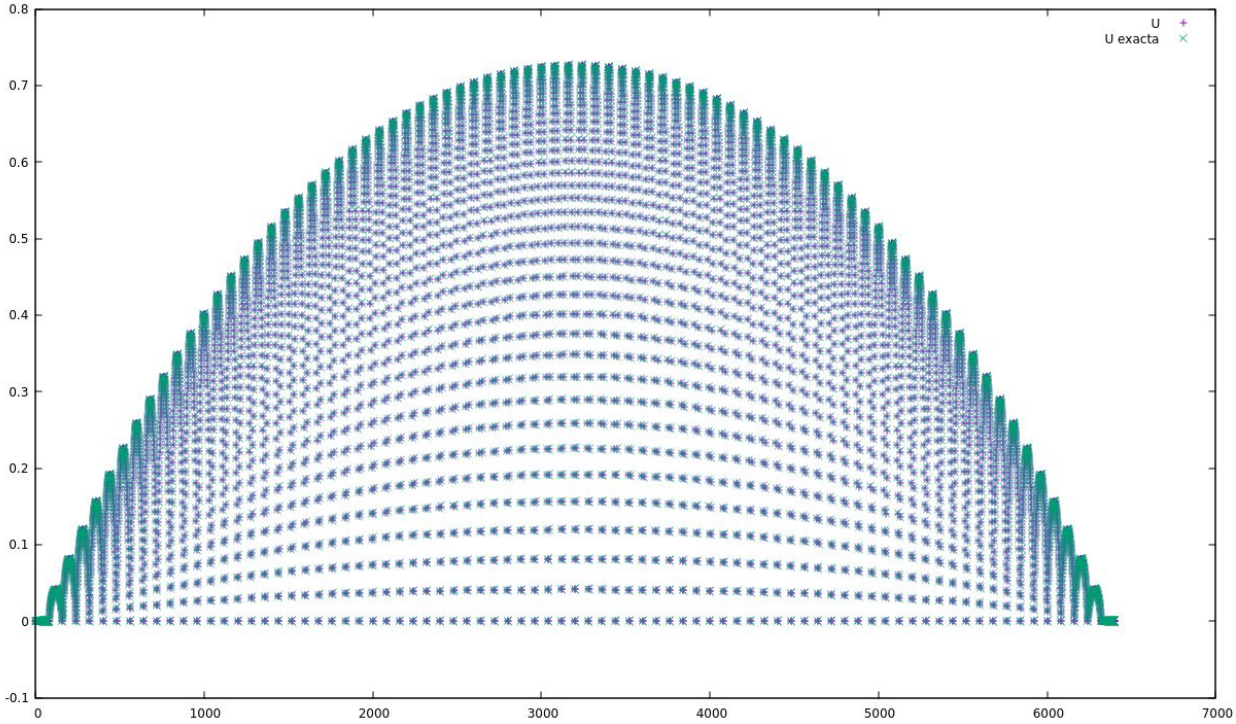


Aplicaciones de métodos numéricos para resolución de dos ecuaciones.

Grupo B
Barbero Angulo, Vicente 15502793B
García-Tenorio Fernández, Sergio 05455863X
Esteban Gracia, Javier 26280026L
Monje Lola, Oscar Ángel 49100617-X
de Lanzas García, Carlos 21003452-J

Formulación de los problemas a resolver

Para este trabajo vamos a explorar la implementación de un método de resolución de ecuaciones lineales iterativo que resuelve sistemas donde la matriz que introducimos es simétrica y definida positiva. Esto nos va a venir muy bien ya que también se nos plantea la resolución de una ecuación diferencial ordinaria y una en derivadas parciales por métodos numéricos.

Sistemas de resolución de ecuaciones como gauss tienen el problema de que conforme aumentamos más y más el sistema de ecuaciones, que es lo que vamos a querer para resolver estos problemas, gauss se vuelve bastante ineficiente.

Es por este motivo que se vuelve necesario para resolver estos sistemas de manera más precisa implementar un método de resolución como el de el gradiente conjugado, muy ventajoso para solucionar nuestros problemas.

Estos problemas que vamos a solucionar se tratan por una parte de la búsqueda de soluciones para esta ecuación diferencial ordinaria:

$$x^2\frac{d^2u}{dx^2} + x\frac{du}{dx} + x^2u = 0$$

Teniendo en cuenta las condiciones que se nos indican, que en el punto 1 la función tenga como valor el 0 y que en 2 tenga como valor el 1.

$$u(1) = 0 \qquad u(2) = 1$$

Por otro lugar se nos plantea la resolución de otro problema, la resolución de la ecuación de Poisson, que nos aparece en la electrostática por ejemplo y encontrar soluciones precisas bajo ciertas condiciones de contorno es bastante útil para los diseñadores de circuitos electrónicos por ejemplo que quieren un diseño completamente optimizado, que para ciertas aplicaciones puede resultar crítico[1-Nagel].

