

# IPM407: Modelación Computacional con Algoritmos Rápidos

## Métodos Basados en Transformada de Fourier

*Departamento de Ingeniería Mecánica,  
Universidad Técnica Federico Santa María  
Valparaíso, Chile*

Profesor: Christopher Cooper

Alumno: Javier Gómez G.

15/06/2017



UNIVERSIDAD TECNICA  
FEDERICO SANTA MARIA





## Resumen:

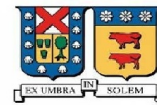
Este Informe contiene información sobre los resultados obtenidos al realizar el estudio de la implementación de métodos Multi mallas para acelerar el calculo en códigos para métodos de relajación. Para dicha tarea, este informe parte por dar una pequeña descripción del equipo y el que se utilizaron. Posteriormente se describe la situación de estudio a partir de un caso particular de la ecuación de Poisson el cual se discretiza en base a un esquema de diferencias finitas centrado de orden cuadrático. Luego de la descripción del caso estudiado se exponen los codigos utilizados para llevar a cabo esa labor.

Posterior a todas las descripciones preliminares se exponen los resultados obtenidos a partir del código desarrollado, estos exponen partiendo de los resultados cualitativos para un campo escalar teórico y para las aproximaciones usando los métodos de relajación y las formas aceleradas de dichos métodos, luego se exponen cuadros resumen de los resultados cuantitativos para posteriormente pasar a gráficos que reflejan la complejidad algoritmica de los métodos.

En base a los resultados expuestos en este trabajo se da lugar a la discusión de los análisis y conclusiones a las que se llegaron luego de desarrollar los pasos descritos previamente.

## Objetivos:

- **Principal:** Implementar Métodos multi-mallas para resolver sistemas lineales reconociendo sus ventajas y desventajas.
- **Específicos:**
  - Reconocer las cualidades de los métodos de relajación.
  - Implementar *V-cycle* y establecer su ventaja respecto a los métodos de relajación simple.
  - Implementar *Full-multigrid-scheme* y reconocer sus ventajas y desventajas respecto a *V-cycle* y relajación simple.
  - Comprobar que se cumplen las predicciones teóricas con respecto a los tiempos de cálculos.



# Índice

<b>1. Computador y lenguaje utilizado</b>	<b>4</b>
<b>2. Desarrollo</b>	<b>5</b>
2.1. Discretización y Esquemas . . . . .	5
2.2. Códigos . . . . .	6
<b>3. Resultados</b>	<b>12</b>
3.1. Validación . . . . .	12
3.1.1. Relajaciones Simples . . . . .	13
3.1.2. V-Cycle . . . . .	14
3.1.3. Full-Multigrid . . . . .	15
3.2. Tiempo, Residuo y Errores . . . . .	17
3.2.1. Comparación multi malla a iguales condiciones . . . . .	22
<b>4. Análisis y Conclusiones</b>	<b>23</b>

## 1. Computador y lenguaje utilizado

El pc utilizado para realizar los cálculos fue:

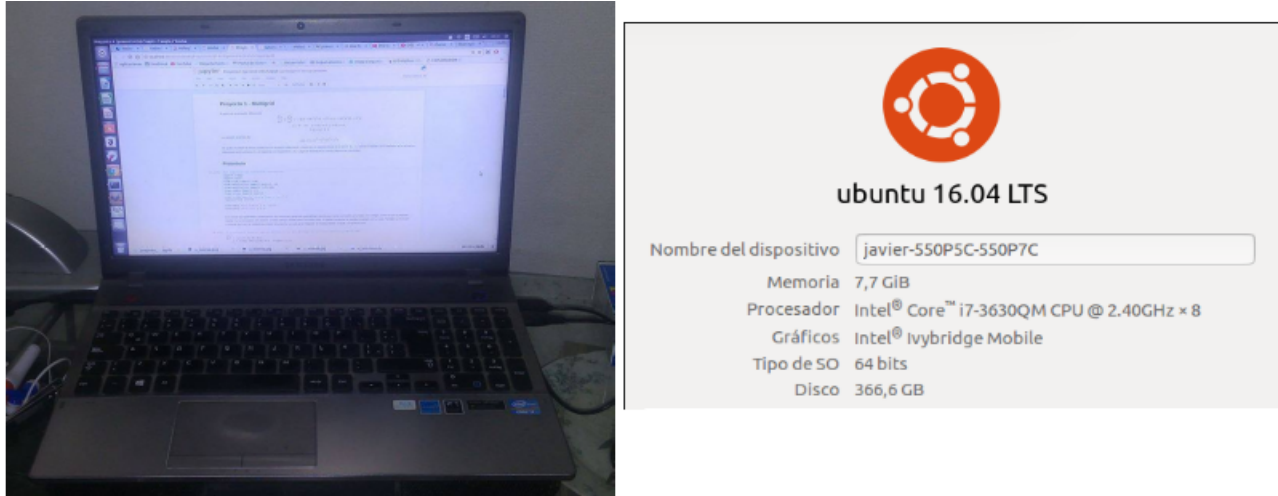


Figura 1: Presentación y resumen de las características del PC a utilizado para hacer los cálculos

En cuanto al lenguaje utilizado para realizar los códigos fue python. Los codigos se escribieron en notbooks inter-activos conocidos como **ipython notebook** mediante la interfaz de jupyter.



Figura 2: Python versión 2.7 y Jupyter versión 4.2.0

## 2. Desarrollo

### 2.1. Discretización y Esquemas

Con el fin de llevar a cabo los objetivos, se estudio el comportamiento de la siguiente ecuación de Poisson:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -2[(1 - 6x^2)y^2(1 - y^2) + (1 - 6y^2)x^2(1 - x^2)] \quad (1)$$

$$u = 0 \quad \text{en} \quad x = 0, x = 1, y = 0, y = 1$$

$$0 \leq x \leq 1 \quad 0 \leq y \leq 1$$

Que tiene solución analítica:

$$u(x, y) = (x^2 - x^4)(y^4 - y^2) \quad (2)$$

Para resolver de forma numérica la EDP de la ecuación (1), se utilizó un esquema de discretización por diferencias finitas centrada de orden dos para las segundas derivadas, es decir:

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} + o(\Delta x^2) \quad y \quad \frac{\partial^2 u}{\partial y^2} = \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2} + o(\Delta y^2) \quad (3)$$

donde en este caso  $o(\Delta x^2)$  y  $o(\Delta y^2)$  son el orden del error de la discretización producto de la expansión en serie de Taylor para las segundas derivadas de  $u$ .

Ademas, de ahora en adelante y con el fin de simplificar la notación, se dirá que:

$$f_{i,j} = f(x_i, y_j) = -2[(1 - 6x_i^2)y_j^2(1 - y_j^2) + (1 - 6y_j^2)x_i^2(1 - x_i^2)] \quad (4)$$

En este trabajo se pretende resolver el sistema  $Au = f$  obtenido al reemplazar las discretizaciones de (3) en (1) y usando métodos de relajación como Jacobi, Gauss-Seidel y Red-Black Gauss-Seidel [1]. Luego, considerando que  $v$  es una aproximación de  $u$  y representado los métodos de iteración libre de matrices, se tiene:

$$Jacobi : \quad v_{i,j}^{k+1} = \frac{1}{2} \frac{\Delta x^2 \Delta y^2}{\Delta y^2 + \Delta x^2} \left[ -f_{i,j} + \left( \frac{u_{i-1,j}^k + u_{i+1,j}^k}{\Delta x^2} + \frac{u_{i,j-1}^k + u_{i,j+1}^k}{\Delta y^2} \right) \right] \quad (5)$$

$$Gauss - Seidel : \quad v_{i,j}^{k+1} = \frac{1}{2} \frac{\Delta x^2 \Delta y^2}{\Delta y^2 + \Delta x^2} \left[ -f_{i,j} + \left( \frac{u_{i-1,j}^{k+1} + u_{i+1,j}^k}{\Delta x^2} + \frac{u_{i,j-1}^{k+1} + u_{i,j+1}^k}{\Delta y^2} \right) \right] \quad (6)$$

Para Red-Black Gauss-Seidel se dividen la malla en nodos *rojos* y *negros* (Figura 3), se resuelven los nodos *rojos* con *Jacobi* y luego, usando los resultados calculados con *Jacobi* se calculan los nodos *negros* usando *Gauss - Seidel*.

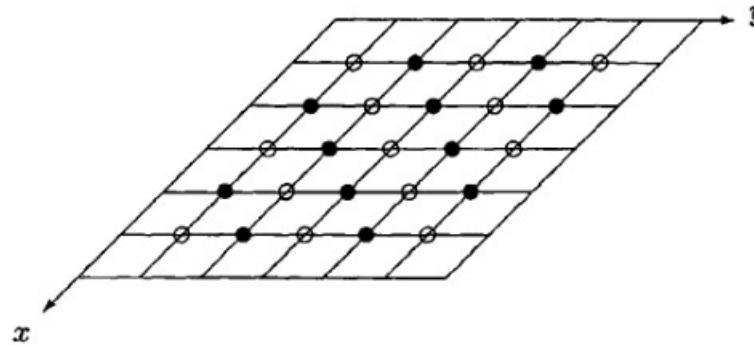


Figura 3: Esquema de una malla dividida en nodos para aplicar RB-GS. Los nodos rojos.<sup>en</sup> este esquema son los blancos.

## 2.2. Códigos

Antes de escribir los algoritmos se importan las librerías de *numpy*, *matplotlib*, *time* y *numba*, para definir arreglos y algunas operaciones matemáticas, hacer gráficos, medir tiempo y acelerar la compilación de algunas rutinas, respectivamente.

