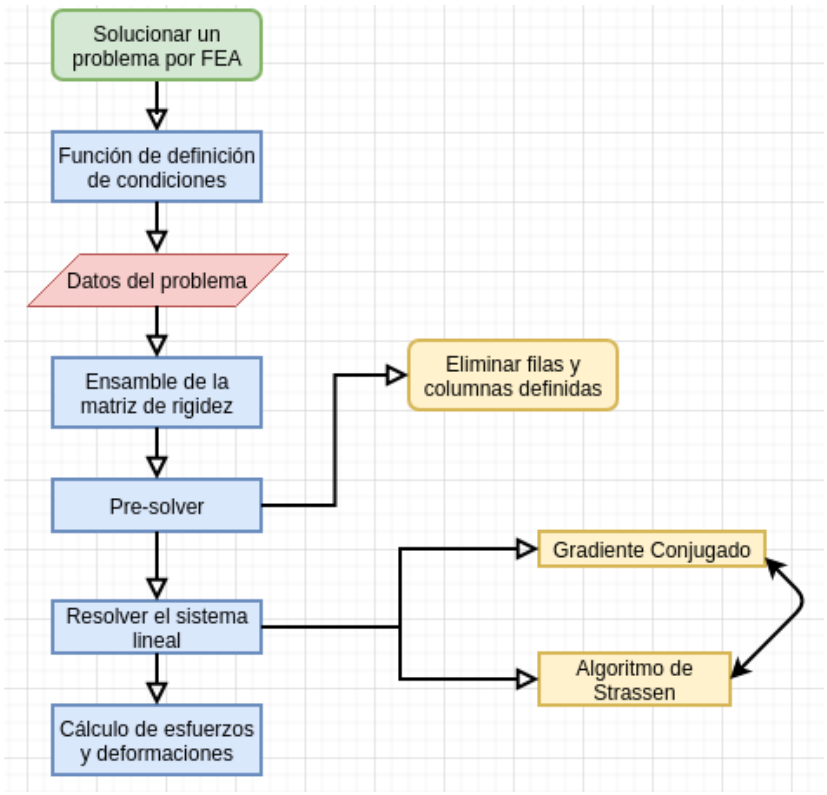
1° Práctica de Cálculo por Elementos Finitos - MC516

Josue Huaroto Villavicencio - 20174070I

Sección: E

1 Diagrama de flujo



2 Optimizadores de rendimiento

Para optimizar el código, se eligió reducir la complejidad computacional de la multiplicación de matrices y la complejidad del solucionador clásico de ecuaciones lineales.

2.1 Algoritmo de Strassen

Para mejorar la alta complejidad de la mutliplicación ingenua se utiliza el algoritmo de Strassen, la cual puede reducir la complejidad computacional al usar un enfoque Divide & Conquer y reducir la cantidad de operaciones de una matriz a 7 en vez de 8; entonces la complejidad final se reduce de $\log_2 8$ a $\log_2 7$. $\mathcal{O}(n^{2.8})$.

Código 1: Multiplicación Rápida: $\mathcal{O}(n^{\log_2 7})$

```
def FastMultiply(oldA,oldB):
    rows = oldA.shape[0]
    columns = oldB.shape[1]
    if(rows <=2 or columns<=2 or oldA.shape[1]<=2 or oldB.shape[0] <= 2):
        return np.matmul(oldA,oldB)
    A = ModifyMatrix(oldA)
    B = ModifyMatrix(oldB)
    N1 = A.shape[0]
    N2 = A.shape[1]
    N3 = B.shape[0]
    N4 = B.shape[1]
    a = A[0:N1//2,0:N2//2]
    b = A[0:N1//2,N2//2:N2//2+N2//2]
    c = A[N1//2:N1//2+N1//2,0:N2//2]
    d = A[N1//2:N1//2+N1//2,N2//2:N2//2+N2//2]
    e = B[0:N3//2,0:N4//2]
    f = B[0:N3//2,N4//2:N4//2+N4//2]
    g = B[N3//2:N3//2+N3//2,0:N4//2]
    h = B[N3//2:N3//2+N3//2,N4//2:N4//2+N4//2]
    p1 = FastMultiply(a,(f-h))
    p3 = FastMultiply((c+d),e)
    p2 = FastMultiply((a+b),h)
    p4 = FastMultiply(d,(g-e))
    p5 = FastMultiply((a+d),(e+h))
    p6 = FastMultiply((b-d),(g+h))
    p7 = FastMultiply((a-c),(e+f))
    C = np.zeros((rows,columns))
    c11 = p5 + p4 - p2 + p6
```

```
    c12 = p1 + p2
    c21 = p3 + p4
    c22 = p1 + p5 - p3 - p7
    for i in range(0,N1//2):
        for j in range(0,N4//2):
            C[i][j] = c11[i][j]
            if(j + N4//2 < columns):
                C[i][j+N4//2] = c12[i][j]
            if(i + N1//2 < rows):
                C[i+N1//2][j] = c21[i][j]
                if(j + N4//2 < columns):
                    C[i+N1//2][j+N4//2] = c22[i][j]
    return C
```