Este es el problema que se nos plantea, la resolución de esta ecuación para un contorno que nos proporcionan. El lector avispado podrá ver rápidamente que ahora en lugar de una variable nos estamos moviendo en dos, esto tendrá como consecuencia algunas diferencias respecto al método que explicamos anteriormente.

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = 1 \qquad 0 < x < \pi \quad , \quad 0 < y < \pi$$

$$\begin{aligned} u(0,y) &= 0 \quad \text{para} \quad 0 \le y \le \pi \\ u(\pi,y) &= 0 \quad \text{para} \quad 0 \le y \le 1\pi \\ u(x,0) &= 0 \quad \text{para} \quad 0 < x < \pi \\ u(x,\pi) &= 0 \quad \text{para} \quad 0 < x < \pi \end{aligned}$$

Explicación de la estrategia de resolución e implementación de esta

En primer lugar vamos a explorar qué es lo que haría el método del gradiente conjugado.

El gradiente conjugado lo veremos en una subrutina. Esta tiene como propósito dar la opción de resolver complejos sistemas de ecuaciones. Este subprograma que estará recogido en el módulo sistemas.

Consiste en que a través de un número “nmax” de iteraciones, calculará un itinerante a partir de su anterior, con una dirección de descenso (v).

A través de estas iteraciones, se calculara el residuo, que irá disminuyendo a lo largo de esta dirección de descenso.

Lo primero que se calculará en cada iteración será el paso (tao), con este, se calculara el nuevo itinerante y a continuación, el nuevo residuo (r).

Cuando el residuo sea mínimo(menor que la norma de b, multiplicada por la tolerancia), el programa parará, y tendremos nuestro itinerante, el cual, será la solución de nuestro sistema de ecuaciones.

Si este no fuera el caso, y el itinerante no fuera mínimo, se calcularía un parámetro(s), a partir del cual hallaríamos la nueva dirección de descenso y, con esta, se repetirá el proceso.

Lo podemos ver implementado y comentado en nuestro código.

```
v=-r1           !direccion inicial de descenso
do i=1,nmax
    t= reshape(v,[1,n]) !vector traspuesto a v para el calculo del paso(tao)
    tao = real(norma(r1)**2)/real(norma(matmul(matmul(t,A),v))) ! calculo del paso
    x= x + tao*v !calculo del nuevo itinerante para cada ciclo
    r=r1 !cambio de variable ya que al calcular el parametro s, se necesitan tanto r1 como su anterior; r
    r1 = r + tao*matmul(A,v) !calculo del nuevo residuo
    if (norma(r1)<tol*norma(b)) EXIT ! condicion puesta para asegurar exactitud en los resultados
    s=(norma(r1)**2)/(norma(r)**2) ! calculo del parametro s
    v=(-1)*r1 + (s*v) !calculo de la nueva direccion de descenso
end do
```

Para resolver la diferencial ordinaria se ha seguido la siguiente estrategia: En primer lugar lo que se ha hecho es generar un vector de puntos equiespaciados en el entorno que queremos evaluar la diferencial, aunque que sean espaciados no es estrictamente necesario.

De esta forma ya tenemos nuestro vector x dividido en muchos trocitos, una vez tenemos este lo que vamos a hacer es tomar las derivadas obtenidas por los métodos numéricos,

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_i = \frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2} + \mathcal{O}(\Delta x)^2 \qquad \frac{u_{i+1} - u_{i-1}}{2\Delta x}$$ [2-Kuzmin]

y para cada caso tendremos que evaluar un entorno de 3 puntos. Es en estos puntos que luego tenemos que despejar las u-1,u,u+1 en nuestro sistema de ecuaciones, que deberemos generar.

```
subroutine invoca_matvectdif(m,r,x,np,h,a,alpha,beta)          !Crea la matriz y el vector que usaremos para solucionar la
        integer, intent(in) :: np                              !diferencial ordinaria
        real(8), intent(in) :: h,alpha,beta,a
        real(8), intent(inout) :: m(:,:),r(:),x(:)
        integer :: i
        m=0.0d0
        m(1,1)=1.0d0
        m(np,np)=1.0d0                                         !Introducimos en la matriz los 1 correspondientes a nuestro contorn

        do i=1,np
        x(i)=a+(real(i)-1.0d0)*h                               !Generamos el vector x equiespaciado para resolver el problema
        end do

        do i=2,np-1
        m(i,i-1)=(x(i)*x(i))-x(i)*(h/2.0d0)                    !Establecemos las ecuaciones correspondientes a haber hecho la
        m(i,i)=-2.0d0*(x(i)*x(i))+((x(i)*x(i))*(h*h))          !derivación numérica, que contiene las relaciones correspondientes
        m(i,i+1)=(x(i)*x(i))+x(i)*(h/2.0d0)
        r(i)=0                                                 !Como nos dicen que el vector resultado es 0 pues lo imponemos
        end do
        r(1)=alpha
        r(np)=beta                                             !Alpha y beta son los dos valores que nos dicen
end subroutine invoca_matvectdif
```