**Preámbulo**

```
#se importan las librerías necesarias
import numpy
import math
from time import time
from matplotlib import pyplot, cm
from matplotlib import rcParams
from numba import jit
from scipy import sparse
from scipy.sparse.linalg import spsolve
%matplotlib inline

rcParams['font.family'] = 'serif'
rcParams['font.size'] = 8
```

Figura 4: Códigos para importar librerías a utilizar

Antes de definir los algoritmos para los métodos de relajación es necesario definir algunas funciones que se ocuparan a lo largo de todo el código, estas funciones se pueden ver en la Figura 5. Que hace cada una de las funciones de la Figura 5 son descritas en los comentarios que aparecen en la figura

```
# La primera función que se define es la que entrega la solución analítica de la EDP.
@jit
def u_teo(hx,hy,Nx,Ny):
    u = numpy.empty([Ny,Nx], dtype=float)
    for i in range(Ny):
        for j in range(Nx):
            u[i,j]=(((j*hx)**2)-((j*hx)**4))*(((i*hy)**4)-((i*hy)**2))
    return u

# Se define la función f del sistema Ax=f en que f estará dada por
# f(x,y)=-2[(1-6x^2)y^2(1-y^2)+(1-6y^2)x^2(1-x^2)]
@jit
def function_f(hx,hy,Nx,Ny):
    f = numpy.empty([Nx,Ny], dtype = float)
    for i in range(Ny):
        for j in range(Nx):
            f[i,j]= -2.*((1.-6.*((j*hx)**2))*((i*hy)**2)*(1.-((i*hy)**2))+
                (1.-6.*((i*hy)**2))*((j*hx)**2)*(1.-((j*hx)**2)))
    return f

# La función f en el sistema Ax=f es un vector, pero por simplicidad se utilizará como una matriz

# Con la siguiente función se pretende ordenar vectores en forma de matriz para así poder determinar máximos
# en caso de ser necesario.
@jit
def matrix_to_vector(matrix,Nx,Ny):
    NN = (Nx-2)*(Ny-2)
    vector = numpy.empty([NN])
    j = 1
    k = 1
    for i in range(NN):
        vector[i] = matrix[j,k]
        k = k+1
        if k == (Nx):
            j = j+1
            k = 1

    return vector

# La siguiente función calcula los componentes del residual de la forma r=f-Av en una matriz.
def residual(v,f,hx,hy,Nx,Ny):
    # Av = numpy.zeros([Nx+1,Ny+1])
    r = numpy.zeros([Ny,Nx])
    r[1:Ny-1,1:Nx-1]=(f[1:Ny-1,1:Nx-1]-(v[0:Ny-2,1:Nx-1]-2.*v[1:Ny-1,1:Nx-1]+v[2:Ny,1:Nx-1])/(hy**2)
        - (v[1:Ny-1,0:Nx-2]-2.*v[1:Ny-1,1:Nx-1]+v[1:Ny-1,2:Nx])/(hx**2))
    return r

# Recibe un vector y devuelve la norma infinita y la norma euclídea del vector.
def norma(vector,H):
    norm = numpy.zeros(2)
    norm[0] = max(abs(vector))
    norm[1] = H*(sum(vector**2)**(0.5))
    return norm
```

Figura 5: Funciones preliminares

A continuación se presentan las funciones que realizan las relajaciones:

```
# Aplicando el metodo iterativo de Gauss-Seidel para encontra v como aproximación de u se crean
def Relax(v_in,f_in,hx,hy,T_o_R):
    nx,ny = len(v_in[0,:]), len(v_in[:,0])
    if T_o_R == 'Jacobi':
        v = jacobi(v_in,f_in,hx,hy,nx,ny)
    elif T_o_R == 'Gauss-Seidel':
        v = gauss_seidel(v_in,f_in,hx,hy,nx,ny)
    elif T_o_R == 'Red-Black Gauss-Seidel':
        v = red_black_gauss_seidel(v_in,f_in,hx,hy,nx,ny)
    return v

def jacobi(v_in,f_in,hx,hy,nx,ny):
    vn = numpy.copy(v_in)
    vn1 = numpy.copy(v_in)
    factor = 0.5*((hx**2)*(hy**2))/((hx**2)+(hy**2))
    vn1[1:ny-1,1:nx-1] = factor*(-f_in[1:ny-1,1:nx-1]+
                                   (vn[0:ny-2,1:nx-1]+vn[2:ny,1:nx-1])/(hy**2)+
                                   (vn[1:ny-1,0:nx-2]+vn[1:ny-1,2:nx])/(hx**2))

    return vn1

@jit
def gauss_seidel(vini,f,hx,hy,Nx,Ny):
    vn = numpy.copy(vini)
    vn1 = numpy.copy(vini)
    factor = ((hx**2)*(hy**2))/((hx**2)+(hy**2))

    for i in range(1,Ny-1):
        for j in range(1,Nx-1):
            vn1[i,j] = 0.5*factor*(-f[i,j]+(vn1[i-1,j]+vn[i+1,j])/(hy**2)+(vn1[i,j-1]+vn[i,j+1])/(hx**2))
    return vn1

def red_black_gauss_seidel(v_in,f_in,hx,hy,nx,ny):
    vn = numpy.copy(v_in)
    vn1 = numpy.copy(v_in)
    factor = 0.5*((hx**2)*(hy**2))/((hx**2)+(hy**2))
    #red
    vn1[1:ny-1:2,1:nx-1:2] = factor*(-f_in[1:ny-1:2,1:nx-1:2] +
                                   (vn[0:ny-2:2,1:nx-1:2]+vn[2:ny:2,1:nx-1:2])/(hy**2)+
                                   (vn[1:ny-1:2,0:nx-2:2]+vn[1:ny-1:2,2:nx:2])/(hx**2))
    vn1[2:ny-2:2,2:nx-2:2] = factor*(-f_in[2:ny-2:2,2:nx-2:2] +
                                   (vn[1:ny-3:2,2:nx-2:2]+vn[3:ny-1:2,2:nx-2:2])/(hy**2)+
                                   (vn[2:ny-2:2,1:nx-3:2]+vn[2:ny-2:2,3:nx-1:2])/(hx**2))
    #black
    vn1[1:ny-1:2,2:nx-2:2] = factor*(-f_in[1:ny-1:2,2:nx-2:2] +
                                   (vn1[0:ny-2:2,2:nx-2:2]+vn[2:ny:2,2:nx-2:2])/(hy**2)+
                                   (vn1[1:ny-1:2,1:nx-3:2]+vn[1:ny-1:2,3:nx-1:2])/(hx**2))
    vn1[2:ny-2:2,1:nx-1:2] = factor*(-f_in[2:ny-2:2,1:nx-1:2] +
                                   (vn1[1:ny-3:2,1:nx-1:2]+vn[3:ny-1:2,1:nx-1:2])/(hy**2)+
                                   (vn1[2:ny-2:2,0:nx-2:2]+vn[2:ny-2:2,2:nx:2])/(hx**2))

    return vn1
```

Figura 6: Códigos para las relajaciones

Posteriormente se escriben los códigos para V-cycle y para Multigrid, pero antes -debido al movimiento entre mallas finas y gruesas- es necesario escribir códigos para transferencia de mallas.



```
def intergrid_transfer(matrixh,Nx,Ny,int_transf):
    if int_transf == 'Injection':
        Matrix2h = injection(matrixh,Nx,Ny)
    elif int_transf == 'Full-Weighting':
        Matrix2h = full_weighting(matrixh, Nx,Ny)
    return Matrix2h

def injection(matrixh, Nx, Ny):
    nnx = 1 + (Nx-1)/2
    nny = 1 + (Ny-1)/2
    Matrix2h = numpy.zeros([nny,nnx])
    Matrix2h[0,:] = matrixh[0,0:Nx:2]#superior
    Matrix2h[nny-1,:] = matrixh[Ny-1,0:Nx:2]#inferior
    Matrix2h[1:nny-1,0] = matrixh[2:Ny-2:2,0] #izquierda
    Matrix2h[1:nny-1,nnx-1] = matrixh[2:Ny-2:2,Nx-1] #derecha
    Matrix2h[1:nny-1,1:nnx-1] = matrixh[2:Ny-2:2,2:Nx-2:2]#interno
    return Matrix2h

def full_weighting(matrixh, Nx, Ny):
    nnx = 1 + (Nx-1)/2
    nny = 1 + (Ny-1)/2
    Matrix2h = numpy.zeros([nny,nnx])
    Matrix2h[0,:] = matrixh[0,0:Nx:2] #superior
    Matrix2h[nny-1,:] = matrixh[Ny-1,0:Nx:2] #inferior
    Matrix2h[1:nny-1,0] = matrixh[2:Ny-2:2,0] #izquierda
    Matrix2h[1:nny-1,nnx-1] = matrixh[2:Ny-2:2,Nx-1] #derecha
    Matrix2h[1:nny-1,1:nnx-1] = (4*matrixh[2:Ny-2:2,2:Nx-2:2] +
        (matrixh[1:Ny-3:2,1:Nx-3:2]+matrixh[1:Ny-3:2,3:Nx-1:2]
        +matrixh[3:Ny-1:2,1:Nx-3:2]+matrixh[3:Ny-1:2,3:Nx-1:2]))+
        2*(matrixh[1:Ny-3:2,2:Nx-2:2]+matrixh[3:Ny-1:2,2:Nx-2:2]
        +matrixh[2:Ny-2:2,1:Nx-3:2]+matrixh[2:Ny-2:2,3:Nx-1:2]))/16. #internos

    return Matrix2h

# Interpolation es un funcion intergrid que genera mallas mas finas interpolando los puntos de una malla
# mas gruesa. Esta funcion no esta incluida en las opciones de intergrid_transfer
def interpolation(matrix2h, Nx, Ny):
    nnx = 1 + 2*(Nx-1)
    nny = 1 + 2*(Ny-1)
    Matrixh = numpy.zeros([nny,nnx])
    Matrixh[0:nny:2,0:nnx:2] = matrix2h[0:Ny,0:Nx] #directo
    Matrixh[1:nny-1:2,0:nnx:2] = 0.5*(matrix2h[0:Ny-1,0:Nx]+matrix2h[1:Ny,0:Nx]) #por filas
    Matrixh[0:nny:2,1:nnx-1:2] = 0.5*(matrix2h[0:Ny,0:Nx-1]+matrix2h[0:Ny,1:Nx]) #por columnas
    Matrixh[1:nny-1:2,1:nnx-1:2] = 0.25*(matrix2h[0:Ny-1,0:Nx-1]+matrix2h[1:Ny,0:Nx-1]+
        matrix2h[0:Ny-1,1:Nx]+matrix2h[1:Ny,1:Nx]) #intermedios

    return Matrixh
```