Sin embargo, la constante es demasiado alta. El algoritmo no es eficiente para tamaños pequeños, por debajo de 50 es más eficiente la multiplicación ingenua.

2.2 Gradiente conjugado

Una forma de resolver más rápido sistemas de ecuaciones lineales para matrices simétricas es utilizar el método del gradiente conjugado. La complejidad de la reducción de Cholesky o Gauss, o los métodos iterativos como los de Gauss-Seidel o Jacobi son poco eficientes para matrices grandes; el algoritmo de gradiente conjugado es capaz de reducir la complejidad de $\mathcal{O}(n^3)$ a $\mathcal{O}(n^2)$. Dado a que la mayoría de problemas lineales en elementos finitos presentan matrices de rigidez simétricas es óptimo la elección de este método para acelerar las soluciones.

Código 2: Conjugate Gradient: $\mathcal{O}(2n^2)$

```
def conjugate_grad(A, b, x=None):
    n = b.shape[0]
    if not x:
        x = np.ones((n,1))
    r = np.dot(A, x) - b
    p = - r
    r_k_norm = FastMultiply(np.transpose(r), r)
    for i in range(2*n):
        Ap = np.dot(A, p)
        alpha = r_k_norm/FastMultiply(np.transpose(p), Ap)
        x += alpha * p
        r += alpha * Ap
        r_kplus1_norm = FastMultiply(np.transpose(r), r)
        beta = r_kplus1_norm/r_k_norm
        r_k_norm = r_kplus1_norm
        p = beta * p - r
    return x
```

3 Solucionador

Para simplificar la sintaxis general de resolver un problema de elementos finitos que implique una solución directa y elementos barras se diseñó un código para separar tareas generales que permiten automatizar el código.

3.1 Definición de condiciones

Esta parte del código sirve para definir las condiciones de frontera como los soportes o fuerzas nodales.

Código 3: Condiciones de frontera

```
NodesCondition = []
ForcesCondition = []
def UBoundaryCondition(nU,u,i):
    nU[i][0] = u
    NodesCondition.append(i)
def FBoundaryCondition(nF,f,i):
    nF[i][0] = f
    ForcesCondition.append(i)
```

3.2 Ensamble de matriz de rigidez

Con los datos del problema se crea una matriz de rigidez simétricas ensamblada que se utilizará para resolver el problema.

Código 4: Ensamble de matriz de rigidez

```
def AssemblyStiffness(nStiffnessMatrix,k,i,j):
    nStiffnessMatrix[i][i] += k
    nStiffnessMatrix[i][j] += -k
    nStiffnessMatrix[j][i] += -k
    nStiffnessMatrix[j][j] += k
def Initialize(nStiffnessMatrix,nU,nF):
    for i in range(0,NumberOfElement):
        AssemblyStiffness(nStiffnessMatrix,K[i][0],int(K[i][1]),int(K[i][2]))
```

3.3 Preparación de la matriz de rigidez

Se elimina filas y columnas de la matriz de rigidez que permita despejar el sistema y dejar una forma equivalente a $\mathbf{A}x = \mathbf{B}$.

Código 5: Preparación de matriz de rigidez

```
def PreSolvingStiffness(nStiffnessMatrix):
    nsize = Nodes-len(NodesCondition)
    newStiffness = np.zeros((nsize,nsize))
    contr = -1
    for i in range(0,Nodes):
        contc = -1
        flagr = False
        for k in range(0,len(NodesCondition)):
            if(i == NodesCondition[k]):
                flagr = True
                break
        if(flagr):
            continue
        contr += 1
        for j in range(0,Nodes):
            flagc = False
            for k in range(0,len(NodesCondition)):
                if(j == NodesCondition[k]):
                    flagc = True
                    break
            if(flagc):
                continue
            contc += 1
            newStiffness[contr][contc] = nStiffnessMatrix[i][j]
    return newStiffness
```