La estrategia de implementación correspondiente a este código ha consistido en básicamente con una subrutina que lo que hace es generar nuestra matriz y vector resultados, que podemos ver en la imágen nos va poniendo los valores que corresponden en la matriz. Hemos tomado como que las soluciones que nos dan, ponerlas como soluciones triviales del sistema de ecuaciones en vez de simplemente definirlas.

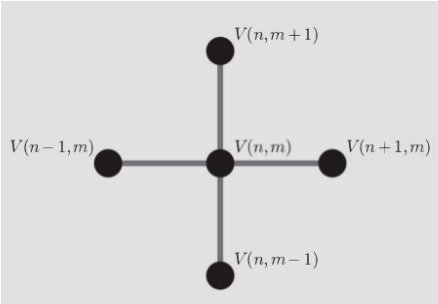
Para crear la matriz simplemente se han puesto los datos de u-1, u, u+1 en cada fila, quedando una matriz simétrica y de esta forma ir sacando todos los valores.

```
np=100
alpha=0.0d0
beta=1.0d0
b=2.0d0
a=1.0d0
h=(b-a)/(np-1)
allocate(x(np),m(np,np),u(np),r(np))

call invoca_matvectdif(m,r,x,np,h,a,alpha,beta)        !Llama a la subrutina que crea matriz, vector resultado y valores d
call gausspivote(besselmatrix,val,coeficientes)        !resuelve el sistema que hemos creado, ya tenemos coeficientes para
                                                       !aproximar nuestra funcion.

call grad_conj(m,r,u,1.0d-6,100)                       !Resuelve el sistema para hallar nuestra aproximacion numérica
```

Después simplemente se resuelve este sistema normalmente

El problema de Poisson es muy similar al anterior, la estrategia de resolución por lo tanto va ser equivalente, con la diferencia de que en lugar de tener una variable vamos a tener dos y si antes teníamos puntos equiespaciados en la x ahora vamos a tener puntos equiespaciados en x e y, de esta forma el número de incógnitas que necesitamos es el número de puntos equiespaciados de cada eje al cuadrado.

Dicho esto lo demás es muy similar, en primer lugar lo que debemos hacer es como en el anterior, usando los métodos de derivación numérica. El principio es igual solo que vamos a ir tomando cruces, de la manera siguiente debido a que nos encontramos en dos variables.



|[1-Nagel]

El contorno que nos dan serán los bordes, formando un cuadrado, y los puntos internos serían como si fuese un plano.

Dicho esto, la solución es ir despejando con cuidado, despejando nos damos cuenta que la matriz tiene que tener, para este problema en concreto ya que no nos aparecen x ni y delante de los diferenciales siempre en cada fila centrándonos en la diagonal de nuestra matriz tenemos siempre un u-np,u-1,u,u+1,u+np.

Esto es fácil de ver si tomas un punto cualquiera y haces la cruz, siempre tienes un u-1,u,u+1 en horizontal y luego las ligaduras en vertical si numeramos nuestros puntos uno a uno para la fila anterior le tendremos que restar el número de puntos de cada fila o columna, al haber el mismo número de filas y de columnas en nuestros puntos equiespaciados. Esto no sería cierto si no hubiéramos tomado el mismo valor de equi espaciamiento en x y en y.

En el código que genera la matriz lo que se ha hecho es en primer lugar un bucle que nos colocaría la matriz los resultados que ya sabemos correspondientes a la primera y última fila del contorno para que nos los resuelva automáticamente, por otra parte también va rellenando estos valores para el vector respuesta.