Figura 7: Los codigos de transferencia de malla son *injection* y *fullweighting* para ir de mallas finas a gruesas e *interpolation* para ir de mallas gruesas a finas

```
def v_cycle(v_aprox_grid,ng,nul,nu2,f,hx,hy,ToR,in_transf):
    v = numpy.empty(ng, dtype = object)
    f_vcy = numpy.empty(ng, dtype = object)
    v[0]=numpy.copy(v_aprox_grid)
    f_vcy[0]=numpy.copy(f)
    nx, ny = len(v[0][0,:]), len(v[0][:,0])
    hhx, hhy = numpy.copy(hx), numpy.copy(hy)

    #para abajo
    for j in range(ng-1):
        for i in range(nul):
            v[j] = Relax(v[j],f_vcy[j],hhx,hhy,ToR)
            r_aux = residual(v[j],f_vcy[j],hhx,hhy,nx,ny)
            f_vcy[j+1] = intergrid_transfer(r_aux, nx, ny,in_transf)
            nx, ny = len(f_vcy[j+1][0,:]), len(f_vcy[j+1][:,0])
            hhx, hhy = 2*hhx, 2*hhy
            v[j+1] = numpy.zeros([ny,nx])
            del r_aux

    #Malla mas gruesa. Metodo directo de resolucio
    factor = 2.*(hhx**2 + hhy**2)/(hhx**2 + hhy**2)

    A_dense = numpy.array([[ -factor,hhy**(-2),0,hhx**(-2),0,0,0,0,0],
                           [hhy**(-2),-factor,hhy**(-2),0,hhx**(-2),0,0,0,0],
                           [0,hhy**(-2),-factor,0,0,hhx**(-2),0,0,0],
                           [hhx**(-2), 0, 0,-factor,hhy**(-2),0,hhx**(-2),0,0],
                           [0,hhx**(-2),0,0,-factor,hhy**(-2),0,hhx**(-2),0],
                           [0,0,hhx**(-2),0,hhy**(-2),-factor,0,0,hhx**(-2)],
                           [0,0,0,hhx**(-2),0,0,-factor,hhy**(-2),0],
                           [0,0,0,0,hhx**(-2),0,hhy**(-2),-factor,hhy**(-2)],
                           [0,0,0,0,0,hhx**(-2),0,hhy**(-2),-factor]])

    A_sparse = sparse.csr_matrix(A_dense)

    f_aux = numpy.array([f_vcy[ng-1][1,1],f_vcy[ng-1][1,2],f_vcy[ng-1][1,3],
                        f_vcy[ng-1][2,1],f_vcy[ng-1][2,2],f_vcy[ng-1][2,3],
                        f_vcy[ng-1][3,1],f_vcy[ng-1][3,2],f_vcy[ng-1][3,3]])

    x_aux = spsolve(A_sparse,f_aux)

    v[ng-1][1,1:4], v[ng-1][2,1:4], v[ng-1][3,1:4] = (numpy.copy(x_aux[0:3]),
                                                         numpy.copy(x_aux[3:6]),
                                                         numpy.copy(x_aux[6:9]))

    #para arriba
    for j in range(ng-1,0,-1):
        v[j-1] = v[j-1]+interpolation(v[j], nx, ny)
        nx, ny = len(v[j-1][0,:]), len(v[j-1][:,0])
        hhx, hhy = hhx/2, hhy/2
        for i in range(nu2):
            v[j-1]=Relax(v[j-1],f_vcy[j-1],hhx,hhy,ToR)

    return v[0]
```

Figura 8: Codigos para ciclo V. La malla mas gruesa se resuelve con métodos directos.

```
def Full_multigrid(f_in, ng, nu0, nu1, nu2, hx, hy, ToR, int_transf):
    v, f = numpy.empty(ng, dtype = object), numpy.empty(ng, dtype = object)
    f[0] = numpy.copy(f_in)
    nx, ny = len(f_in[0,:]), len(f_in[:,0])
    hhx, hhy = numpy.copy(hx), numpy.copy(hy)

    # Saltos de malla para f
    for i in range(1,ng):
        f[i] = intergrid_transfer(f[i-1], nx, ny, int_transf)
        nx,ny = 1+(nx - 1)/2, 1+(ny - 1)/2

    hhx, hhy = (2**(ng-1))*hhx, (2**(ng-1))*hhy

    #Malla mas gruesa. Metodo directo de resolucion
    factor = 2.*(hhx**2 + hhy**2)/(hhx**2 * hhy**2)

    A_dense = numpy.array([[ -factor, hhy**(-2), 0, hhx**(-2), 0, 0, 0, 0, 0],
        [hhy**(-2), -factor, hhy**(-2), 0, hhx**(-2), 0, 0, 0, 0],
        [0, hhy**(-2), -factor, 0, 0, hhx**(-2), 0, 0, 0],
        [hhx**(-2), 0, 0, -factor, hhy**(-2), 0, hhx**(-2), 0, 0],
        [0, hhx**(-2), 0, 0, -factor, hhy**(-2), 0, hhx**(-2), 0],
        [0, 0, hhx**(-2), 0, hhy**(-2), -factor, 0, 0, hhx**(-2)],
        [0, 0, 0, hhx**(-2), 0, 0, -factor, hhy**(-2), 0],
        [0, 0, 0, 0, hhx**(-2), 0, hhy**(-2), -factor, hhy**(-2)],
        [0, 0, 0, 0, 0, hhx**(-2), 0, hhy**(-2), -factor]])

    A_sparse = sparse.csr_matrix(A_dense)

    f_aux = numpy.array([f[ng-1][1,1], f[ng-1][1,2], f[ng-1][1,3],
        f[ng-1][2,1], f[ng-1][2,2], f[ng-1][2,3],
        f[ng-1][3,1], f[ng-1][3,2], f[ng-1][3,3]])

    x_aux = spsolve(A_sparse, f_aux)

    v[ng-1] = numpy.zeros([ny, nx])
    v[ng-1][1,1:4], v[ng-1][2,1:4], v[ng-1][3,1:4] = (numpy.copy(x_aux[0:3]),
        numpy.copy(x_aux[3:6]),
        numpy.copy(x_aux[6:9]))

    #subida por multigrid
    ng_aux = 1
    for j in range(ng-1, 0, -1):
        v[j-1] = interpolation(v[j], nx, ny)
        nx, ny = 1 + 2*(nx-1), 1 + 2*(ny-1)
        hhx, hhy = hhy/2, hhy/2
        ng_aux = ng_aux + 1
        for i in range(nu0):
            v[j-1] = v_cycle(v[j-1], ng_aux, nu1, nu2, f[j-1], hx, hy, ToR, int_transf)

    return v[0]
```

Figura 9: Codigos para Full-Multigrid. Al igual que en ciclo-V para la malla mas gruesa se resuelve por métodos directos.

Las funciones anteriormente definidas (relajaciones, cambios de malla, ciclo-V y Full-Multigrid) fueron pensadas inicialmente para ocupar mallas con distinta cantidad de nodos por eje ( $N_x \neq N_y$ ) y que el espacio entre nodos también sea distinta ( $\Delta x \neq \Delta y$ ) pero a lo largo de la programación esa generalidad se perdió y solo es posible utilizar estos códigos cuando se cumple que  $N_x = N_y$  y  $\Delta x = \Delta y$ .

En *Ciclo - V* y en *Full - Multigrid* se fuerza a que la malla mas gruesa sea de  $N_x = N_y = 5$  en la cual, solo los puntos centrales de la malla son incógnitas ya que los bordes son definidos por las condiciones de contorno, en este sentido se genera un sistema de nueve ecuaciones y nueve incógnitas.

Además, *Ciclo - V* cuenta con los parámetros  $\nu_1$  y  $\nu_2$  que son la cantidad de relajaciones cuando va cambiando de matrices finas a gruesas y gruesas a finas respectivamente. Para *Full - Multigrid* además se define  $\nu_0$  que es la cantidad de *ciclos*<sub>V</sub> que se realizan mientras se va "subiendo" desde las mallas gruesas a las finas.

### 3. Resultados

#### 3.1. Validación

Para saber si los resultados obtenidos por los códigos es apropiado primero se hace una validación cualitativa.

