Notebook: Geometric multigrid for 2D-PDE's

Solving Partial Differential Equations has never been an easy task on mathematicians. In fact, in many cases it's even considered *impossible* to do with our current mathematics, so our best way to approach it is with numerical/computational approximations. It's this exact field that has specialized in solving PDE's numerically: *Using Iterative methods, Fourrier Transforms, Finite Differences, etc.*.

This project is about the **Multigrid methods** which are one of the most interesting assets of the numerical computing toolbox. We're also going to do some benchmarking of their performance regarding their iterative counterparts.

Finally, to be able to start, we should first do the initial step of the numerical problem resolution framework: **Discretizing the PDE space**

```
# Used Packages Index
begin
    using LinearAlgebra      # Linear Algebra
    using SparseArrays       # Sparse Arrays Optimization
    using Plots              # Plotting & Visualization
end ;
```

Constructing the Laplacian 2D-operator and the Problems PDEs

$$Au = \Delta u = \frac{\partial^2 u}{\partial x^2}\, e_x + \frac{\partial^2 u}{\partial y^2}\, e_y$$

To construct the laplacian operator, we're using the **Kronecker product** with the identity matrix to develop the axe-wise differentiation operators and sum them up lately to obtain Δ.

$$\begin{bmatrix} \boldsymbol{A}\otimes\boldsymbol{B}_{11} & \cdots & \boldsymbol{A}\otimes\boldsymbol{B}_{1m} \\ \vdots & \ddots & \vdots \\ \boldsymbol{A}\otimes\boldsymbol{B}_{m1} & \cdots & \boldsymbol{A}\otimes\boldsymbol{B}_{mm} \end{bmatrix}$$

$$\begin{bmatrix} -4 & 1 & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & -4 & 1 & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & -4 & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ 1 & \cdot & \cdot & -4 & 1 & \cdot & 1 & \cdot & \cdot \\ \cdot & 1 & \cdot & 1 & -4 & 1 & \cdot & 1 & \cdot \\ \cdot & \cdot & 1 & \cdot & 1 & -4 & \cdot & \cdot & 1 \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & -4 & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & 1 & -4 & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & 1 & -4 \end{bmatrix}$$

Δ Operator Matrix

Regarding the Kronecker product, it is actually a **computationally expensive** operation, that is why we have to use **sparse matrices** for this extent. The use of sparse matrices is further supported by the sparsity factor of our Laplacian operator that isn't to be ignored.

We then need to transform our differential equation into a linear problem in the form of $\mathbf{Au=f}$. For that, we only have to use the matrix σI to then factorize the equation by $u(x,y)$.

$$\begin{gathered} -\Delta u(x,y) + \sigma u(x,y) = f(x,y) \quad (P_1) \\ (\sigma I - \Delta)u(x,y) = f(x,y) \\ A_1 = \sigma I - \Delta \end{gathered}$$

As for the Anisotropic problem P_2, the construction is similar but only using the partial derivatives

$$\begin{gathered} -\frac{\partial^2 u(x,y)}{\partial x^2} - \epsilon \frac{\partial^2 u(x,y)}{\partial y^2} = f(x,y) \quad (P_2) \\ A_2 = (\frac{\partial^2}{\partial x^2} - \epsilon \frac{\partial^2}{\partial y^2})I \end{gathered}$$

```
begin
    function A₁(n::Int, σ::Float64)
        ∂²  = Tridiagonal(ones(n-1), -2 * ones(n), ones(n-1))
        ∂x² = (n^2) * kron(sparse(∂²), I(n))    # kron is the Kronecker product
        ∂y² = (n^2) * kron(I(n), sparse(∂²))    # h is the unit of displacement
        Δ   = ∂x² + ∂y²
        return σ * I(n^2) - Δ
    end

    function A₂(n::Int, ϵ::Float64)
        ∂²  = Tridiagonal(ones(n-1), -2 * ones(n), ones(n-1))
        ∂x² = (n^2) * kron(sparse(∂²), I(n))    # kron is the Kronecker product
        ∂y² = (n^2) * kron(I(n), sparse(∂²))    # h is the unit of displacement
        return - ∂x² - ϵ * ∂y²
    end

    A₁(σ::Float64) = n -> A₁(n, σ)
    A₂(ϵ::Float64) = n -> A₂(n, ϵ)
end ;
```

Before jumping to start modelling the solvers, we need first to iterate on the boundaries conditions. The PDE's we're solving are both using the same boundaries conditions $u(x,y) = 0$ on $\partial\Omega$ so we're just coding them in a function that will be called after each solver iteration to reinject the boundaries so that we don't miss the track of them.

```
function boundaries(v)
    n = Int(sqrt(size(v,1)))
    gridᵥ = reshape(v, (n, n))'
    gridᵥ[1,:] .= 0
    gridᵥ[end,:] .= 0
    gridᵥ[:,1] .= 0
    gridᵥ[:,end] .= 0
    return gridᵥ |> transpose |> vec
end ;
```