Siguiendo esta idea de tomar simplemente los índices de cada uno de los puntos para sacar nosotros el valor del índice de las columnas lo que se va a hacer es ir saltando cada vez un np y coger los dos puntos correspondientes al contorno, esto se puede ver en el bucle.

```
subroutine invoca_matvect(m,r,h,np) !Invoca la matriz y el vector resultado del problema de poisson
       integer, intent(in) :: np
       real(8), intent(in) :: h
       real(8), intent(inout) :: m(:,:),r(:)
       integer :: i,np2
       !Declaracion de variables
       m=0.0d0                          !Puesta de la matriz principal a 0
       np2=np**2
       do i=1,np                        !Este bucle coloca en la matriz los resultados que conocemos correspondientes al contorno
       m(i,i)=1.0d0                     !Colocaría los valores correspondientes a la condicion que nos dan de la primera fila y ultima
       m(np2-i+1,np2-i+1)=1.0d0         !Escribe por encima y por abajo los 1 en la diagonal, para los que u es trivial
       r(i)=0.0d0
       r(np2-i+1)=0.0d0                 !De igual manera este bucle nos da la condicion de que en los extremos el resultado es 0
       end do

       do i=np+1,np2,np                 !Este bucle de igual manera nos coloca los otros vectores que conocemos, correspondiente
       m(i-1,i-1)=1.0d0                 !a los extremos que no estan situados en la primera fila ni en la ultima, los intermedios
       m(i,i)=1.0d0
       r(i-1)=0.0d0
       r(i)=0.0d0
       end do

       do i=np,np2-np
       if (mod(i,np)/=0.AND.mod(i-1,np)/=0) then       !Este bucle condicionado coloca los valores no triviales que debemos resolver
              m(i,i-np)=-1.0d0                         !La condicion es que no esté en un extremo, de los que ya conocemos
              m(i,i-1)=-1.0d0
              m(i,i)=4.0d0
              m(i,i+1)=-1.0d0                          !Se colocan filas de estos valores que vienen de despejar al resolver la
              m(i,i+np)=-1.0d0                         !ecuacion empleando la derivacion numérica
              r(i)=h**2                                !El vector resultado sale h² de multiplicar todo por h²
       end if
       end do
```

Finalmente para asignar la matriz simplemente va rellenando cada fila con un bucle condicionado, teniendo cuidado de no sobreescribir los valores que hemos colocado como contorno. Es decir que cuando pase por los múltiplos de np y np-1 no haga nada.

```
np=20
np2=np**2 !np elevado al cuadrado, ya que np significa el numero de puntos en una dimension (util para la matriz)
alpha=0
beta=1
b=pi
a=0
h=(b-a)/(np-1)
allocate(m(np2,np2),u(np2),r(np2),error(np2),uapro(np,np)) !Alocatar es importante

call invoca_matvect(m,r,h,np)                          !Llama a la subrutina que saca la matriz y vector solucion
call rverdadero(uapro,h,a,np)                          !Llama a la subrutina que nos devuelve la aproximación
call grad_conj(m,r,u,tolconj,iteraconj)                !Resuelve el sistema por el gradiente conjugado
```

Como se puede observar, de forma similar al anterior en el principal lo único que hacemos es acudir a las subrutinas, que en primer lugar sacan la matriz y el vector y en segundo lugar nos saca la función que vamos a usar para validar el código.

Validación de subprogramas

En lo que concierne la validación se han llevado a cabo varias pruebas para asegurarnos de que todos los subprogramas funcionasen de la forma esperada.

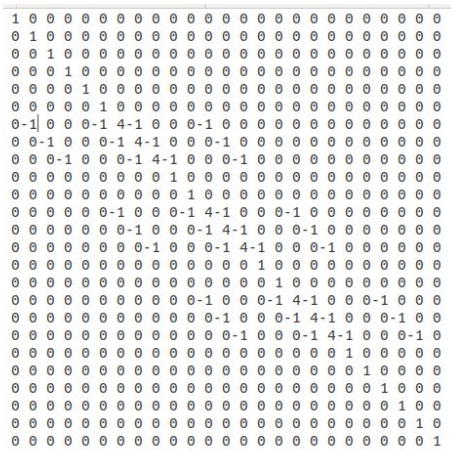
En primer lugar, hemos debido de validar la subrutina que nos hace el gradiente conjugado, para esto hemos resuelto sistemas similares con gauss pivote, subrutina que sabemos que funciona bien, ya testeada con otros sistemas y además hemos comparado al usar el gradiente conjugado con la solución exacta de los problemas de poisson y diferencial ordinaria, en ambos casos con un error dentro de los márgenes que queremos, además hemos comprobado que al aumentar la tolerancia y número de puntos la precisión aumenta (aunque también el tiempo de cálculo).

A la hora de testear las subrutinas de contorno vamos, de forma indirecta, a afianzar nuestra confianza sobre esta subrutina ya que vamos a comprobar estas otras subrutinas usando el gradiente conjugado.

La primera validación del código es que la matriz que nos genera el método de diferencias finitas es la correcta, para esto se añadió un bucle en el programa principal que nos generase un archivo con la matriz, de esta forma al inspeccionarla visualmente podemos encontrar errores.