El campo  $u$  teórico se ve de la siguiente forma

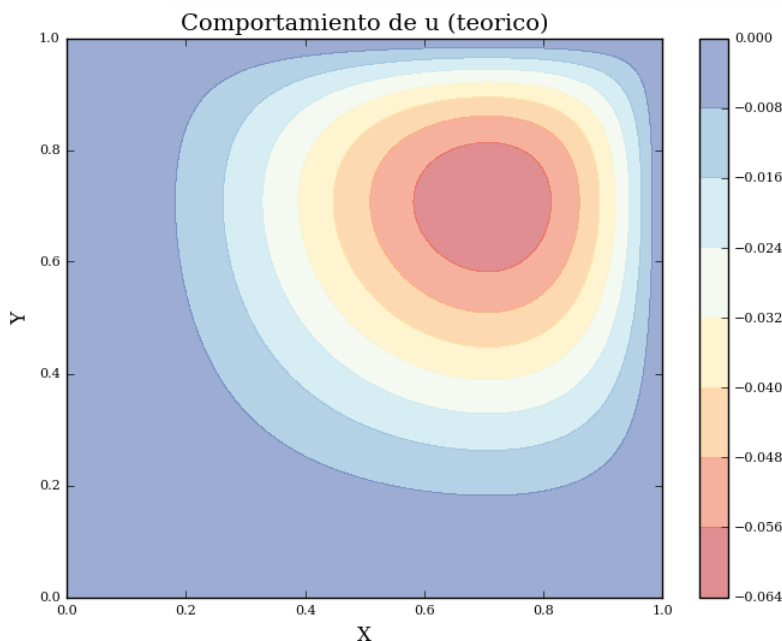
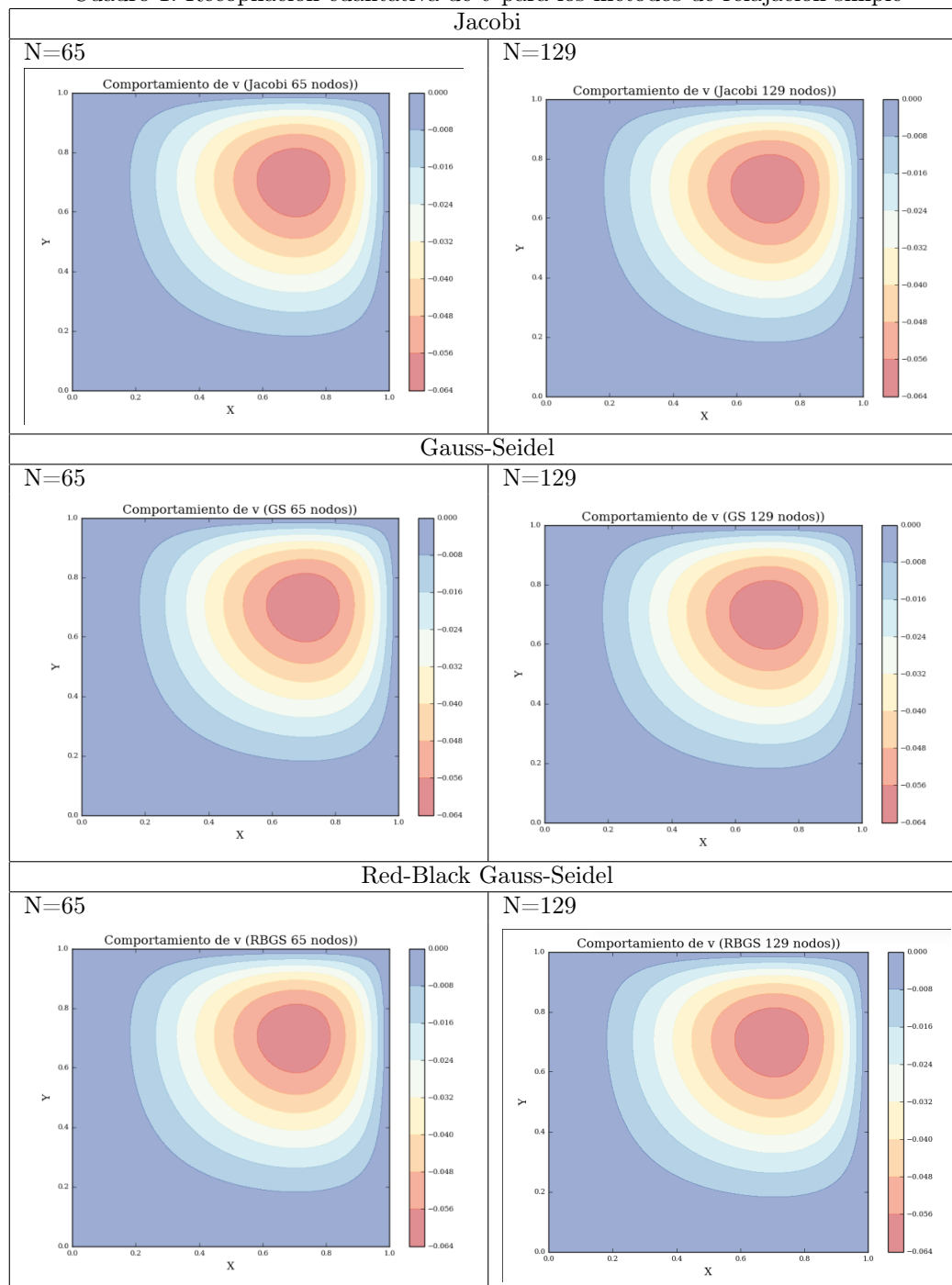


Figura 10: Campo  $u$  teórico

Las aproximaciones  $v$  de  $u$  con los distintos métodos y una tolerancia para el residual de  $10^{-8}$  se ven de la siguiente forma.

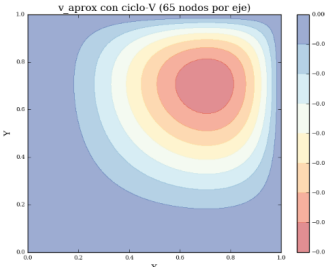
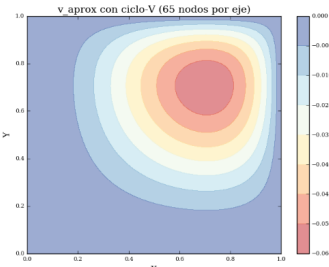
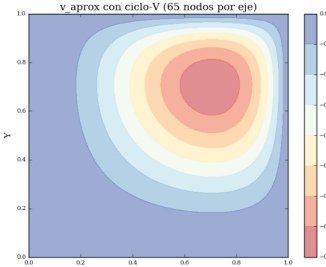
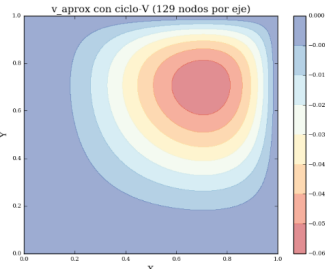
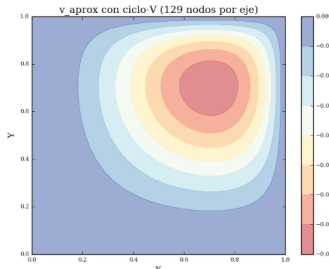
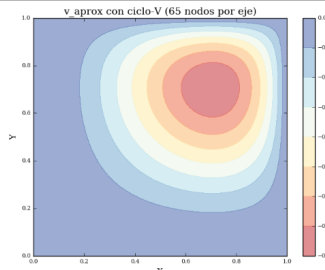
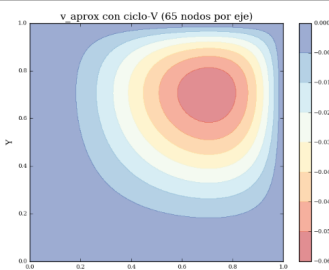
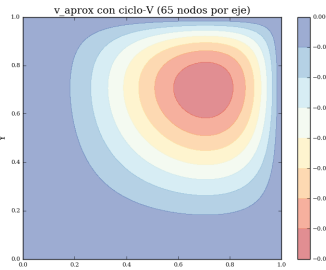
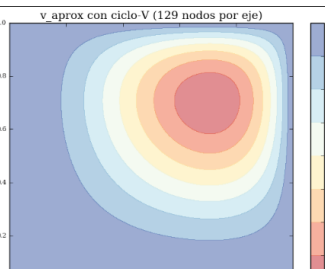
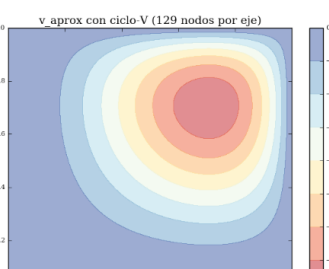
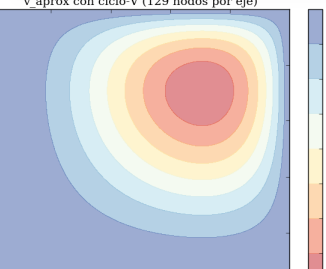
### 3.1.1. Relaciones Simples

Cuadro 1: Recopilación cualitativa de  $v$  para los metodos de relajación simple



### 3.1.2. V-Cycle

Cuadro 2: Recopilación cualitativa de  $v$  para  $V - Cycle$  usando  $\nu_1 = 2$ ,  $\nu_2 = 2$