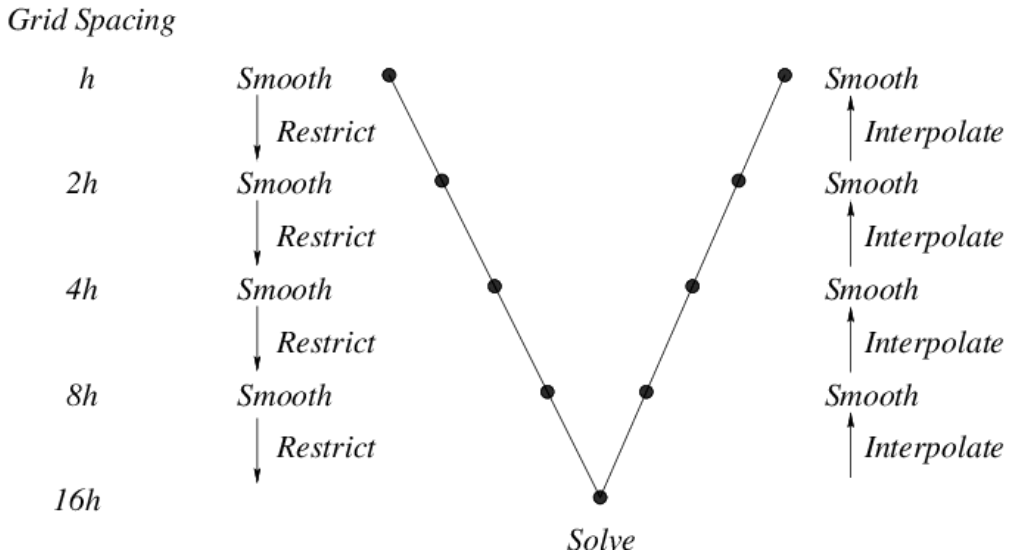
Constructing the solvers algorithms

For the stationary smoothers, we're each time choosing one of the defined functions below either to be used on its own for comparison or to use it as the smoother in our Multigrid implementation.

```
function Jacobi(A, b, u₀ = zeros(size(A, 1)), ϵ = 1e-7, maxiter = 10, bounds = false)
    u = u₀; n = Int(sqrt(size(A, 1))); iter = 0
    M = Diagonal(A)
    N = UnitLowerTriangular(A) + UnitUpperTriangular(A) - 2*I(n^2)
    while iter <= maxiter
        iter += 1
        if bounds
            u = boundaries(u)
        end
        u = inv(M) * (N*u + b)
        (norm(b - A*u, 2) > ϵ) || break
    end
    return u
end ;
```

```
function JOR(A, b, ω, u₀ = zeros(size(A, 1)), ϵ = 1e-7, maxiter = 10, bounds = false)
    u = u₀; iter = 0
    M = Diagonal(A) / ω
    while iter <= maxiter
        iter += 1
        if bounds
            u = boundaries(u)
        end
        r = b - A*u
        z = inv(M) * r
        u += z
        (norm(r, 2) > ϵ) || break
    end
    return u
end ;
```

```
function SOR(A, b, ω, u₀ = zeros(size(A, 1)), ϵ = 1e-7, maxiter = 10, bounds = false)
    u = u₀; n = Int(sqrt(size(A, 1))); iter = 0
    D = Diagonal(A)
    L = UnitLowerTriangular(A) - I(n^2)
    U = UnitUpperTriangular(A) - I(n^2)
    while iter <= maxiter
        iter += 1
        if bounds
            u = boundaries(u)
        end
        u = inv(D + ω * L) * (ω * b - (ω * U + (ω-1) * D) * u)
        (norm(b - A*u, 2) > ϵ) || break
    end
    return u
end ;
```



As shown in the simple form of grid-spacing in a **V-cycle** multigrid, we need some **interpolation** operators to be used to move from one grid to another:

- *Restriction*: From a *fine* grid to a *coarser* one
- *Prolongation*: From a *coarse* grid to a *finer* one

Therefore, the methods defined below will be used interchangeably.

Also, as we're treating 2 different problems:

- *isotropic*: $A_1 u(x, y) = f(x, y)$
- *ϵ-anisotropic*: $A_2 u(x, y) = f(x, y)$

That's why we're going to need two different strategies for grids interpolation as we're not moving in the same space for both problems.

```
# Multigrid's **Isotropic** Interpolation operators
begin
    # Restriction
    injection(grid) = grid[2:2:end, 2:2:end]

    function halfweight(grid)
        g = Float64.(grid)
        for i=2:2:size(grid,1)-1, j=2:2:size(grid,2)-1
            g[i,j] = g[i,j] / 2 - (
                g[i-1,j] + g[i+1,j]
                + g[i,j-1] + g[i,j+1]) / 8
        end
        return injection(g)
    end

    function fullweight(grid)
        g = Float64.(grid)
        for i=2:2:size(grid,1)-1, j=2:2:size(grid,2)-1
            g[i,j] = g[i,j] / 4 - (
                g[i-1,j] + g[i+1,j]
                + g[i,j-1] + g[i,j+1]) / 8 - (
                g[i-1,j-1] + g[i+1,j+1]
                + g[i-1,j+1] + g[i+1,j-1]) / 16
        end
        return injection(g)
    end
```

```
    # Prolongation
    function enlarge(grid)
        n = size(grid,1) * 2
        n == 2 && return repeat(grid, n, n)
        g = zeros((n,n))
        for i=2:n-1, j=2:n-1
            g[i, j] = grid[i÷2, j÷2]
        end
        return g
    end

    function linearize(grid)
        n = size(grid,1) * 2
        n == 2 && return repeat(grid, n, n)
        g = zeros((n,n))
        for i=2:n-1, j=2:n-1
            g[i, j] = (grid[Int(floor((i+1)/2)), Int(floor((j+1)/2))]
                       + grid[Int(ceil((i+1)/2)), Int(floor((j+1)/2))]
                       + grid[Int(floor((i+1)/2)), Int(ceil((j+1)/2))]
                       + grid[Int(ceil((i+1)/2)), Int(ceil((j+1)/2))]) / 4
        end
        return g
    end
end ;
```

Algorithm 1: V-cycle multigrid iteration

$w_h := \mathrm{MG}(h, L_h, w_h^0, b_h)$

IF $(h == h_0)$
    Solve exactly: $w_h = L_h^{-1} b_h$
ELSE
    Pre-smooth ν_1 times: $w_h := \mathrm{SMOOTH}^{\nu_1}(L_h, w_h^0, b_h)$
    Restriction: $r_H := I_h^H (b_h - L_h w_h)$
    Initialize correction: $\delta_H := 0$
    Recursion : $\delta_H := \mathrm{MG}(H, L_H, \delta_H, r_H)$
    Prolongation: $\delta_h := I_H^h \delta_H$
    Correction: $w_h := w_h + \delta_h$
    Post-smooth ν_2 times: $w_h := \mathrm{SMOOTH}^{\nu_2}(L_h, w_h, b_h)$
ENDIF
RETURN w_h.

```
"""
    multigrid(A, b, u, l, ω, ϵ, steps,
              restrict, prolong, iter)

l-level Multigrid cycle. Each sub-level dive is held with x10 smoothing iterations.
    - A: the PDE's linear operator constructor
    - restrict: function for grid restriction
    - prolong: function for grid prolongation
    - iter: number of iterations of **current** grid smoothing
    - steps: number of multigrid callbacks per cycle:
        - 1 (default) is for V-cycle
        - 2 is for normal W-cycle
        - n (> 2) is for W-cycles with n sub-refinements
"""
function multigrid(A, b, u, l, ω, ϵ=1e-7, steps=1,
        restrict=injection, prolong=enlarge, iter=10)
    n  = Int(sqrt(size(b, 1)))
    Aₙ = A(n)
    if l == 0
        # We can also use a direct solver instead
```

```
        # u = Array(Aₙ) \ b
        u = JOR(Aₙ, b, ω, u, ϵ, iter)           # Resolution
    else
        u = JOR(Aₙ, b, ω, u, ϵ, iter)           # Pre-smoothing

        # Defect restriction
        r = reshape(b - Aₙ*u, (n, n)) |> transpose |>
            restrict |> transpose |> vec

        # Coarse-level Correction
        δᵣ = zeros(size(r))
        for i=1:steps
            δᵣ = multigrid(A, r, δᵣ, l-1, ω, ϵ,
                    steps, restrict, prolong, iter*3)
        end

        # Defect Prolongation δᵣ → δ
        δ = reshape(δᵣ, (n÷2, n÷2)) |> transpose |>
            prolong |> transpose |> vec

        u += δ                                  # Correction
        u = JOR(Aₙ, b, ω, u, ϵ, iter)           # Post-smoothing
    end
    return u
end ;
```