Con este bucle simplemente podemos ver una matriz que seria de 25x25 correspondiente a np=5 en el caso de poisson, para mantener la cosa simple.

```
open(unit=11, file='matriz.txt')
do i=1,np2
write(11,"(25I2)")int(m(i,:))
end do
close(11)
```



Con esto verificamos que la matriz es correcta

Para validar la subrutina completamente en realidad tenemos que intentar que coincidan los resultados aproximados a los resultados exactos de la función en cada punto. Para esto usamos otra subrutina que nos genera la función exacta en el caso de Poisson la subrutina es esta:

```
subroutine rverdadero(uapro,h,a,np)                    !Esta subrutina es una subrutina de validacion, se trata simplemente de
integer, intent(in) :: np                              !una subrutina que nos da una solucion muy cercana a la exacta, ya que
real(8), intent(in) :: h,a                             !la exacta nos la han dado en forma de suma infinita.
real(8), intent(inout) :: uapro(:,:)
real(8),dimension(np) :: x,y
integer :: xcont, ycont, i, j
real(8),parameter :: pi=dacos(-1.0d0)
        do i=1,np
        x(i)=a+(real(i)-1.0d0)*h                       !Sacamos el vector x e y que tienen valores equiespaciados
        y(i)=a+(real(i)-1.0d0)*h
        end do

        uapro=0.0d0
        do xcont=1,np
        do ycont=1,np
                do j=1,100,2
                        do i=1,100,2                   !Realizamos la suma de la funcion 100 veces ya que el 100 es un numero
        uapro(xcont,ycont)=uapro(xcont,ycont)+&        !bastante grande y guay, lo hacemos para cada x e y (xcont,ycont)
        &(16.0d0/((real(j)**2+real(i)**2)*real(i)*real(j)*pi**2))*dsin(real(j)*x(xcont))*dsin(real(i)*y(ycont))
                        end do
                end do
        end do
        end do
```

La subrutina nos calcula los puntos, se elige 100 el sumatorio porque no tenemos suficiente tiempo como para sumar infinitas veces, y ya es una buena aproximación.

En el caso de la diferencial ordinaria la función exacta viene dada por las funciones de bessel y tenemos que calcular los coeficientes, esto se hace de forma relativamente sencilla. El programa los calcula con gauss, sin embargo para verificar se comprobó el relativamente sencillo sistema a mano.

```
 bessel en 1  0.76519768655796661        8.8256964215676956E-002
 bessel en 2  0.22389077914123567       0.51037567264974515
 coef, bessel
 -0.23803159219317838        2.0637603535605855
```

Dicho esto simplemente es mostrar las dos funciones uno al lado de la otra y ver que el error sea aceptable, para esto se nos muestra por pantalla el error en cada una de las funciones.

```
        1 Valores de u   0.0000000000000000      val aprox   0.0000000000000000      error   0.0000000000000000
        2 Valores de u   0.0000000000000000      val aprox   0.0000000000000000      error   0.0000000000000000
        3 Valores de u   0.0000000000000000      val aprox   0.0000000000000000      error   0.0000000000000000
        4 Valores de u   0.0000000000000000      val aprox   0.0000000000000000      error   0.0000000000000000
        5 Valores de u   0.0000000000000000      val aprox   0.0000000000000000      error   0.0000000000000000
        6 Valores de u   0.0000000000000000      val aprox   0.0000000000000000      error   0.0000000000000000
        7 Valores de u  0.42408456430344149      val aprox  0.44695826683919404      error   2.2873702535752549E-002
        8 Valores de u  0.53974399240275872      val aprox  0.56587310488000686      error   2.6129112477248140E-002
        9 Valores de u  0.42408456430344149      val aprox  0.44695826683919410      error   2.2873702535752605E-002
       10 Valores de u   0.0000000000000000      val aprox   1.0978859253189991E-016 error   1.0978859253189991E-016
       11 Valores de u   0.0000000000000000      val aprox   0.0000000000000000      error   0.0000000000000000
       12 Valores de u  0.53974399240275872      val aprox  0.56587310488000686      error   2.6129112477248140E-002
       13 Valores de u  0.69395656342499590      val aprox  0.72710584213093710      error   3.3149278705941199E-002
       14 Valores de u  0.53974399240275872      val aprox  0.56587310488000697      error   2.6129112477248251E-002
       15 Valores de u   0.0000000000000000      val aprox   1.3142966188525054E-016 error   1.3142966188525054E-016
       16 Valores de u   0.0000000000000000      val aprox   0.0000000000000000      error   0.0000000000000000
       17 Valores de u  0.42408456430344149      val aprox  0.44695826683919415      error   2.2873702535752660E-002
       18 Valores de u  0.53974399240275872      val aprox  0.56587310488000697      error   2.6129112477248251E-002
       19 Valores de u  0.42408456430344149      val aprox  0.44695826683919421      error   2.2873702535752716E-002
       20 Valores de u   0.0000000000000000      val aprox   1.0978859253189992E-016 error   1.0978859253189992E-016
       21 Valores de u   0.0000000000000000      val aprox   0.0000000000000000      error   0.0000000000000000
       22 Valores de u   0.0000000000000000      val aprox   1.0978859253189988E-016 error   1.0978859253189988E-016
       23 Valores de u   0.0000000000000000      val aprox   1.3142966188525054E-016 error   1.3142966188525054E-016
       24 Valores de u   0.0000000000000000      val aprox   1.0978859253189991E-016 error   1.0978859253189991E-016
       25 Valores de u   0.0000000000000000      val aprox   5.5362919696701877E-032 error   5.5362919696701877E-032
 9.1664215503178097E-003
```