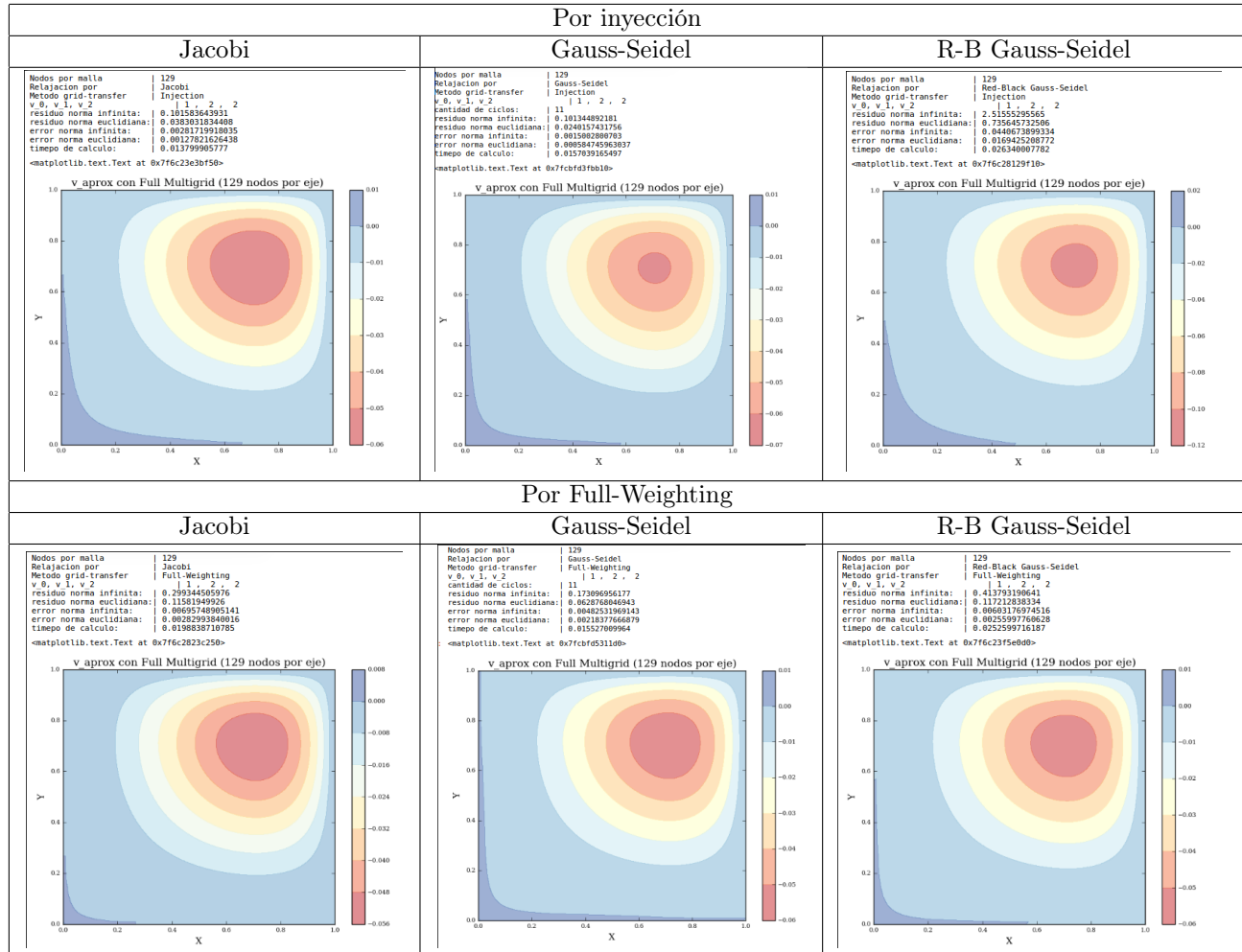
Inyección																						
Jacobi	Gauss-Seidel	R-B Gauss-Seidel																				
N=65																						
																						
N=129																						
		<table><tr><td>Nodos por malla</td><td>129</td></tr><tr><td>Relajacion por</td><td>Red-Black Gauss-Seidel</td></tr><tr><td>Metodo grid-transfer</td><td>Injection</td></tr><tr><td><math>\nu_1, \nu_2</math></td><td>2, 2</td></tr><tr><td>cantidad de ciclos:</td><td>4588</td></tr><tr><td>residuo norma infinita:</td><td>nan</td></tr><tr><td>residuo norma euclidiana:</td><td>nan</td></tr><tr><td>error norma infinita:</td><td>nan</td></tr><tr><td>error norma euclidiana:</td><td>nan</td></tr><tr><td>tiempo de calculo:</td><td>34.8428270817</td></tr></table>	Nodos por malla	129	Relajacion por	Red-Black Gauss-Seidel	Metodo grid-transfer	Injection	$\nu_1, \nu_2$	2, 2	cantidad de ciclos:	4588	residuo norma infinita:	nan	residuo norma euclidiana:	nan	error norma infinita:	nan	error norma euclidiana:	nan	tiempo de calculo:	34.8428270817
Nodos por malla	129																					
Relajacion por	Red-Black Gauss-Seidel																					
Metodo grid-transfer	Injection																					
$\nu_1, \nu_2$	2, 2																					
cantidad de ciclos:	4588																					
residuo norma infinita:	nan																					
residuo norma euclidiana:	nan																					
error norma infinita:	nan																					
error norma euclidiana:	nan																					
tiempo de calculo:	34.8428270817																					
Full-Weighting																						
Jacobi	Gauss-Seidel	R-B Gauss-Seidel																				
N=65																						
																						
N=129																						
																						



### 3.1.3. Full-Multigrid

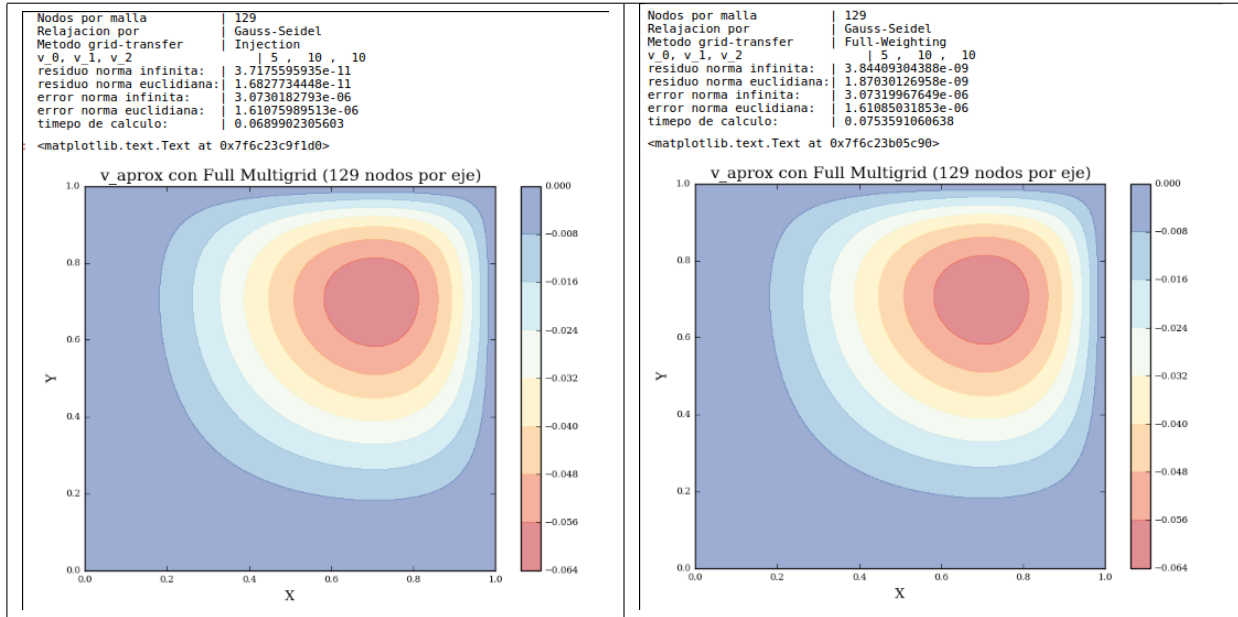
Para Full-Multigrid, debido a que no se consigue el valor cualitativo adecuado para los parámetros  $\nu$  utilizados, es que solo se muestra como se ve el campo para 129 nodos por eje

Cuadro 3: Recopilación cualitativa de  $v$  para *Multigrid* usando  $\nu_1 = 2$ ,  $\nu_2 = 2$  y  $\nu_0 = 1$

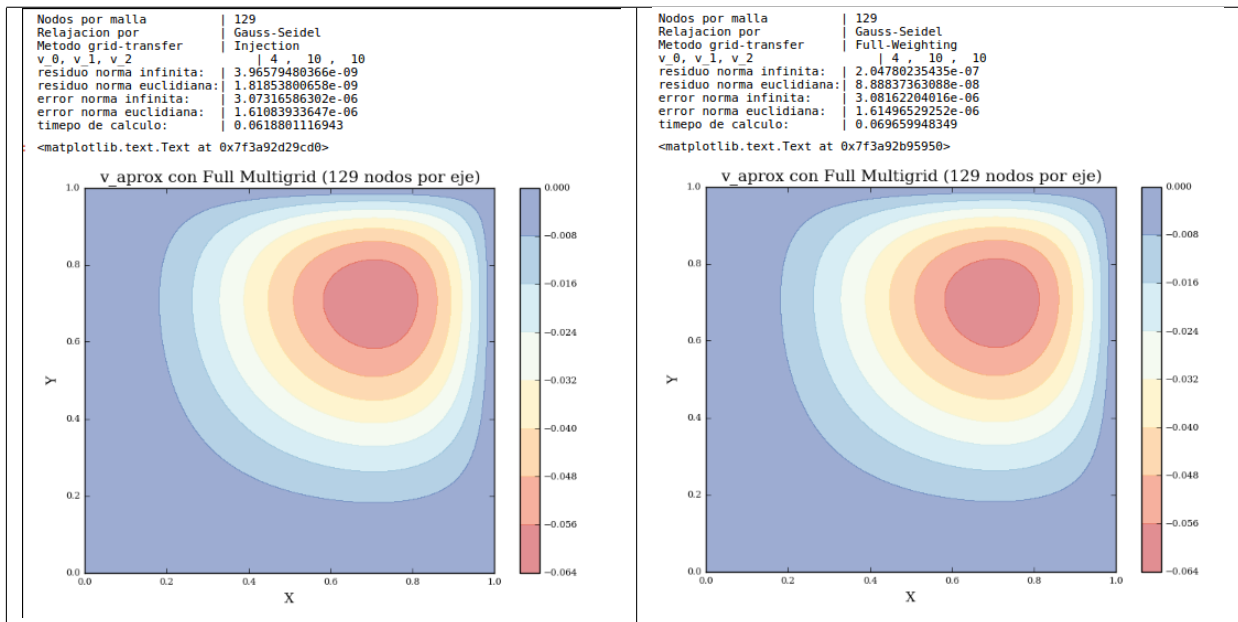


Si se cambian los valores de  $\nu$

Cuadro 4: Resultados cualitativos para *Multigrid* usando Gauss-Seidel,  $\nu_1 = 10$ ,  $\nu_2 = 10$  y  $\nu_0 = 5$