Simulation and Results Interpretation

We start by running our *isotropic* problem resolution governed by the A_1 operator. We're also only interested in the *unit square* resolution of the function f, and for it we discretize the space on $n = 1024$ units, $h = 1/n$ being the displacement unit.

```
begin
    n = 1024
    h = 1 / n
    ω = 0.95
    σ = 0.7
    A = A₁(σ)
    u = zeros(n^2)
    f = 3
    b = [sin(2π*f*i*j) for i=0:h:1-h for j=0:h:1-h]
end ;
```

```
9.14521813
```

```
# Jacobi Over Relaxation
@elapsed u₁ = JOR(A(n), b, ω, u, 1e-30, 50, true)
```

```
13.729009692
```

```
# Multigrid V-cycle
@elapsed u₂ = multigrid(A, b, u, 3, ω, 1e-30, 1, injection, linearize)
```

Examining the cells above, we can notice that both solvers took a similar time to finish their iterations. If we then thoroughly examine the written multigrid code, we'll find out that for the current simulation, while it yielded nearly the **same execution-time** as the Jacobi-Over-Relaxation method, we're actually running a *3-level V-cycle multigrid* that computes over **800 iterations** and that also have given a by-far **better approximation** than its iterative counterpart even when running most of those iterations on the *coarser* grid 128x128 instead of the original 1024x1024 one.

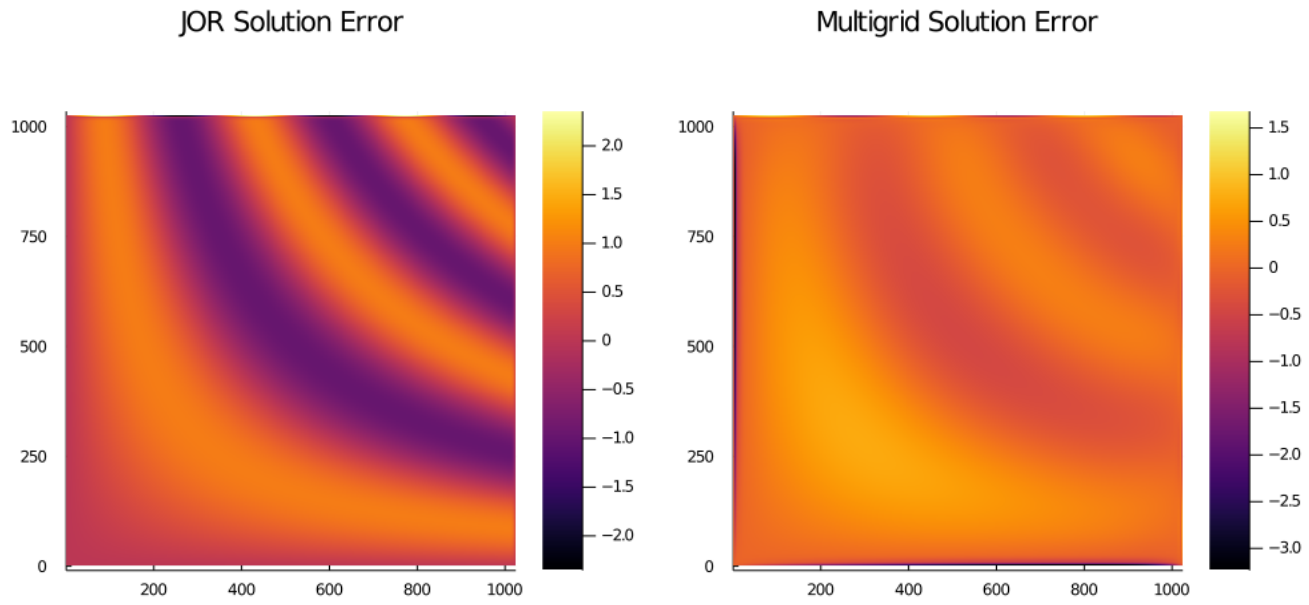
```
"||eₘ|| < ||eⱼ||"
```

```
"Error norm gain: 318.32821"
```

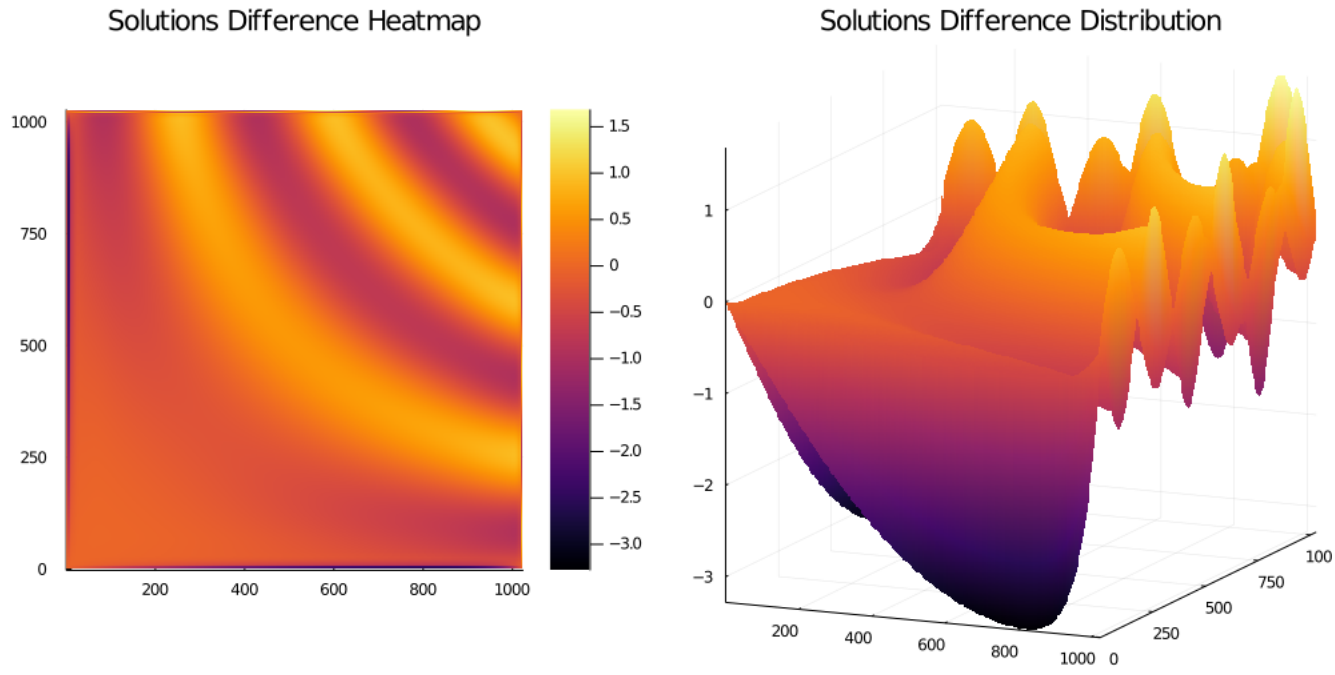
The near-similarity of the execution-time between both solvers is actually due to the fact that the 300 iterations ran by the Multigrid are equivalent to only 48 ones of the Jacobi-Over-Relaxation method.

```
"Multigrid JOR-relative cost: 48.359375 iterations"
```