3.4 Pre-solver de la matriz de fuerzas

Código 6: Pre-solver de la matriz de fuerzas

```
def PreSolvingF(nF,nS,nU):
    nsize = Nodes-len(NodesCondition)
    newF = np.zeros(nsize).reshape(nsize,1)
    contr = -1
    for i in range(0,Nodes):
        flagr = False
        for k in range(0,len(NodesCondition)):
            if(i == NodesCondition[k]):
                flagr = True
                break
        if(flagr):
            for k in range(0,Nodes):
                nF[k][0] = nF[k][0]-nS[k][i]*nU[i][0]
            continue
    for i in range(0,Nodes):
        flagr = False
        for k in range(0,len(NodesCondition)):
            if(i == NodesCondition[k]):
                flagr = True
                break
        if(flagr):
            continue
        contr += 1
        newF[contr][0] = nF[i][0]
    return newF
```

3.5 Solver

Se utiliza el método del gradiente conjugado para resolver el sistema despejado, finalmente; se reemplaza en las matrices originales y se usa el algoritmo de Strassen para armar la solución final.

Código 7: Solver

```
def Solve(nStiffnessMatrix,nU,nF):
    newStiffness = PreSolvingStiffness(nStiffnessMatrix)
    newF = PreSolvingF(nF,nStiffnessMatrix,nU)
    u = conjugate_grad(newStiffness,newF)
    contr = -1
    for i in range(0,Nodes):
        flagr = False
        for k in range(0,len(NodesCondition)):
            if(i == NodesCondition[k]):
                flagr = True
                break
```

```
        if(flagr):
            continue
        contr += 1
        nU[i][0] = u[contr][0]
    nnF = FastMultiply(StiffnessMatrix,nU)
    return nU,nnF
```

4 Ejecución del código

Las condiciones del problema se establecen en esta sección del código, aquí se definen las condiciones de frontera, características geométricas y físicas y fuerzas aplicadas; así como la cantidad de nodos y elementos para la solución.

Código 8: Condiciones del problema

```
NodesCondition = []
Nodes = 4
Nodes = 2*Nodes+1
E = 2*1e5 #MPA
L = 1500 #mm
b0 = 1000 #mm
bf = 0 #mm
e = 120 #mm
P = 50000 #N
y = 0.0784532e-3 #N/mm3
A = []
K = []
for i in range(0,Nodes):
    b = b0-i*((b0-bf)/(Nodes-1))
        nb = b-(b0-bf)/(Nodes-1)
    A.append((b+nb)*e/2)
for i in range(0,Nodes-1):
    K.append([E*A[i]/(L/(Nodes-1)),i,i+1])
NumberOfElement = len(K)
StiffnessMatrix = np.zeros((Nodes,Nodes))
U = np.zeros(Nodes).reshape(Nodes,1)
F = np.zeros(Nodes).reshape(Nodes,1)
Initialize(StiffnessMatrix,U,F)
UBoundaryCondition(U,0,0)
for i in range(1,Nodes):
    if(i%2 == 1):
        W = y*(A[i]+A[i-1])*(L/(Nodes-1))
        FBoundaryCondition(F,W,i)
    else:
        FBoundaryCondition(F,0,i)
if((NumberOfElement//2)%2 == 1):
    W = y*(A[NumberOfElement//2]+A[NumberOfElement//2-1])*(L/(Nodes-1))
    FBoundaryCondition(F,P+W,NumberOfElement//2)
else:
    FBoundaryCondition(F,P,NumberOfElement//2)
U,F=Solve(StiffnessMatrix,U,F)
print("Stiffness Matrix:\n",StiffnessMatrix,'\n')
print("Displacements:\n",U,'\n')
print("Forces:\n",F)
```

La gráfica de la fuerza sobre los nodos se muestra a continuación, se puede apreciar que existen unos pequeños picos para los primeros elementos; esto es porque corresponden al peso, luego para la mitad se ve un pico más pronunciado que corresponde a la carga central; luego para los últimos elementos se observa que el peso es prácticamente insignificante para la parte final de la placa triangular en comparación con la base.