Cuadro 5: Resultados cualitativos para *Multigrid* usando Gauss-Seidel,  $\nu_1 = 10$ ,  $\nu_2 = 10$  y  $\nu_0 = 4$





### 3.2. Tiempo, Residuo y Errores

Cuadro 6: Tabla resumen para la aplicación de los métodos de relajación

Jacobi	Nodos	Tiempo [s]	Ciclos	$\ r\ _{\infty}$	$\ e\ _{\infty}$
	9	0.035	230	9.65E-09	0.0007639
	17	0.155	939	9.81E-09	0.0001967
	33	1.117	3772	9.98E-09	4.92E-05
	65	13.873	15107	9.99E-09	1.23E-05
	129	203.512	60445	1.00E-08	3.07E-06
	257	3098.59	241797	1.00E-08	7.69E-07
GS	Nodos	Tiempo [s]	Ciclos	$\ r\ _{\infty}$	$\ e\ _{\infty}$
	9	0.016	118	9.22E-09	0.0007639
	17	0.097	474	9.83E-09	0.0001967
	33	0.511	1894	1.00E-09	4.92E-05
	65	6.675	7569	9.99E-09	1.23E-05
	129	100.535	30252	1.00E-08	3.07E-06
	257	1566.40	120957	1.00E-08	7.69E-07
RB GS	Nodos	Tiempo [s]	Ciclos	$\ r\ _{\infty}$	$\ e\ _{\infty}$
	9	0.045	177	9.59E-09	0.0007638829
	17	0.1752	718	9.81E-09	0.0001967
	33	1.0191	2878	9.99E-09	4.92E-05
	65	11.0131	11517	9.99E-09	1.23E-05
	129	156.3741	46065	1.00E-08	3.07E-06
	257	2408.29	184242	1.00E-08	7.69E-07

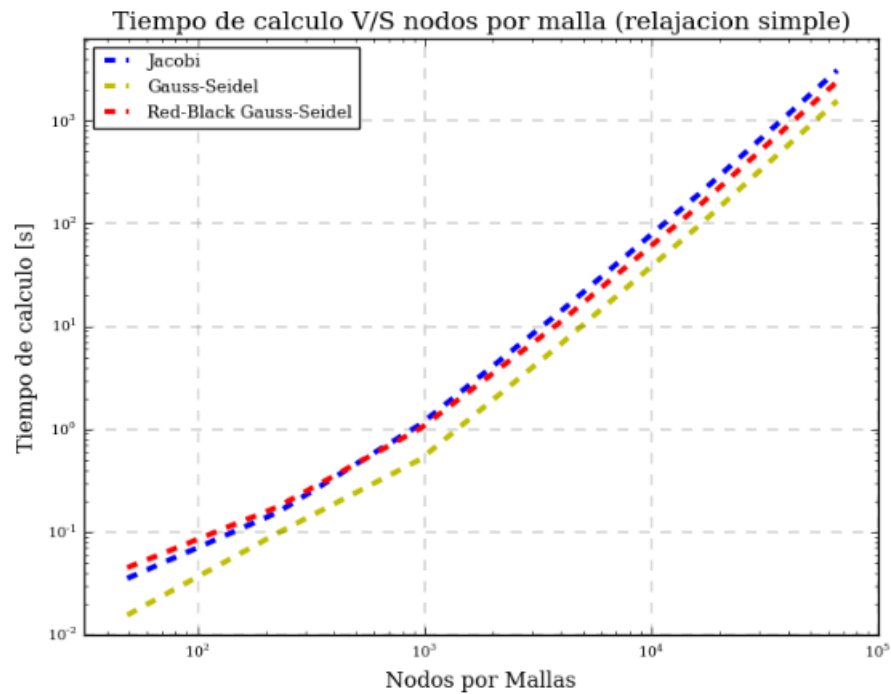


Figura 11: Complejidad algoritmica para métodos de relajación

Cuadro 7: Tabla resumen para la aplicación de v-cycle con inyección

Jacobi	Nodos	Tiempo [s]	Ciclos	$\ r\ _{\infty}$	$\ e\ _{\infty}$
	9	0.045602	49	9.91E-09	0.0007639
	17	0.183961	184	9.49E-09	0.0001967
	33	0.931226	667	9.87E-09	4.92E-05
	65	5.984544	2383	9.95E-09	1.23E-05
	129	49.432473	8382	1.00E-08	3.07E-06
	257	568.544919	28924	1.00E-08	7.69E-07
GS	Nodos	Tiempo [s]	Ciclos	$\ r\ _{\infty}$	$\ e\ _{\infty}$
	9	0.0091	8	5.55E-09	0.0007639
	17	0.00928	8	3.37E-09	0.0001967
	33	0.01050	8	3.00E-09	4.92E-05
	65	0.02366	8	2.84E-09	1,23E-05
	129	0.06263	8	2.81E-08	3,07E-06
	257	0.15729	8	2.81E-08	7.68E-07
RB GS	Nodos	Tiempo [s]	Ciclos	$\ r\ _{\infty}$	$\ e\ _{\infty}$
	9	0.012425	9	6.59E-09	0.0007638829
	17	0.028025	17	7.38E-09	0.0001967
	33	0.053658	23	9.87E-09	4.92E-05
	65	0.253512	70	8.00E-09	1.23E-05
	129	-	-	-	-
	257	-	-	-	-

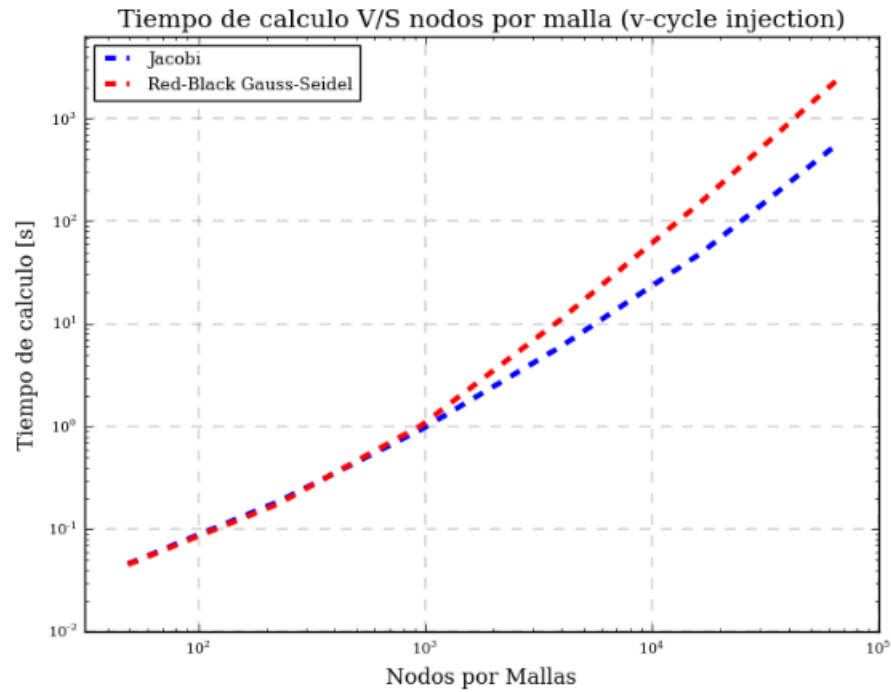


Figura 12: Complejidad algorítmica para v-cycle con inyección

Cuadro 8: Tabla resumen para la aplicación de v-cycle con Full-Weighting

Jacobi	Nodos	Tiempo [s]	Ciclos	$\ r\ _\infty$	$\ e\ _\infty$
	9	0.03758	48	7.64E-09	0.0007639
	17	0.20320	175	9.82E-09	0.0001967
	33	0.93587	631	9.96E-09	4.92E-05
	65	5.72426	2239	9.98E-09	1.23E-05
	129	49.13004	7808	9.99E-08	3.07E-06
	257	537.07808	26945	1.00E-08	7.68E-07
GS	Nodos	Tiempo [s]	Ciclos	$\ r\ _\infty$	$\ e\ _\infty$
	9	0.0110450	10	2.02E-09	0.0007639
	17	0.0134871	10	2.40E-09	0.0001967
	33	0.0178421	10	2.59E-09	4.92E-05
	65	0.0285549	10	2.64E-09	1,23E-05
	129	0.0721941	10	2.65E-08	3,07E-06
	257	0.213666	10	2.66E-08	7.68E-07
RB GS	Nodos	Tiempo [s]	Ciclos	$\ r\ _\infty$	$\ e\ _\infty$
	9	0.015742	10	7.87E-09	0.0007638829
	17	0.025217	11	2.53E-09	0.0001967
	33	0.035711	11	3.27E-09	4.92E-05
	65	0.051221	11	3.57E-09	1.23E-05
	129	0.101164	11	3.66E-09	3.07E-06
	257	0.288597	11	3.70E-09	7.68E-07