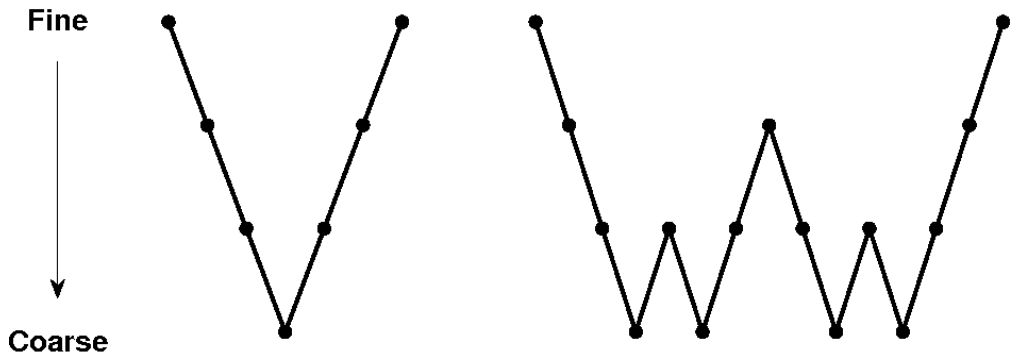
Solutions Visualization & interpretation



From the graphs shown above and below, we can tell that the multigrid's solution is by far a better solution than the iterative JOR solver *(Multigrid's being dimmer and softer which by the graph's legend means a* ***nearer to 0 error****)*. Also, the solutions difference distribution tells us even more about the struggle that Jacobi-Over-Relaxation method had trying to polish the *lowest frequencies* in only 50 iterations.



V-cycle vs W-cycle



(a) Multigrid V-Cycle (b) Multigrid W-Cycle

We'll now proceed to compare the convergence of the V-cycle and the W-cycle variants of the multigrid solver for various σ and n levels for the problem operator $A_1(\sigma, n)$.

1- $n = 1024\ ,\ \sigma = 0.7$

```
begin
    σ₁  = 0.7
    n₁  = 1024
    h₁  = 1 / n₁
    uₙ₁ = zeros(n₁^2)
    bₙ₁ = [sin(2π*f*i*j) for i=0:h₁:1-h₁ for j=0:h₁:1-h₁]
    A₁₁ = A₁(σ₁)
end ;
```

```
11.645717189
```

```
# Multigrid V-cycle
@elapsed uᵢ₁ = multigrid(A₁₁, bₙ₁, uₙ₁, 3, ω, 1e-30, 1)
```

```
26.908595232
```

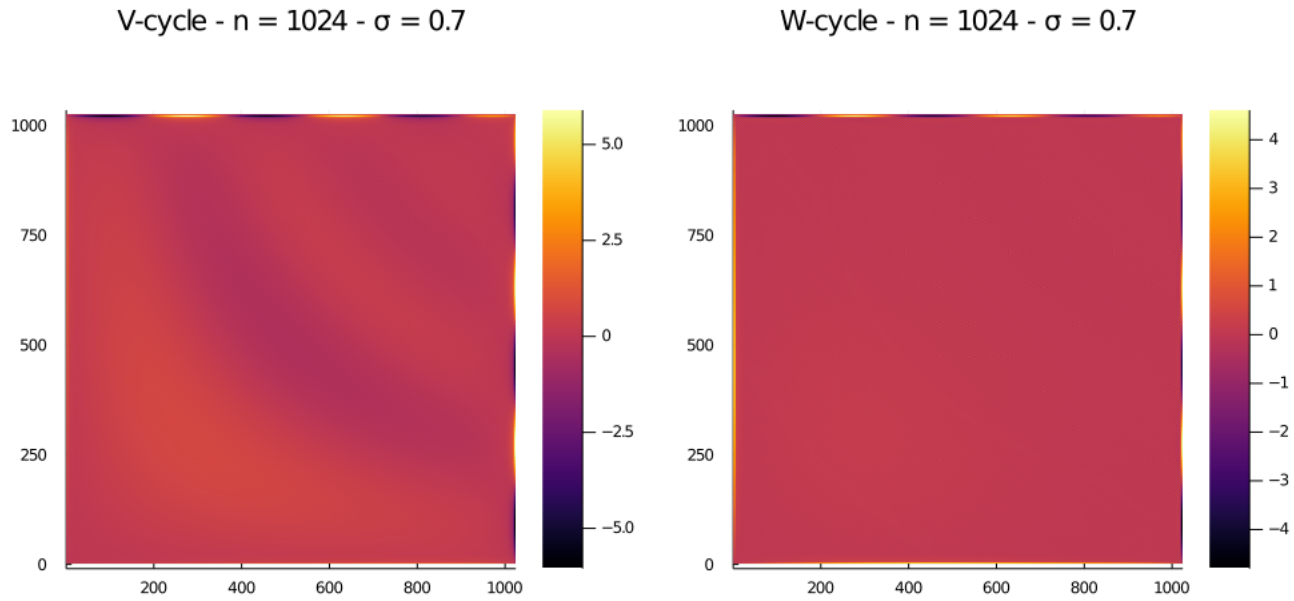
```
# Multigrid 2-steps W-cycle
@elapsed uⱼ₁ = multigrid(A₁₁, bₙ₁, uₙ₁, 3, ω, 1e-30, 2)
```

By just executing both solvers, we can already notice the difference in time between both solvers. The difference shown above actually makes sense: Our W-cycle is a **2-step** one on a **3-level** multigrid, which means it takes the exact form of a W, then we're executing the same number of iterations but **twice**. This gets reflected by the *more-than-twice* execution time we have with the W-cycle, compared to the V-cycle.

```
"||eᵥ|| > ||eₓ||"
```

From here, we can confirm that the W-cycle was actually worth the wait, as it did help smoothing out the error in a more extreme way than the V-cycle did.

```
"Error norm gain: 95.90758"
```



2- $n = 1024$, $\sigma = 7$

```
begin
    σ₂  = 7.
    A₁₂ = A₁(σ₂)
    uᵢ₂ = multigrid(A₁₂, bₙ₁, uₙ₁, 3, ω, 1e-30, 1)      # Multigrid V-cycle
    uⱼ₂ = multigrid(A₁₂, bₙ₁, uₙ₁, 3, ω, 1e-30, 2)      # Multigrid W-cycle
    eᵢ₂     = bₙ₁ - A₁₂(n)*uᵢ₂                           # V-cycle solution error
    eⱼ₂     = bₙ₁ - A₁₂(n)*uⱼ₂                           # W-cycle solution error
    fastr₂  = norm(eᵢ₂, 2) - norm(eⱼ₂, 2)
end ;
```

```
"||eᵥ|| > ||eₓ||"
```

```
"Error norm gain: 103.13397"
```

```
"Error norm gain - V-cycle: 8.03361"
```

```
"Error norm gain - W-cycle: 15.26"
```

Increasing σ by a factor of x10 lead both solvers to better solutions relatively, but it did have the same effect on both of them as the error gain of the W-cycle over the V-cycle is still nearly the same.