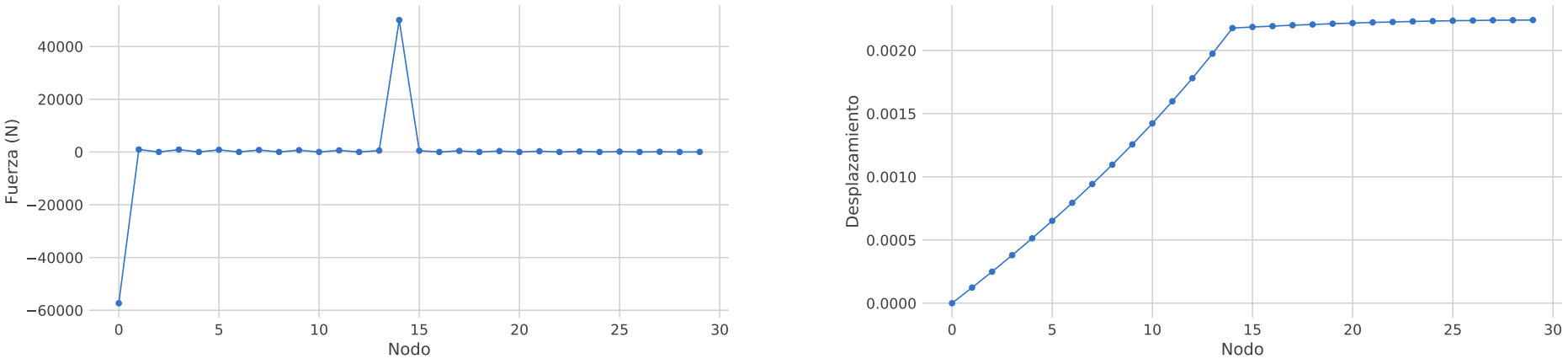


Figura 1: Fuerza y desplazamiento para 30 nodos

La gráfica de la deformación muestra un comportamiento lineal hasta el nodo donde se aplica la carga externa; para luego cambiar de pendiente.

5 Resultados del problema

El problema se modela de la siguiente forma: dos elementos se representan con 3 nodos, y el nodo central es utilizado para colocar el peso de los dos elementos; de esta forma el peso de los dos elementos es colocado en el nodo medio como una carga externa única para ese nodo; dado que dos elementos consecutivos se diferencian en un volumen muy pequeño el nodo medio representa al centro de gravedad de los dos elementos juntos. Con esta formulación se duplica la cantidad de elementos pero es posible modelar la placa como una unión simple de elementos barra.

```
Stiffness Matrix:
 [[ 56000000. -56000000.         0.         0.         0.]
 [-56000000.  96000000. -40000000.         0.         0.]
 [        0. -40000000.  64000000. -24000000.         0.]
 [        0.         0. -24000000.  32000000.  -8000000.]
 [        0.         0.         0.  -8000000.   8000000.]]

Displacements:
 [[0.        ]
 [0.00101894]
 [0.00231307]
 [0.00238662]
 [0.00238662]]

Forces:
 [[-5.70607880e+04]
 [ 5.29559100e+03]
 [ 5.00000000e+04]
 [ 1.76519700e+03]
 [-2.04818207e-09]]

Esfuerzos (MPa):
 [[0.76081051]
 [0.98130351]
 [0.0504342 ]
 [0.02941995]]
```

Figura 2: Matriz de rigidez, fuerzas, desplazamientos y esfuerzos para 4 elementos

La fuerza de reacción en el apoyo corresponde a la fuerza en el nodo 0: -57060.788 N