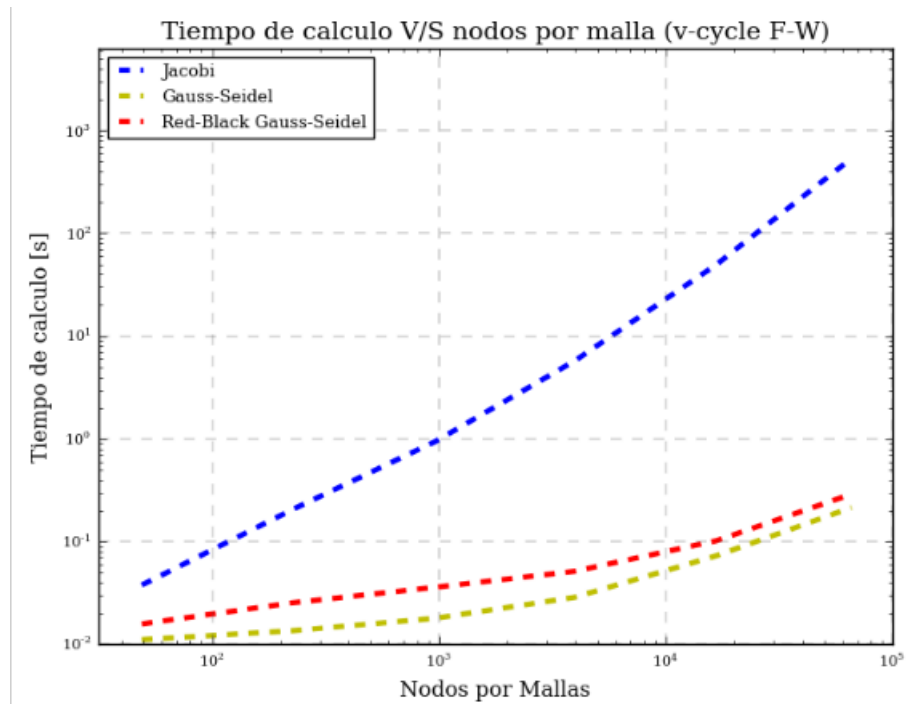


Figura 13: Complejidad algorítmica para v-cycle con Full-Weighting

Para Multigrid, debido a que Gauss-Seidel tiene el mejor comportamiento, se recogen los resultados en base a esta forma de relajar.

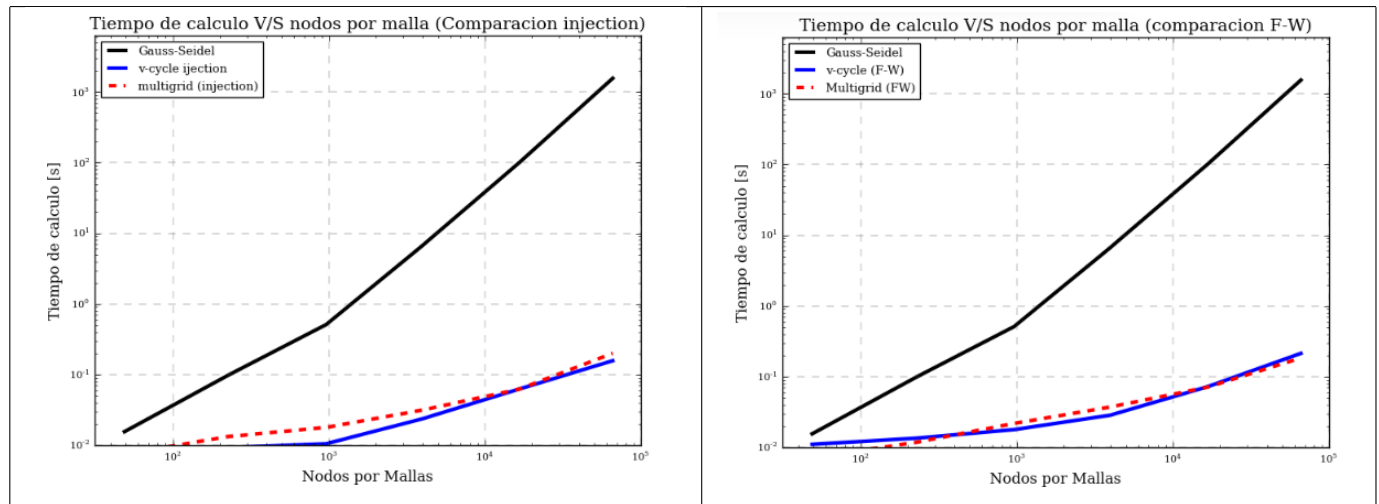
Cuadro 9: Tabla resumen para la aplicación de Multigrid  $\nu_0 = 5$ ,  $\nu_1 = 10$ ,  $\nu_2 = 10$

Full-Weighting				
GS	Nodos	Tiempo [s]	$\ r\ _\infty$	$\ e\ _\infty$
	9	0.007142	2.53E-09	0.0007639
	17	0.011761	1.55E-09	0.0001967
	33	0.021827	3.08E-09	4.92E-05
	65	0.037276	3.66E-09	1,23E-05
	129	0.071036	3.84E-08	3,07E-06
	257	0.183730	3.92E-08	7.68E-07
Injection				
GS	Nodos	Tiempo [s]	$\ r\ _\infty$	$\ e\ _\infty$
	9	0.0074651	7.17E-12	0.0007639
	17	0.0132601	2.55E-11	0.0001967
	33	0.0178909	3.25E-11	4.92E-05
	65	0.0314591	3.59E-11	1,23E-05
	129	0.0623319	3.72E-11	3,07E-06
	257	0.2000990	3.80E-11	7.68E-07

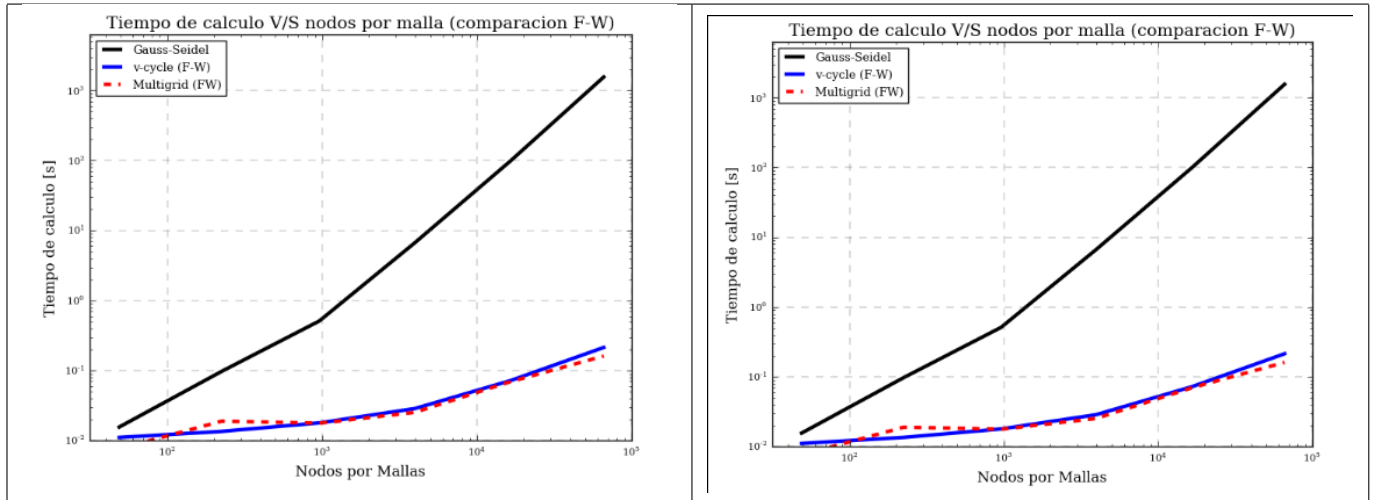
Cuadro 10: Tabla resumen para la aplicación de Multigrid  $\nu_0 = 4, \nu_1 = 10, \nu_2 = 10$

Full-Weighting				
GS	Nodos	Tiempo [s]	$\ r\ _\infty$	$\ e\ _\infty$
	9	0.00757	2.54E-09	0.0007639
	17	0.01892	8.50E-08	0.0001967
	33	0.01766	1.56E-07	4.92E-05
	65	0.02522	1.89E-07	1,23E-05
	129	0.06966	2.05E-07	3,08E-06
	257	0.16068	2.09E-07	7.77E-07
Injection				
GS	Nodos	Tiempo [s]	$\ r\ _\infty$	$\ e\ _\infty$
	9	0.004955	8.79E-10	0.0007639
	17	0.011606	3.06E-09	0.0001967
	33	0.018774	3.57E-09	4.92E-05
	65	0.034140	3.86E-09	1,23E-05
	129	0.062984	3.97E-09	3,07E-06
	257	0.152861	4.00E-09	7.68E-07

Cuadro 11: Comparación de la complejidad algorítmica de los tres casos con *Gauss – Seidel*. Multigrid con  $\nu_0 = 5, \nu_1 = 10, \nu_2 = 10$



Cuadro 12: Comparación de la complejidad algorítmica de los tres casos con *Gauss – Seidel*. Multigrid con  $\nu_0 = 4, \nu_1 = 10, \nu_2 = 10$



### 3.2.1. Comparación multi malla a iguales condiciones

Cuadro 13: Reportes para comparar v-cycle con multigrid usando inyección para  $\nu_0 = 4, \nu_1 = 10, \nu_2 = 10$