3- $n = 1024$, $\sigma = 70$

```
begin
    σ₃  = 70.
    A₁₃ = A₁(σ₃)
    uᵢ₃ = multigrid(A₁₃, bₙ₁, uₙ₁, 3, ω, 1e-30, 1)      # Multigrid V-cycle
    uⱼ₃ = multigrid(A₁₃, bₙ₁, uₙ₁, 3, ω, 1e-30, 2)      # Multigrid W-cycle
    eᵢ₃     = bₙ₁ - A₁₃(n)*uᵢ₃                           # V-cycle solution error
    eⱼ₃     = bₙ₁ - A₁₃(n)*uⱼ₃                           # W-cycle solution error
    fastr₃  = norm(eᵢ₃, 2) - norm(eⱼ₃, 2)
end ;
```

```
"||eᵥ|| > ||eₓ||"
```

```
"Error norm gain: 121.73866"
```

```
"Error norm gain - V-cycle: 67.6177"
```

```
"Error norm gain - W-cycle: 86.22238"
```

Increasing σ by a factor of x100 lead both solvers to better solutions, and this time, it started showing its effect on the W side of things more than on the V-cycle's side. But still, it also increased the error-wise gain of the V-cycle multigrid solver too.

4- $n = 256\ ,\ \sigma = 7$

```
begin
    n₂  = 256
    h₂  = 1 / n₂
    uₙ₂ = zeros(n₂^2)
    bₙ₂ = [sin(2π*f*i*j) for i=0:h₂:1-h₂ for j=0:h₂:1-h₂]
    uᵢ₄ = multigrid(A₁₂, bₙ₂, uₙ₂, 3, ω, 1e-30, 1)      # Multigrid V-cycle
    uⱼ₄ = multigrid(A₁₂, bₙ₂, uₙ₂, 3, ω, 1e-30, 2)      # Multigrid W-cycle
    eᵢ₄     = bₙ₂ - A₁₂(n₂)*uᵢ₄                          # V-cycle solution error
    eⱼ₄     = bₙ₂ - A₁₂(n₂)*uⱼ₄                          # W-cycle solution error
    fastr₄  = norm(eᵢ₄, 2) - norm(eⱼ₄, 2)
end ;
```

```
"||eᵥ|| > ||eₓ||"
```

```
"Error norm gain: 5.50434"
```

```
"Error norm gain - V-cycle: 417.54523"
```

```
"Error norm gain - W-cycle: 319.9156"
```

Decreasing n by a factor of x4 lead both solvers as usual to **better solutions**, this time, having a **huge effect on the V-cycle**'s side compared to the W-cycle as the difference between the solutions is converging to none. Still, the latter is the better solver on these simulation settings.

As for the effect this had on improving the approximation, this was expected: Actually, as we go down in the grid's size, we're releasing the multigrid from the burden of having to interpolate on bigger grids *(the ones that we got rid of when we divided the fine size by 4)* which means the solver now is **less error-prone** than on the bigger grid case. Also, let's not forget that by doing such, we're giving the multigrid the opportunity to **spend the same number of iterations but on coarser grids**, this helps it converge faster than before. And this is exactly why we witnessed these huge evolutions on the approximation error.

5- $n = 32\ ,\ \sigma = 7$

```
begin
    n₃  = 32
    h₃  = 1 / n₃
    uₙ₃ = zeros(n₃^2)
    bₙ₃ = [sin(2π*f*i*j) for i=0:h₃:1-h₃ for j=0:h₃:1-h₃]
    uᵢ₅ = multigrid(A₁₂, bₙ₃, uₙ₃, 3, ω, 1e-30, 1)      # Multigrid V-cycle
    uⱼ₅ = multigrid(A₁₂, bₙ₃, uₙ₃, 3, ω, 1e-30, 2)      # Multigrid W-cycle
```

```
    e_i5     = b_n3 - A_12(n_3)*u_i5                    # V-cycle solution error
    e_j5     = b_n3 - A_12(n_3)*u_j5                    # W-cycle solution error
    fastr_5  = norm(e_i5, 2) - norm(e_j5, 2)
end ;
```

```
"||e_v|| > ||e_x||"
```

```
"Error norm gain: 0.00611"
```

```
"Error norm gain - V-cycle: 480.95131"
```

```
"Error norm gain - W-cycle: 377.82345"
```

On the abstract side of the story, as the theory states, the W-cycle is still a better solver than the V-cycle. But as we're still approaching coarser grid sizes *(here dividing by a factor of x32)*, the fine-tuning is happening on even coarser grids *(with a size of 4x4 in these settings)* so increasing the number of iterations or further smoothing out the approximation n-times *(as it's the case for the W-cycle)* **isn't going to make any difference**.

Restriction Operators

Trying to visualize the workings of the Multigrid solver, we always stumble on the matrix transformations we're using to interpolate the grid we have to a coarser one *(restriction)* or to a finer one *(prolongation)*. As we already created these 3 operators *(see code in the solvers implementation part)*, we're going to compare them according to what they give in term of approximation error.

For this benchmarking, we're conducting the $n = 1024$, $\sigma = 7$ simulation. We're also going to use the **4-level 2-steps W-cycle** multigrid equipped with the **linearization** prolongation operator for better results and more complexity to be able to well-assess the performance difference between all runs.

```
45.967655775
```

```
# Restriction by Injection
@elapsed u_j6 = multigrid(A_12, b_n1, u_n1, 4, ω, 1e-30, 2, injection, linearize)
```

```
46.039371688
```

```
# Restriction by Halfweighting
@elapsed u_j7 = multigrid(A_12, b_n1, u_n1, 4, ω, 1e-30, 2, halfweight, linearize)
```

```
46.10548316
```

```
# Restriction by Fullweighting
@elapsed u_j8 = multigrid(A_12, b_n1, u_n1, 4, ω, 1e-30, 2, fullweight, linearize)
```

