Norges Naturvitenskapelige Universitet

TMR4160 - Datametoder for ingeniørtekniske anvendelser

Prosjektrapport: Løsning og visualisering av den todimensjonale Poisson-likningen

Av:
Kandidatnummer 741680

1 Introduksjon

I denne oppgaven blir Poissons ligning i 2D løst vha. numeriske metoder implementert i Fortran. Videre visualiseres løsningen i et C-program ved bruk av OpenGL, i form av et tredimensjonalt plott.

2 Løsning av Poissons ligning

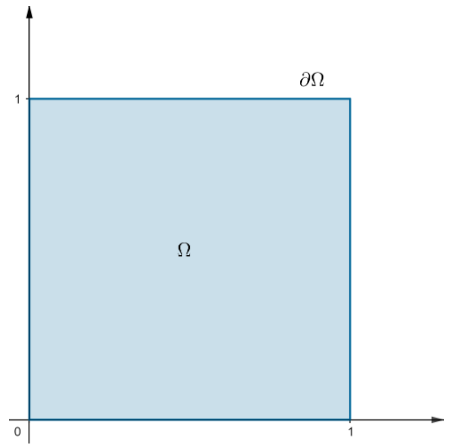
Poissons ligning er en av de viktigste elliptiske partielle differensialligningene, med applikasjoner innenfor felt som elektrostatikk, varmeledning og fluidmekanikk. Hvis vi begrenser oss til to romlige dimensjoner tar ligningen formen

$$\nabla^2\phi(x,y) = \phi_{xx} + \phi_{yy} = g(x,y). \tag{1}$$

Problemet beror på å bestemme den ukjente funksjonen ϕ, når g er kjent innenfor et gitt domene. Vi begrenser oss i det følgende til det kvadratiske domenet Ω, gitt ved

$$\Omega := (0,1) \times (0,1). \tag{2}$$

Randen til Ω betegnes $\partial\Omega$ (se figur 1). I vårt problem skal løsningen kunne ta hensyn til to randbetingelser: Dirichlets og Neumanns.



Figur 1: Domenet Ω med rand $\partial\Omega$

2.1 Dirichlets randbetingelse

Når Dirichlets randbetingelse er aktiv, er verdien av ϕ kjent på randen. I vårt tilfelle har vi

$$\phi(x,y) = \frac{1}{4}(x^2+y^2), \qquad \forall(x,y) \in \partial\Omega. \tag{3}$$

Videre er funksjonen g gitt som:

$$g(x,y) = 1 \tag{4}$$

Med denne informasjonen kan man anvende standard løsningsteknikker for partielle differensialligninger, for å finne den analytiske løsningen

$$\phi(x,y) = \frac{1}{4}(x^2+y^2) \tag{5}$$

2.2 Neumanns randbetingelse

Neumanns randbetingelse innebærer at vi kjenner den normalderiverte av ϕ på randen. Betingelsen er her

$$\frac{\partial\phi}{\partial n} = 0, \qquad \forall(x,y) \in \partial\Omega \tag{6}$$

hvor den positive normalen peker ut fra Ω. På de vertikale grensene svarer den normalderiverte til den deriverte mhp. x. Likeledes er den normalderiverte på de horisontale grensene lik den y-deriverte. Merk at den normalderiverte ikke er veldefinert i hjørnepunktene. Når denne betingelsen er aktiv er det i vårt tilfelle videre gitt at løsningen tilfredsstiller

$$\phi(0,0) = 0. \tag{7}$$

I tillegg er g kjent, men her er det to muligheter som skal tas høyde for i programmet:

$$g(x,y) = 12 - 12x - 12y \tag{8}$$
$$g(x,y) = (6-12x)(3y^2-2y^3) + (3x^2-2x^3)(6x-12y) \tag{9}$$

Dersom vår g er (8), finner man at den analytiske løsningen er

$$\phi(x,y) = 3(x^2+y^2) - 2(x^3+y^3) \tag{10}$$

mens (9) vil gi løsningen

$$\phi(x,y) = (3x^2-2x^3)(3y^2-2y^3). \tag{11}$$

2.3 Diskretisering av Poissonligningen

For å løse (1) ved datametoder er grunnidéen å omforme differensialligningen til en diskret differensligning definert på en $m \times n$ grid [1]. Her er griden kvadratisk, med lik steglengde i x- og y-retning. Taylorekspansjonen av (1) om x er

$$\phi(x+h,y) \approx \phi(x,y) + h\phi_x(x,y) + \frac{1}{2}h^2\phi_{xx}(x,y) + \frac{1}{6}h^3\phi_{xxx}(x,y) + \mathcal{O}(h^4) \tag{12a}$$
$$\phi(x-h,y) \approx \phi(x,y) - h\phi_x(x,y) + \frac{1}{2}h^2\phi_{xx}(x,y) - \frac{1}{6}h^3\phi_{xxx}(x,y) + \mathcal{O}(h^4) \tag{12b}$$