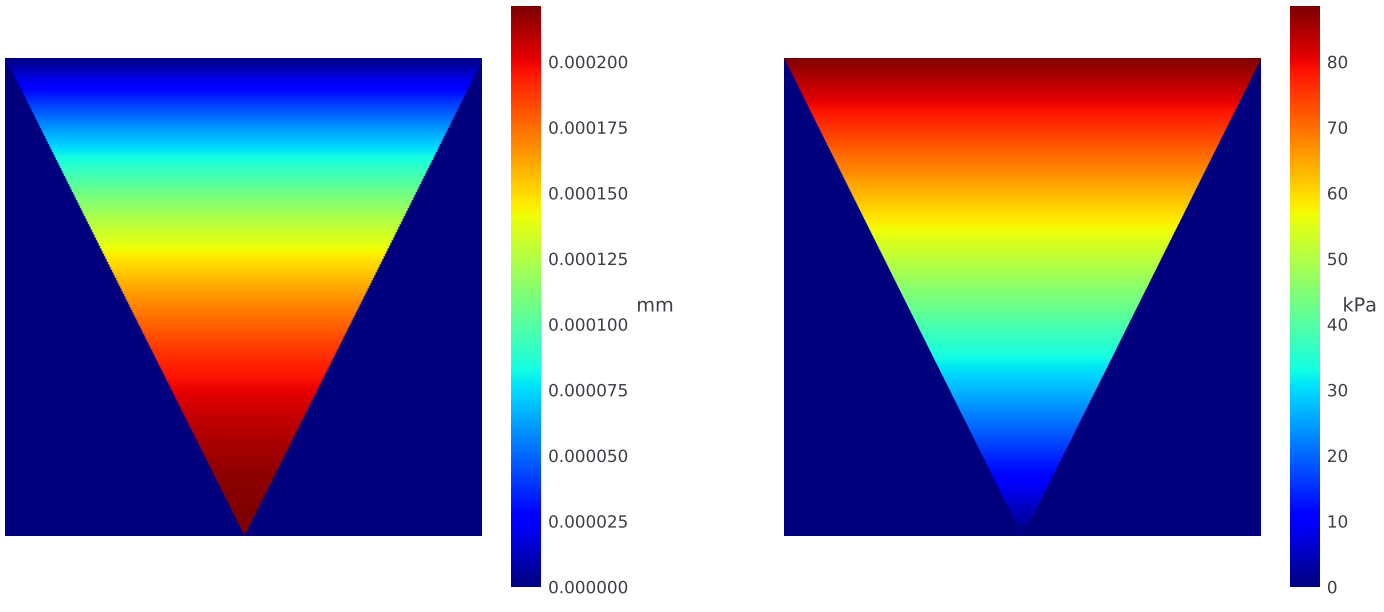


Figura 3: Distribución de deformaciones y esfuerzos considerando solo el peso de la placa