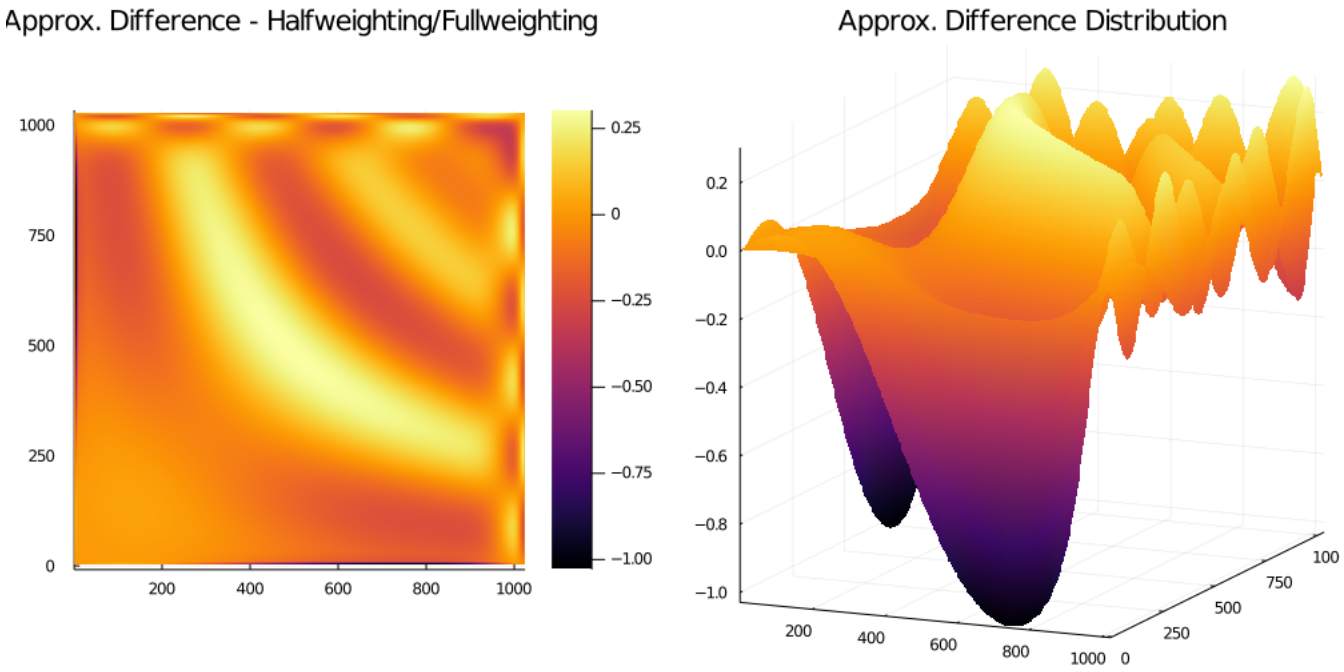
From the execution-times of the 3 variants, we can conclude that thanks to Julia's loops optimization, we can use the 3 restriction operators interchangeably without any loss in performance.

```
# Comparing solutions error rates
begin
    e_j6     = b_n1 - A_12(n)*u_j6          # Injection solution error - e_i
    e_j7     = b_n1 - A_12(n)*u_j7          # Halfweighting solution error - e_h
```

```
    e_j8      = b_n1 - A_12(n)*u_j8          # Fullweighting solution error - e_φ
    fastr_67 = norm(e_j6, 2) - norm(e_j7, 2)
    fastr_68 = norm(e_j6, 2) - norm(e_j8, 2)
    fastr_78 = norm(e_j7, 2) - norm(e_j8, 2)
end ;
```

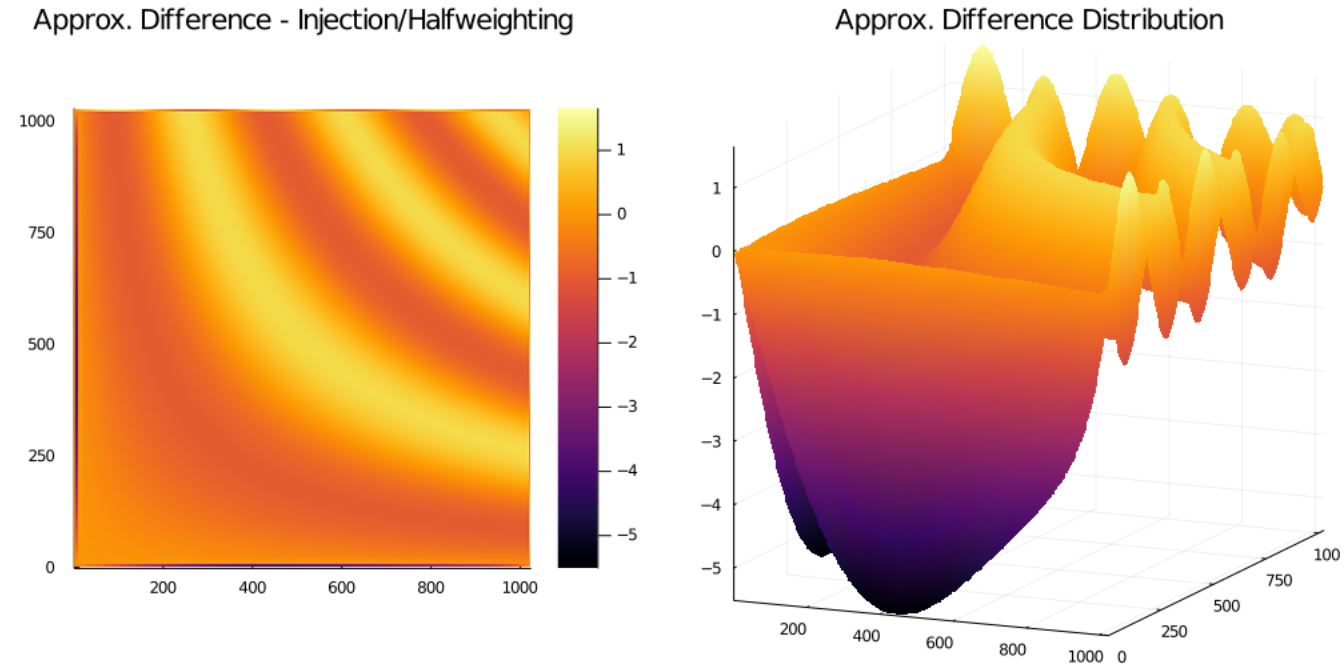
"||eₕ|| < ||eᵩ||"

"Error norm gain - HW/FW: 113.10642"



"||eᵢ|| < ||eₕ||"

"Error norm gain - I/HW: 370.85025"



"||eᵢ|| < ||eᵩ||"

