) \in [1,\cdots,m] \times [1,\cdots m]$ be a mapping from the coordinate indices of the grid point to linear index. Next, we consider partitioning the vector u and b the RHS vector by rows on

the grid using natural ordering.

$$u = \begin{bmatrix} u^{[1]} \\ u^{[2]} \\ \vdots \\ u^{[m]} \end{bmatrix} \quad \vec{b} = \begin{bmatrix} b^{[1]} \\ b^{[2]} \\ \vdots \\ b^{[m]} \end{bmatrix} \quad \vec{f} = \begin{bmatrix} f^{[1]} \\ f^{[2]} \\ \vdots \\ f^{[m]} \end{bmatrix} \tag{1.1}$$

$$u^{[i]} = \begin{bmatrix} u_{1,i} \\ u_{2,i} \\ \vdots \\ u_{m,i} \end{bmatrix}$$

For the bounary conditions, we consider them row by row. Consider the first row:

$$\begin{aligned} \vec{f}_{\mathcal{I}(i,1)} &= \frac{1}{h^2}\text{sum}\left(\begin{bmatrix} & u_{i,2} & \\ u_{i-1,1} & -4u_{i,1} & u_{i+1,1} \\ & u_{i,0} & \end{bmatrix}\right) \quad \forall 1 \le i \le m \\ \vec{b}^{[1]} &= -\frac{1}{h^2}\begin{bmatrix} u_{1,0} \\ u_{2,0} \\ \vdots \\ u_{m,0} \end{bmatrix} - \frac{u_{0,1}\mathbf{e}_1^{(m)}}{h^2} - \frac{u_{m+1,1}\mathbf{e}_m^{(m)}}{h^2} + f^{[1]} \end{aligned} \tag{1.2}$$

Next, we consider all the rows in the middle rows:

$$\begin{aligned} \vec{f}_{\mathcal{I}(i,j)} &= \frac{1}{h^2}\text{sum}\left(\begin{bmatrix} & u_{i,j+1} & \\ u_{i-1,j} & -4u_{i,j} & u_{i+1,j} \\ & u_{i,j-1} & \end{bmatrix}\right) \quad \forall 1 \le i \le m,\ 2 \le j \le m-1 \\ \vec{b}^{[j]} &= -\frac{u_{0,j}\mathbf{e}_1^{(m)}}{h^2} - \frac{u_{m+1,j}\mathbf{e}_m^{(m)}}{h^2} + f^{[j]} \end{aligned} \tag{1.3}$$

Finally we consider the last row on the grid point:

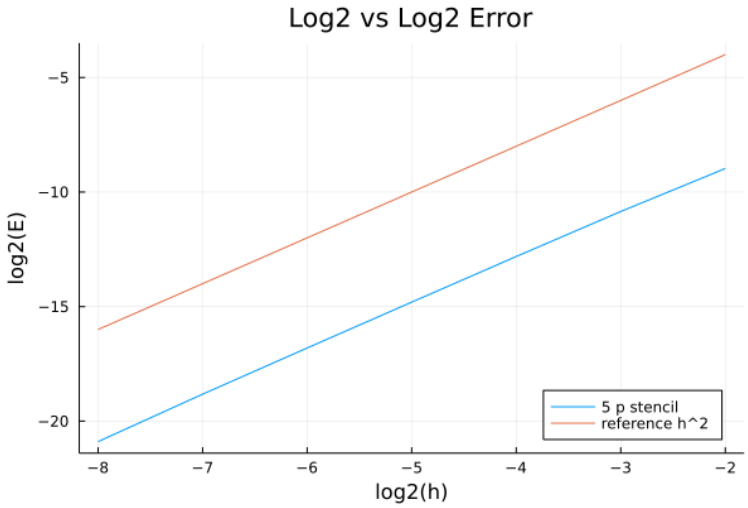
$$\begin{aligned} \vec{f}_{\mathcal{I}(i,m)} &= \frac{1}{h^2}\text{sum}\left(\begin{bmatrix} & u_{i,m+1} & \\ u_{i-1,j} & -4u_{i,m} & u_{i+1,j} \\ & u_{i,m-1} & \end{bmatrix}\right) \quad \forall\ 1 \le i \le m \\ b^{[m]} &= -\frac{1}{h^2}\begin{bmatrix} u_{1,m+1} \\ u_{2,m_1} \\ \vdots \\ u_{m,m+1} \end{bmatrix} - \frac{u_{0,m}}{h^2}\mathbf{e}_1^{(m)} - \frac{u_{m+1,m}}{h^2}\mathbf{e}_m^{(m)} + \vec{f}^{[m]} \end{aligned} \tag{1.4}$$

This completes the analysis for the RHS vector that includes the boundary. Take notice that a very different approach has been coded. This part is just for analysis.

$\mathcal{O}(h^2)$ Convergence Rate

Error computations is made by comparing solution on sparser grid point with the solution obtained on a fine grid point. I don't happen to know the solution of the PDE besides the solution expressed using the Green's function with the given density function.

For the implementation of this part, refers to the "Problem1()" function. The finer grid is taken with $m = 2^{10} - 1$, and then m follows the pattern of $2^k - 1$ so that the discretized grid-point of the sparser grids is a subset of the finer grid. This is a plot of the error measured by $L2$ norm in 2D in log log plot, referenced with h^2 convergence:

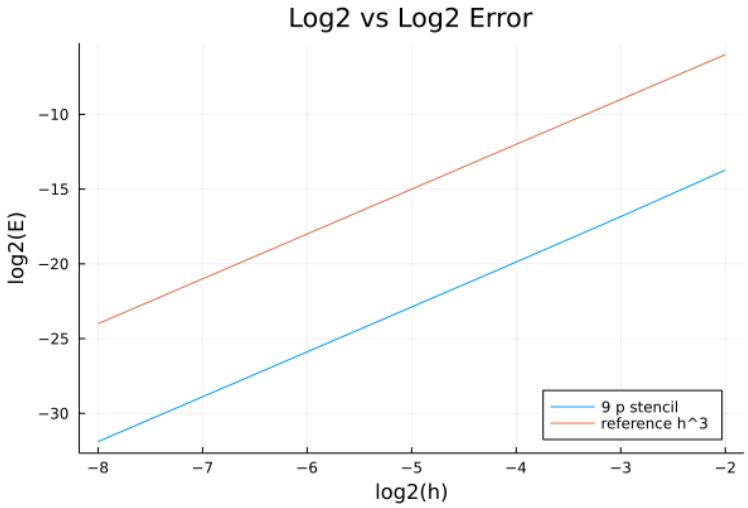


Numerically, I take the slope of the adjacent points on the log log plot and take the averaging, producing an estimate of the rate of convergence for the 5 points stencil method. The esimated rate is: 1.986773401321141.

Problem 2

The same routine is run for the 9-point stencils but with a different definition of the stencil. Deferred correction is implemented by the end, right before the calculation of solution.

The estimated rate of convergnece of error is 3.0268502045280115, The error is $\mathcal{O}(n^3)$, which differs from the theory. Here is a plot of the error under log log plot:



Observe that this is a mismatch from what is predicted by the theory where it's stated that with deferred correction, the error should have convergence rate of $\mathcal{O}(h^4)$.