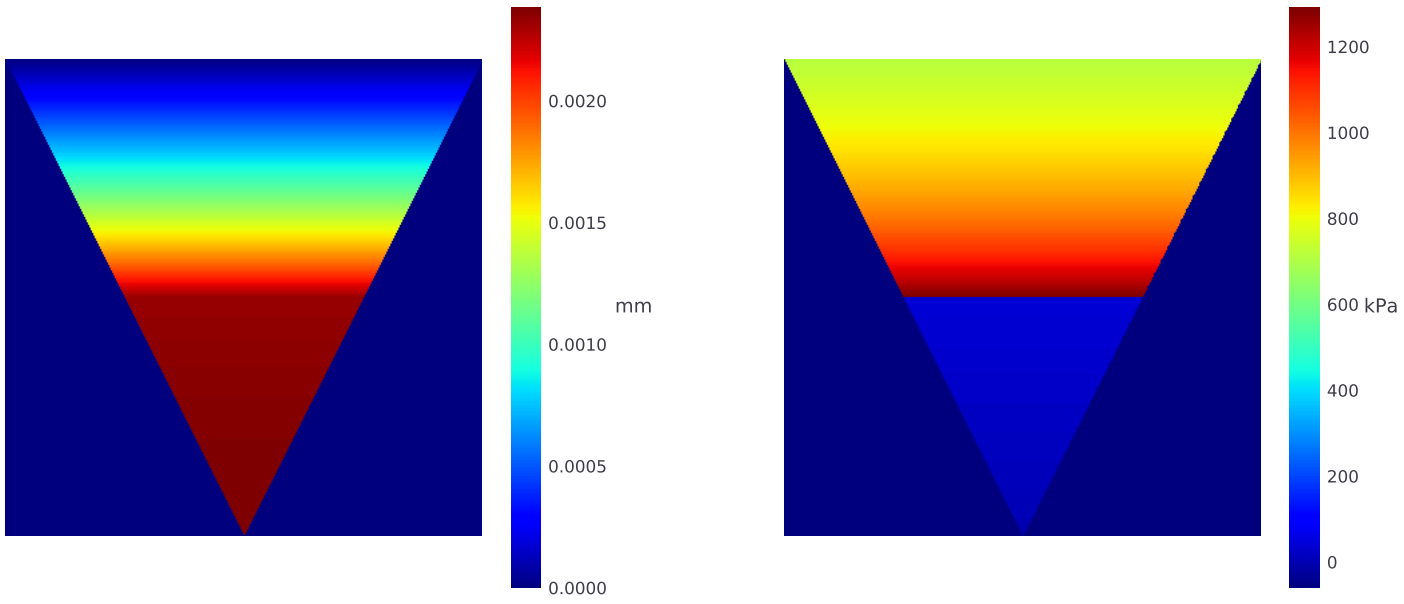


Figura 4: Distribución de deformaciones y esfuerzos de la placa

6 Problema generalizado para n nodos

```
Stiffness Matrix:
 [[ 1.52e+08 -1.52e+08  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00
   0.00e+00  0.00e+00  0.00e+00  0.00e+00]
 [-1.52e+08  2.88e+08 -1.36e+08  0.00e+00  0.00e+00  0.00e+00  0.00e+00
   0.00e+00  0.00e+00  0.00e+00  0.00e+00]
 [ 0.00e+00 -1.36e+08  2.56e+08 -1.20e+08  0.00e+00  0.00e+00  0.00e+00
   0.00e+00  0.00e+00  0.00e+00  0.00e+00]
 [ 0.00e+00  0.00e+00 -1.20e+08  2.24e+08 -1.04e+08  0.00e+00  0.00e+00
   0.00e+00  0.00e+00  0.00e+00  0.00e+00]
 [ 0.00e+00  0.00e+00  0.00e+00 -1.04e+08  1.92e+08 -8.80e+07  0.00e+00
   0.00e+00  0.00e+00  0.00e+00  0.00e+00]
 [ 0.00e+00  0.00e+00  0.00e+00  0.00e+00 -8.80e+07  1.60e+08 -7.20e+07
   0.00e+00  0.00e+00  0.00e+00  0.00e+00]
 [ 0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00 -7.20e+07  1.28e+08
  -5.60e+07  0.00e+00  0.00e+00  0.00e+00]
 [ 0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00 -5.60e+07
   9.60e+07 -4.00e+07  0.00e+00  0.00e+00]
 [ 0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00
  -4.00e+07  6.40e+07 -2.40e+07  0.00e+00]
 [ 0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00
   0.00e+00 -2.40e+07  3.20e+07 -8.00e+06]
 [ 0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00
   0.00e+00  0.00e+00 -8.00e+06  8.00e+06]]

Displacements:
 [[0.        ]
 [0.0003754 ]
 [0.00077627]
 [0.0012306 ]
 [0.00173581]
 [0.00233288]
 [0.00234857]
 [0.00236874]
 [0.0023758 ]
 [0.00238757]
 [0.00238757]]

Forces:
 [[-5.70607880e+04]
 [ 2.54188368e+03]
 [ 2.95403879e-08]
 [ 1.97702064e+03]
 [ 2.36032065e-08]
 [ 5.14121576e+04]
 [ 4.92436811e-08]
 [ 8.47294560e+02]
 [-7.85803422e-10]
 [ 2.82431520e+02]
 [ 4.43833414e-10]]
```

Figura 5: Matriz de rigidez, fuerzas y desplazamientos para 11 nodos

Para incrementar la precisión de los resultados se elige utilizar una mayor cantidad de elementos, debido a las limitaciones de memoria, usaremos solo 2000 elementos; el cual se logra ejecutar en python en alrededor de 1 segundo.

	Deformaciones (mm)	Fuerzas (N)	Esfuerzo (MPa)
0	0.000000	-5.706079e+04	0.713349
1	0.000002	-2.653345e-06	0.713618
2	0.000004	4.722824e-06	0.713887
3	0.000005	-7.797033e-06	0.714156
4	0.000007	-9.028765e-06	0.714425
...	...	...	...
1996	0.002387	-7.244525e-07	0.000188
1997	0.002387	-4.329195e-07	0.000144
1998	0.002387	-3.898458e-07	0.000101
1999	0.002387	3.676541e-07	0.000059
2000	0.002387	7.780072e-07	0.000000