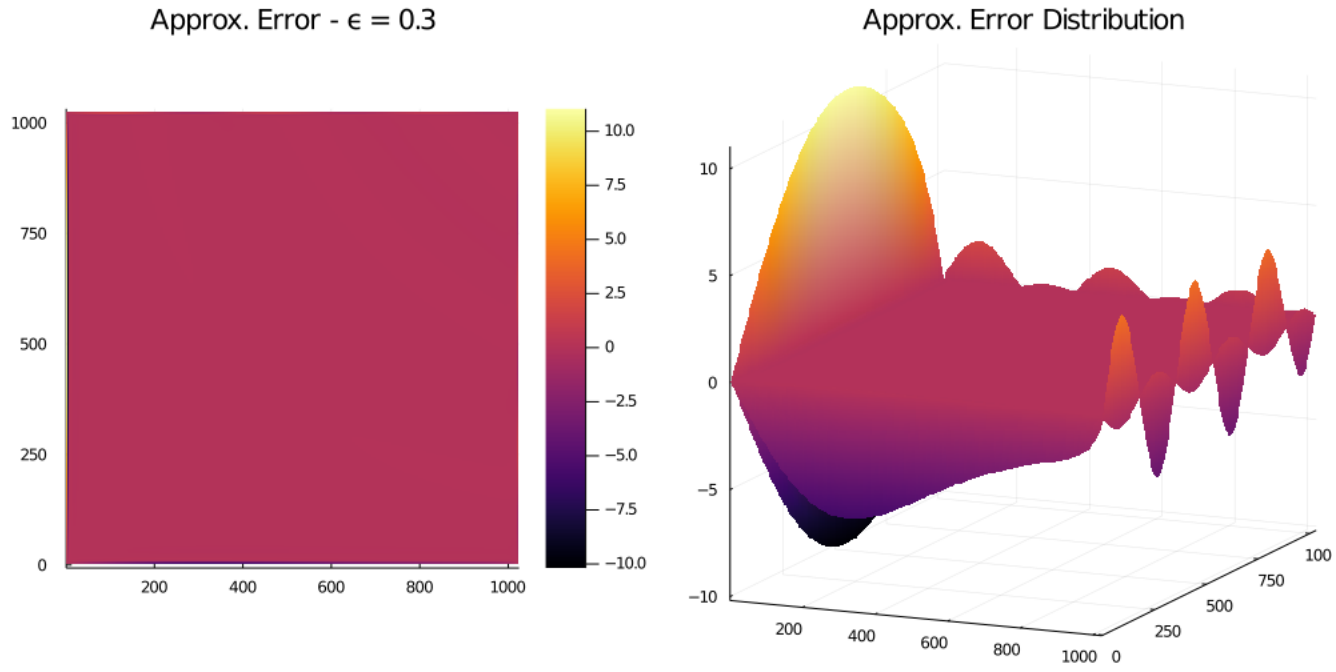
"Error norm gain I/FW: 483.95667"

What we learn from these results is that as long as we're not changing the local values of the solution matrices, there's no problem coarsening the grid some more. Which is in a bit the opposite result of the comparison between the *linearization* and the direct *enlargement* prolongations operators. This may be explained by the fact that when using the multigrid solver, we're trying to port our problem into coarser grids to solve it on that level, so if we're changing the values **even if it's a local-aware change** *(as it's the case for halfweighting and even more in fullweighting)*, we may be **changing the target distribution** and by such furthering our path to the optimal minima.

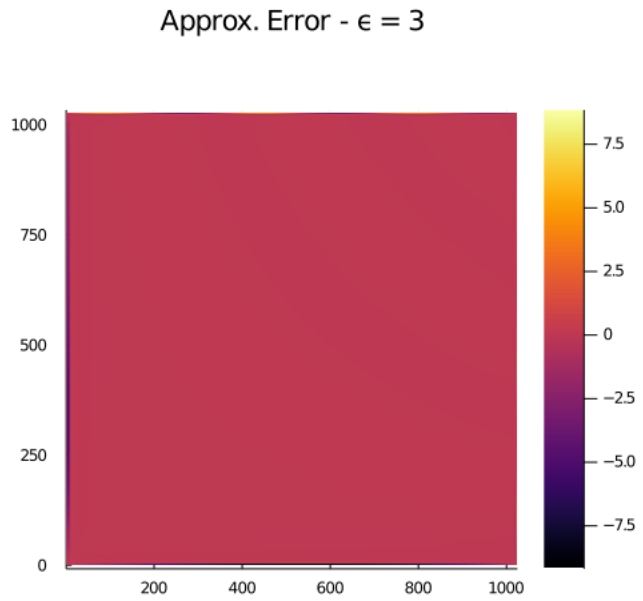
Anisotropic Problem Resolution

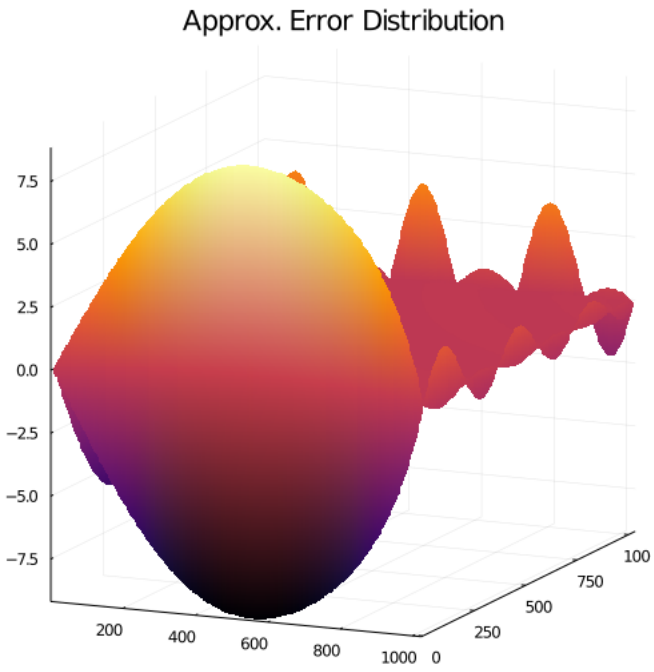
We already constructed the Anisotropic problem PDE (P_2) in the PDE's & Operators Construction part of the document, so we're going to directly try to solve it using our 4-level 2-step W-cycle multigrid equipped by the injection & linearization interpolation operators.