Al aumentar el número de puntos nuestra precisión debería de aumentar, por esto hemos puesto el error vamos a probar si nuestra teoría coincide con lo que el programa nos dice.

Para esto se hacen sucesivos test.

Np=5 Error medio: 9.1164E-3

Np=10 Error medio:2.8469E-3

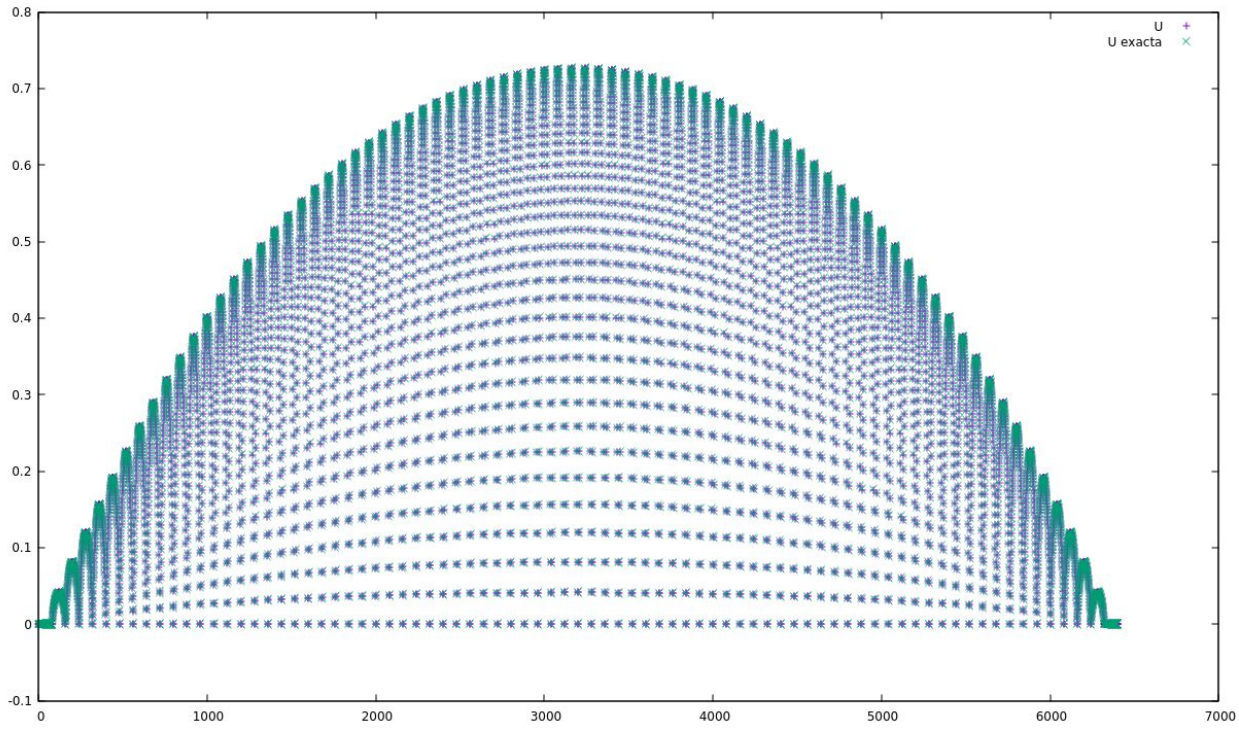
Np=20 Error medio:7.4935E-4

Np=40 Error medio:1.8991E-4

Np=80 Error medio:4.8107E-5

Vemos que al ir aumentando los puntos el error disminuye cada vez más comprobando nuestra teoría al ir haciéndose las aproximaciones de las derivadas por diferencias finitas más exactas, sin embargo para cada np la matriz que tiene que almacenarse aumenta al cuadrado, esto es una matriz para np=80 de 6400x6400 que se vuelve difícil de manejar, especialmente si tenemos en cuenta que contiene 40960000 números que por 8 bytes por número son 327.68 MegaBytes de matriz. Que ploteado en GNUplot nos da la gráfica de la portada del trabajo.

Hay que decir, que realmente debería representarse en 3D sin embargo no hemos tenido tiempo, pero cada vez que la función hace un pico sería una curva de nivel.