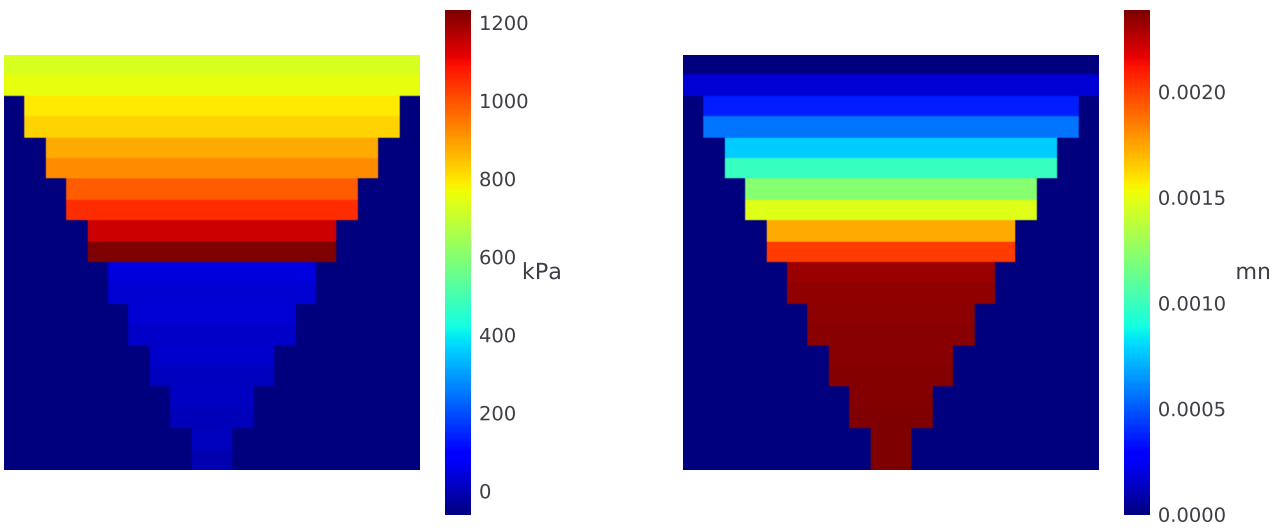


Figura 6: Deformaciones, fuerzas y esfuerzos para 2000 elementos - Discretización con 20 elementos

7 Verificación de resultados

Para la verificación de los cálculos se utilizó el software Ansys.