1- $\epsilon = 0.3$

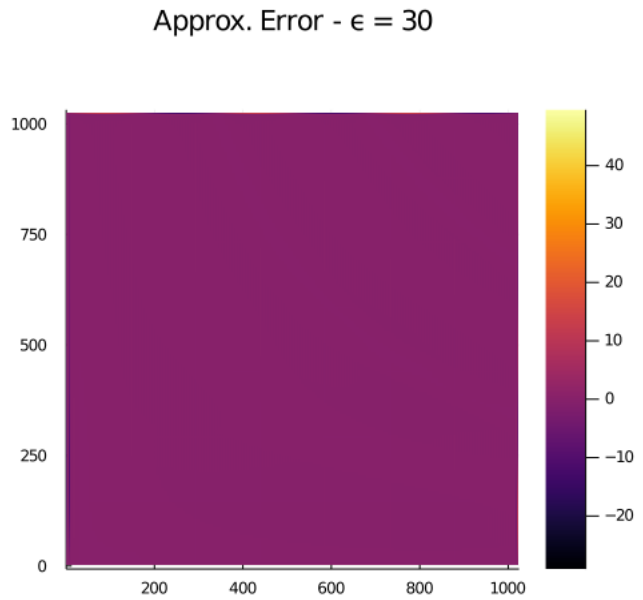


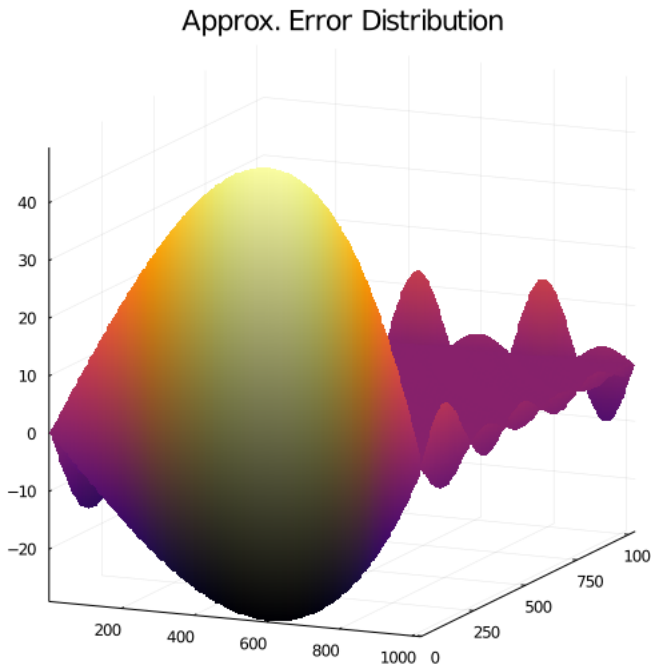
2- $\epsilon = 3$





3- $\epsilon = 30$





Trying to analyze the figures we get from the 3 simulations we ran, we can identify one main problem which is the **non-convergence on the pure directions** being X and Y axis of the vector space. Also, when we look into the difference between the first and the second graphs, we may conclude that the error is also *alternating from one direction into the other* while moving through intervals of values of ϵ.

This is actually caused by the nature of the **deformation** that the PDE applies to the vector space. As the equation P_2 states, differentiating the solution on the X-axis by one unit will directly force differentiating it by ϵ units onto the Y-axis.

$$-\frac{\partial^2 u(x,y)}{\partial x^2} - \epsilon\frac{\partial^2 u(x,y)}{\partial y^2} = f(x,y) \quad (P_2)$$

While this may sound not-harmful at all for our solvers, what we can guess is causing this issue, is actually our coarsening strategies. They're **based on equally distributed spaces** and that's why they're affecting **each one of the neighbors equal coefficients** when using neighbors-based coarsening. And this is the exact reason why the injection is yielding the best restriction therefore the best approximation among all the other operators on this problem.

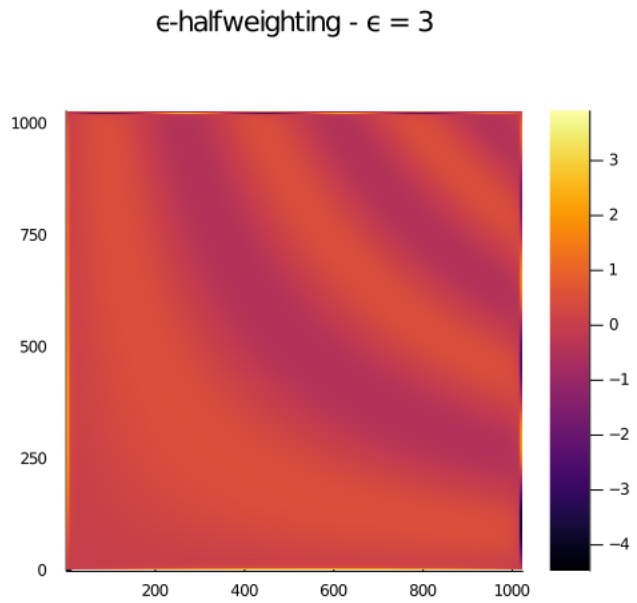
One new way we can think of solving this problem, is to try to create a *halfweighting-like version of a restriction operator*, but **taking into account the ϵ parameter for the Y direction**.

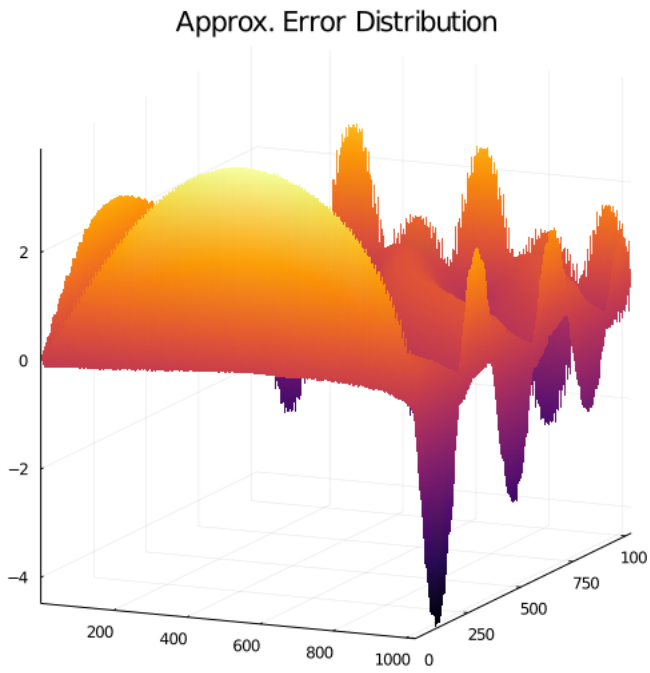
```
begin
    """
        ϵ_halfweight(ϵ, grid)

    A function that returns the ϵ-halfweighting of the grid taken as input.
    """
    function ϵ_halfweight(ϵ, grid)
        g = Float64.(grid)
        for i=2:2:size(grid,1)-1, j=2:2:size(grid,2)-1
            g[i,j] = ϵ * g[i,j] / 4 - (
                g[i-1,j] + g[i+1,j]
                + (1 - ϵ/2) * g[i,j-1] + (1 - ϵ/2) * g[i,j+1]) / 8
        end
        return injection(g)
    end

    """
        ϵ_halfweight(ϵ)

    A function that constructs the ϵ-halfweighting operator.
    """
    ϵ_halfweight(ϵ) = grid -> ϵ_halfweight(ϵ, grid)
end ;
```





```
-55.92162864107422
```

```
norm(e_{j_{12}}, 2) - norm(e_{j_{10}}, 2)
```

And voilà!

We finally obtained an operator that worked better and produced a cleaner approximation with less error norm than the best operator among the restrictors tried on the anisotropic problem.

The idea behind it is that as the Y-axis has got **stretched out by** ϵ, then at every point of our grid, we'll have an ϵ-contribution of one element, and $1 - \epsilon/N$, N being the number of neighbors around it that are taken into account.

By such, we could decrease our norm by **55-units** on the euclidian norm of the vector space.

Rami KADER . HPC-AI20 - Mines ParisTech

Sat. Jan 16, 2021