Uso para resolución de los problemas

En ambos problemas, se pueden modificar los parámetros (número de iteraciones del gradiente conjugado, número de puntos utilizados, tolerancia) para aproximar más a la solución real. También cabe destacar el uso de la subrutina *"gausspivote"* con la que se han resuelto sistemas generados por los programas principales.

Ecuación diferencial ordinaria.

Para la resolución de este problema, el objetivo principal es encontrar los coeficientes que cumplan las condiciones de contorno de las funciones de Bessel (función intrínseca en fortran). Para ello, hemos utilizado la subrutina *gausspivote* para resolver los coeficientes de la función aproximada y además el gradiente conjugado se ha usado para resolver el sistema de ecuaciones principal.

Parámetros variables:

tolconj - Por defecto 10E-6, tolerancia de la subrutina de resolución gradiente conjugado

iteraconj - Por defecto 100, número de iteraciones máximas del gradiente conjugado

np - Por defecto 100, número de puntos a evaluar

a, b, alpha y beta son las condiciones de contorno para este problema en particular

Para usar este programa se pueden variar los valores en el código fuente y compilar el programa usando gfortran

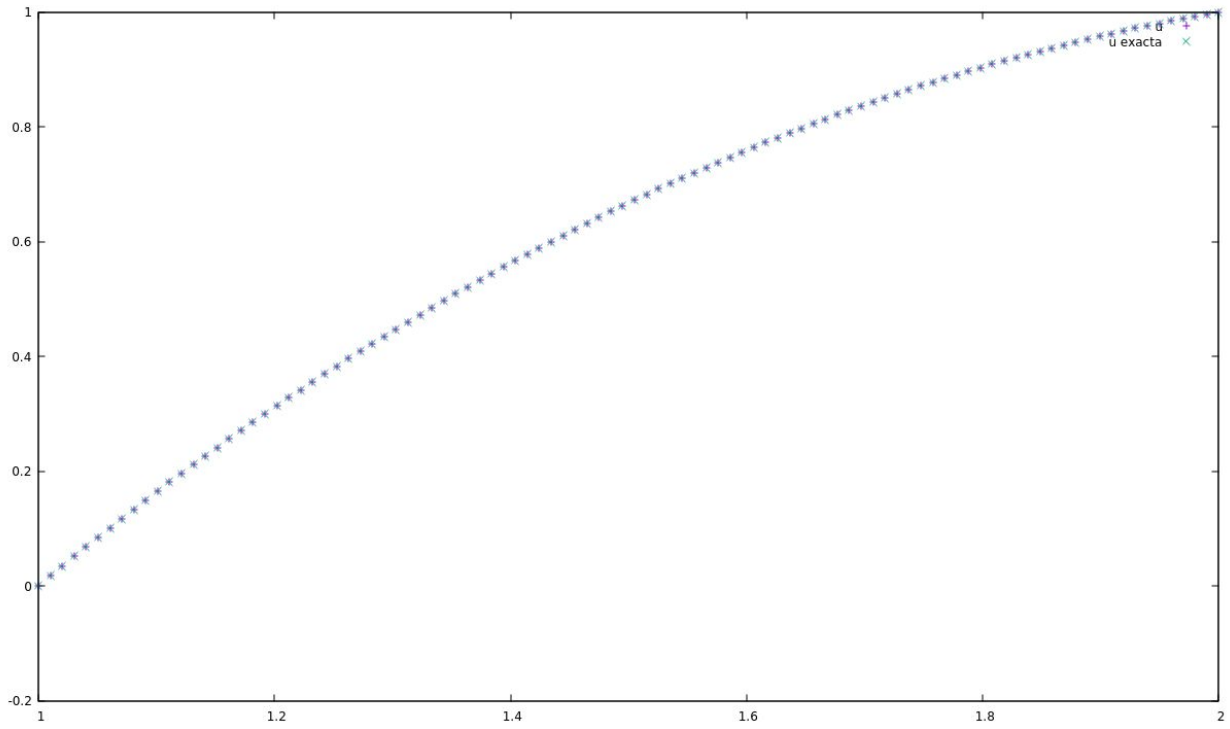
Para compilar, debemos compilar los módulos la primera vez con gfortran y luego el principal con los .o de los módulos generados

Una vez ejecutado el programa, se crea un fichero denominado "solucion_unavariable.dat", archivo que será utilizado para representar las soluciones gráficamente con el programa **Gnuplot**. Dicha gráfica es la que aparece en la página siguiente.

Uso GNUplot:

```
gnuplot> plot "solucion_unavariable.dat" using 1:2 title 'u', \
>"solucion_unavariable.dat" using 1:3 title 'u exacta'
```

[3-Usognuplot]



Ecuación de poisson.