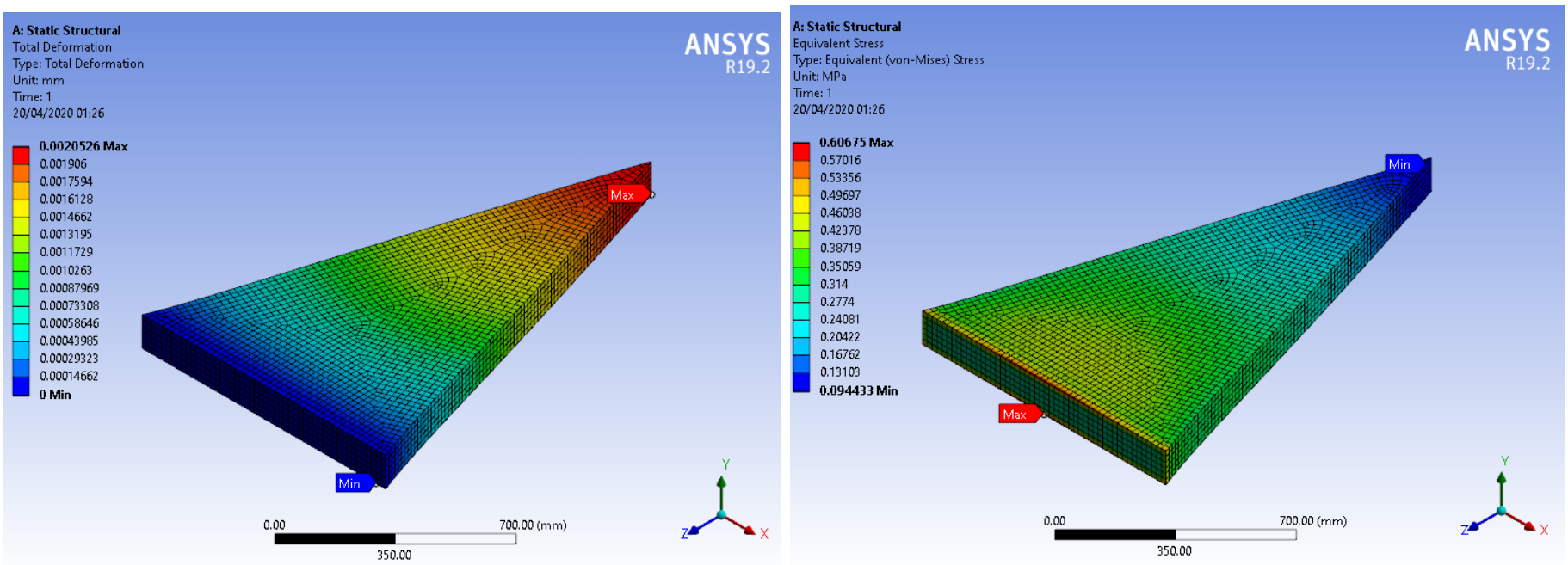


Figura 7: Deformaciones y esfuerzos

8 Conclusiones

1. Es posible despreciar el peso de la placa porque es muy pequeña la fuerza en comparación con la carga central.
2. La cantidad de elementos para la convergencia de la solución es de aproximadamente 50.
3. Incrementar la cantidad de elementos puede parecer una estrategia viable para mejorar la convergencia de la solución; pero al incrementar la cantidad de elementos sus dimensiones son menos aceptables para ser tratado como un elemento barra por sus medidas geométricas.
4. Se sugiere usar un elemento de mayor dimensión para poder modelar de forma más precisa los esfuerzos y deformaciones.
5. La estrategia de ubicar el peso en un nodo medio entre dos elementos es una estrategia que permite acercarnos a la respuesta cuando es necesario ubicar cargas sobre un continuo.
6. El modelado hecho con elemento barra cumple los criterios debido a que la deformación de la placa en otros ejes es despreciable y se considera únicamente deformación en una dirección.
7. El algoritmo de Strassen junto con el algoritmo del gradiente conjugado son capaces de acelerar hasta 10 veces el código para matrices de gran tamaño.
8. El tiempo esperado por solución es de a lo mucho 10 segundos para 10000 elementos.
9. La implementación del código en Python es lo suficientemente rápido y corto para realizarse en unos pocos minutos.
10. La implementación del código en MatLab es más simple y corta pero demasiado lenta; tardando varios minutos para ejecutar 1000 elementos.

Lenguaje/Cantidad de elementos	Tiempo de ejecución (s)
C++/1000	0.3
Python/1000	4.2
MatLab/1000	357.7

9 Agradecimientos

El presente trabajo me permitió desarrollar y aplicar gran parte de mi conocimiento en el desarrollo y optimización de algoritmos; como programador competitivo constantemente intento optimizar mis algoritmos lo más posible y en este caso no fue la excepción. El código permite una generalización para cualquier problema que pueda ser modelado con barras con tan solo unas pocas modificaciones en la parte final del código; como tal puede tratarse como un solver eficiente para elementos finitos. Así mismo, espero que este sea uno de las primeras implementaciones que realice para poder tener un repertorio más amplio de elementos en el futuro.
El trabajo inicial fue muy simple de realizar, pero en busca de optimizar el código me vi en la necesidad de recurrir al algoritmo de Strassen; sin embargo, aún necesitaba de un solver que utilizara de forma eficiente las multiplicaciones hechas por el algoritmo; el método del gradiente conjugado trabaja mejor que Gauss-Seidel para matrices simétricas y grandes; por lo que fue la elección para resolver el sistema.
El código en Python era lo suficientemente rápido como para computar más de 2000 elementos en cuestión de segundos; pero la implementación en C++ no; sucede que el gradiente conjugado trabaja multiplicaciones de un problema algorítmico llamado Matrix-Vector multiplication, siendo la complejidad $\mathcal{O}(n^3)$; dicho problema se encuentra optimizado a través de numpy al trabajar los arrays como una estructura llamada SparseMatrix, haciendo que python sea 20 veces más rápido que C++. Finalmente, el trabajo me permitió conocer sobre los distintos métodos de optimizaciones para el problema de Matrix-Vector multiplication, por ejemplo: Mailman algorithm, Fourier Matrix & FFT, así como de bibliotecas dedicadas a cálculos numéricos en C++ como BLAS. Para mejorar el tiempo de ejecución de C++ se utilizó una libreria de SparseMatrix similar a numpy; los resultados hacen que C++ sea ahora 500 veces más rápido que antes. La estructura de SparseMatrix aprovecha el hecho que la mayor parte de elementos en la matriz son 0, de hecho más del 70% de la matriz es así. Como tal, el código en C++ ahora es capaz de ejecutar hasta un millón de elementos en solo 10 minutos.
Se eligió C++ y Python como principales lenguajes para el desarrollo del código debido a su óptimo rendimiento de ejecución; finalmente, el código de MatLab es solo un port de los códigos anteriores; sin embargo podrá notar que aunque el algoritmo es el mismo, el tiempo de ejecución en MatLab es muy alto, llegando a ser de varios minutos solo para 1000 elementos.

Referencias

[1] Optimized methods in FEM:
https://www.sciencedirect.com/topics/engineering/gauss-seidel-method

[2] Succesive over relaxation:
https://en.wikipedia.org/wiki/Successive_over-relaxation

[3] Strassen Algorithm:
https://www.sciencedirect.com/science/article/pii/0898122195002162

[4] Sparse Matrix:
https://en.wikipedia.org/wiki/Sparse_matrix

[5] Sparse Matrix Library:
https://github.com/uestla/Sparse-Matrix

[6] Mailman algorithm:
http://www.cs.yale.edu/homes/el327/papers/matrixVectorApp.pdf

[7] Fast Algorithms with Preprocessing for Matrix-Vector Multiplication Problems:
https://www.sciencedirect.com/science/article/pii/S0885064X84710211