<p>Nodos por malla: 257</p> <p>Relajacion por: Gauss-Seidel</p> <p>Metodo grid-transfer: Injection</p> <p>v 1, v 2: 10, 10</p> <p>cantidad de ciclos: 4</p> <p>residuo norma infinita: 5.32941157871e-09</p> <p>residuo norma euclidiana: 2.4374809447e-09</p> <p>error norma infinita: 7.68477344433e-07</p> <p>error norma euclidiana: 4.02799951158e-07</p> <p>tiempo de calculo: 0.14604306221</p>	<p>Nodos por malla: 513</p> <p>Relajacion por: Gauss-Seidel</p> <p>Metodo grid-transfer: Injection</p> <p>v 1, v 2: 10, 10</p> <p>cantidad de ciclos: 4</p> <p>residuo norma infinita: 5.34207700298e-09</p> <p>residuo norma euclidiana: 2.44088951382e-09</p> <p>error norma infinita: 1.92271668614e-07</p> <p>error norma euclidiana: 1.00780715542e-07</p> <p>tiempo de calculo: 0.541309118271</p>	<p>Nodos por malla: 1025</p> <p>Relajacion por: Gauss-Seidel</p> <p>Metodo grid-transfer: Injection</p> <p>v 1, v 2: 10, 10</p> <p>cantidad de ciclos: 4</p> <p>residuo norma infinita: 5.35335908936e-09</p> <p>residuo norma euclidiana: 2.44179065461e-09</p> <p>error norma infinita: 4.82173373798e-08</p> <p>error norma euclidiana: 2.52758585367e-08</p> <p>tiempo de calculo: 2.34638094902</p>
<p>Nodos por malla: 257</p> <p>Relajacion por: Gauss-Seidel</p> <p>Metodo grid-transfer: Injection</p> <p>v 0, v 1, v 2: 4, 10, 10</p> <p>residuo norma infinita: 3.99722921429e-09</p> <p>residuo norma euclidiana: 1.82808912685e-09</p> <p>error norma infinita: 7.68427844113e-07</p> <p>error norma euclidiana: 4.02773107569e-07</p> <p>tiempo de calculo: 0.155963897705</p>	<p>Nodos por malla: 513</p> <p>Relajacion por: Gauss-Seidel</p> <p>Metodo grid-transfer: Injection</p> <p>v 0, v 1, v 2: 4, 10, 10</p> <p>residuo norma infinita: 4.00639743603e-09</p> <p>residuo norma euclidiana: 1.83065624514e-09</p> <p>error norma infinita: 1.92221883126e-07</p> <p>error norma euclidiana: 1.00753838021e-07</p> <p>tiempo de calculo: 0.520705938339</p>	<p>Nodos por malla: 1025</p> <p>Relajacion por: Gauss-Seidel</p> <p>Metodo grid-transfer: Injection</p> <p>v 0, v 1, v 2: 4, 10, 10</p> <p>residuo norma infinita: 4.01601665811e-09</p> <p>residuo norma euclidiana: 1.83134888977e-09</p> <p>error norma infinita: 4.816753401e-08</p> <p>error norma euclidiana: 2.52489658446e-08</p> <p>tiempo de calculo: 2.17592287064</p>

## 4. Análisis y Conclusiones

En primer lugar, comparando los gráficos de la Figura 10 con las figuras del Cuadro 1, se observa que al menos de forma cualitativa los códigos implementados para resolver *Jacobi*, *Gauss – Seidel* y *Red – BlackGauss – Seidel* son capaces de dar una buena aproximación  $v$  del campo escalar  $u$ . De el mismo modo, para  $V - cycle$ , usando *Full – Weighting* cualitativamente las relajaciones también son capaces de representar el comportamiento de  $u$  mediante aproximaciones de  $v$  para los valres de  $\nu_0$ ,  $\nu_1$  y  $\nu_2$  que se indican en el Cuadro 2, sin embargo, en dicho cuadro se puede observar que el caso en que se prueba el inyección para 129 nodos por eje, el código no es capaz de calcular los valores de  $v$  que aproximan  $u$  para *Red – Black Gauss – Seidel*.

Antes de pasar al análisis cualitativo de multigríd, se pretende contrastar lo visto anteriormente con la cuantificación de los análisis llevados a cabo anteriormente. Se observa en los cuadros 6, 7, 8 que los errores con respecto al resultado real se reducen cuando el residuo alcanza la tolerancia, esto explica y concuerda con los mapas de contorno que se observan en las figuras expuestas en lo cuadros 1 y 2 y de igual modo, para nodos mayores a 129 por eje, el calculo de *Red – Black Gauss – Seidel* no se puede llevar a cabo, o mejor dicho diverge. En la tabla de la Figura 14 extraída del libro [2] Se observa que para los distintos métodos de relajación, dependiendo de la combinación de  $\nu_1$  y  $\nu_2$  los resultados de la relajación pueden diverger. Aquí se observa que jacobi, con  $\nu_1 = 1$  y  $\nu_2 = 0$ , diverge, pero a medida que se incrementan estos valores, este recupera la convergencia para el caso en que se use inyección.

Relaxation		Injection		Full Weighting		Half-Injection	
$(\nu_1, \nu_2)$	Scheme	Linear	Cubic	Linear	Cubic	Linear	Cubic
(1,0)	Jacobi	–	–	0.49	0.49	0.55	0.62
	GS	0.89	0.66	0.33	0.34	0.38	0.37
	RBGS	–	–	0.21	0.23	0.45	0.42
	Cost	1.00	1.25	1.13	1.39	1.01	1.26
(1,1)	Jacobi	0.94	0.56	0.35	0.34	0.54	0.52
	GS	0.16	0.16	0.14	0.14	0.45	0.43
	RBGS	–	–	0.06	0.05	0.12	0.16
	Cost	1.49	1.75	1.63	1.88	1.51	1.76
(2,1)	Jacobi	0.46	0.31	0.24	0.24	0.46	0.45
	GS	0.07	0.07	0.08	0.07	0.40	0.39
	RBGS	–	–	0.04	0.03	0.03	0.07
	Cost	1.99	2.24	2.12	3.37	1.51	1.76

Figura 14: Tabla de comparación de factores de conmnvergencia extraído del libro *A Multigrif Tutorial* [2]

Lo mencionado anteriormente da pistas de que el metodo de *Red – Black Gauss – Seidel* puede llegar a converger, sin embargo, se probaron combinaciones de  $\nu_1$  y  $\nu_2$  hasta llegar a  $\nu_1 = 10$  y  $\nu_2 = 10$ , y aun así este no converge. Esto puede significar que el factor de convergencia para forma de relajar sigue siendo malo, pero no se quiso probar para un mayor numero de relajaciones ya que se hacia cada vez mas lento por lo cual carece de sentido usar dicha combinación de métodos.

```
Nodos por malla | 129
Relajacion por | Red-Black Gauss-Seidel
Metodo grid-transfer | Injection
v 1, v 2 | 10 , 10
cantidad de ciclos: | 1454
residuo norma infinita: | nan
residuo norma euclidiana: | nan
error norma infinita: | nan
error norma euclidiana: | nan
timepo de calculo: | 29.3376481533
```

Figura 15: Resporte donde se ve que el ciclo- $v$  usando *Red – Black Gauss – Seidel* aun diverge con  $\nu_1 = 10$  y  $\nu_2 = 10$

En cuanto a la complejidad algorítmica, los gráficos de las Figuras 11, 12 y 13 muestra que el orden tiempo que se demoran los calculos es de  $on^2$  y  $on \cdot \log(n)$ . Si se hace el ejercicio de predecir el tiempo para los datos del cuadro 6 es posible ver que el si la cantidad de nodos aumenta cuatro veces, el tiempo de la siguiente iteración aumenta dieciséis veces, lo cual, si bien no entrega el tiempo exacto, la predicción es una buena aproximación del tiempo que se demorara. En cuanto al resultado de los cuadros 7 y 13, la predicción de los tiempos para *Jacobi* no se cumplen, pero para *Gauss – Seidel* si y en el caso de *Red – Black Gauss – Seidel* solo se cumple si se ocupa *Full – Weighting*. De todos modos el gráfico de la Figura 12 demuestra que hay un comportamiento de tipo algoritmico para la complejidad de Jacobi, el que no se pueda predecir a partir de  $n \cdot \log(n)$  es debido a que se aprecia claramente, ya sea en los cuadros 7 y 8 que el tiempo que demora depende de la cantidad ciclos para este método de relajación, lo cual en el caso de *Gauss – Seidel* y *Red – Black Gauss – Seidel* hay indicios de que el residuo cumple con la tolerancia para una cantidad fija de ciclos, o con una variación muy leve en cuanto a la cantidad de ciclos.

Por todo lo expuesto anteriormente, se observa que el método de relajación que mejor se comporta utilizando métodos de multimalla como *V – cycle* es *Gauss – Seidel*. En base a esta conclusión el análisis para *Full – Multigrid* se hace únicamente con este tipo de relajación (y con la finalidad de no extender en demasía el informe).

Si se observan las figuras del Cuadro 3, el comportamiento cualitativo de  $v$  no se acerca al teóricamente esperado por lo cual, se concluye que las condiciones de  $\nu_0 = 1$ ,  $\nu_1 = 2$  y  $\nu_2 = 2$  no son suficientes, esto se debe a que *Multigrid* en si no realiza mas de un ciclo y dentro de ese ciclo debe hacer la cantidad de relajaciones suficiente para que  $v$  se aproxime al valor teórico. Debido a que claramente esta combinación no fue suficiente, es necesario modificar los valores  $nu$  para obtener mejores resultados.





---

## Referencias

- [1] WILLIAM L. BRIGGS, VAN EMDEN HENSON, STEVE F. MCCORMICK *Chapter 2: Basic Iterative Methos, A Multigrid Tutorial* SECOND EDITIONEDITORIAL: SIAM 15 de Noviembre de 1999, Boulder Colorado, (PAGE: 7 - 30)
- [2] WILLIAM L. BRIGGS, VAN EMDEN HENSON, STEVE F. MCCORMICK *Chapter 4: implementation, A Multigrid Tutorial* SECOND EDITIONEDITORIAL: SIAM 15 de Noviembre de 1999, Boulder Colorado, (PAGE: 58 - 85)