El proceso de resolución es similar al problema anterior. En este caso corresponde a una ecuación de derivadas parciales de segundo orden.

De forma similar al anterior:

tolconj - Por defecto 10E-6, tolerancia de la subrutina de resolución gradiente conjugado

iteraconj - Por defecto 100, número de iteraciones máximas del gradiente conjugado

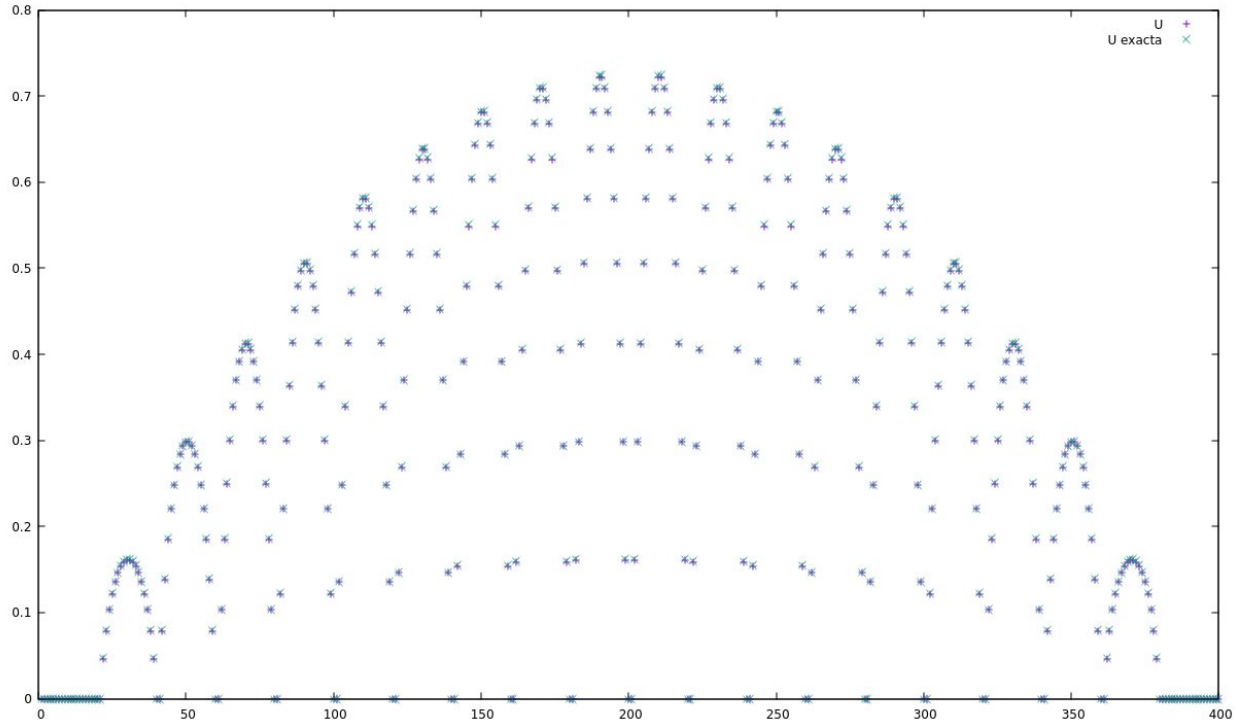
np - Por defecto 20, número de puntos a evaluar

a, b, alpha y beta son las condiciones de contorno para este problema en particular

En este caso, contamos con una solución muy aproximada, calculada en forma de serie infinita, aunque la hemos parado en 100, que utilizaremos para compararla a la solución obtenida por el programa (subrutina *rverdadero*)

Con la subrutina *invoca_matcevt* generamos la matriz y el vector resultado del problema, incluyendo las condiciones de contorno. Una vez generado el sistema, utilizamos la subrutina *grad_conj* que mediante un número de iteraciones y una tolerancia adecuada, nos da una solución muy aproximada a la solución generada por *rverdadero*. Ambas soluciones se escriben en un archivo llamado *poisson_data.dat*.

Con ayuda de **Gnuplot,** representamos ambas soluciones en el mismo gráfico, y como vemos a continuación, prácticamente están una encima de la otra, lo que nos muestra una gran precisión de nuestra solución, donde las variables independientes corresponden a x,y tal que x e y van de 0 a π, teniendo representadas las sucesivas curvas de nivel.



Bibliografía

I. J.R. Nagel; Numerical Solution to Poisson Equations Using the Finite–Difference Method. IEEE Antennas and Propagation Magazine 56(4) (2014).

II. Introduction to CFD de D. Kuzmin http://www.mathematik.uni-dortmund.de/~kuzmin/

III. Uso de gnuplot: http://people.duke.edu/~hpgavin/gnuplot.html