Ved å ignorere ledd av høyere orden enn 2, og subtrahere (12b) fra (12a) fås

$$\phi_x(x,y) \approx \frac{1}{2h}[\phi(x+h,y) - u(x-h,y)]. \tag{13}$$

På samme vis fås

$$\phi_y(x,y) \approx \frac{1}{2h}[\phi(x,y+h) - u(x,y-h)]. \tag{14}$$

Et uttrykk for ϕ_{xx} oppnås ved å addere (12a) og (12b), og ignorere ledd i $h^n,\ n \geq 4$:

$$\phi_{xx}(x,y) \approx \frac{1}{2h}[\phi(x+h,y) - 2\phi(x,y) + \phi(x-h,y)] \tag{15}$$

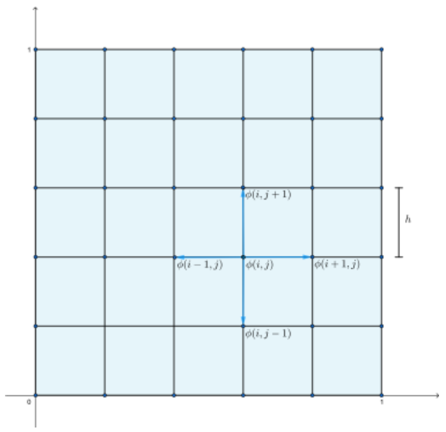
Denne framgangsmåten vil også gi

$$\phi_{yy}(x,y) = \frac{1}{2h}[\phi(x,y+h) - 2\phi(x,y) + \phi(x,y-h)] \tag{16}$$

Innsetting av (15) og (16) i Poissonligningen (1) resulterer så i

$$\phi(x+h,y) + \phi(x,y+h) + \phi(x-h,y) + \phi(x,y-h) - 4\phi(x,y) = h^2 g(x,y) \tag{17a}$$
$$\implies \quad \phi(x,y) = \frac{1}{4}\left[\phi(x+h,y) + \phi(x,y+h) + \phi(x-h,y) + \phi(x,y-h) - h^2 g(x,y)\right] \tag{17b}$$

Dette er altså den diskretiserte Poissonligningen. I det følgende bruker vi subskriptet (i,j) for å indikere funksjonsverdi i node (i,j). Ser vi på figur 2, kan vi se at denne sier oss at verdien av ϕ i node (i,j) avhenger av funksjonsverdien i nabonodene, samt den kjente verdien $g(i,j)$. Vi nummererer nodene med $(i,j) = (1,1)$ i origo, og for hvert steg i positiv retning øker den korresponderende indeksen med 1. Da har vi at (x,y)-verdien i node (i,j) blir $h \cdot (i-1, j-1)$, hvilket medfører



Figur 2: Grid med steglengde $h = 0.2$ og $n \times n$ noder, $n = \frac{1+h}{h} = 6$

$\phi(x,y) = \phi\left(h(i-1), h(j-1)\right)$ i node (i,j). Merk at (17) bare er definert når node (i,j) er intern - er vi på randen utelater vi leddene som havner "utenfor" gridet.

Alternativt kan man skrive om ligningene for hver node til en matriseligning

$$A\phi = b. \tag{18}$$

Til dette trenger vi en måte å ordne nodene på. Vi velger å la rekkefølgen på nodene defineres av "column-major" ordning:

$$\phi = \begin{bmatrix}\phi_{1,1} & \phi_{2,1} & \dots & \phi_{n,1} & \phi_{1,2} & \dots & \phi_{n,2} & \dots & \phi_{n,n}\end{bmatrix}^\top \in \mathbb{R}^{n\times n} \tag{19}$$

Med dette kan vi skrive (17) for alle nodene som

$$A\phi = \begin{bmatrix} D & -I & & & & & \\ -I & D & -I & & & & \\ & -I & D & -I & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & -I & D & -I & \\ & & & & -I & D & -I \\ & & & & & -I & D \end{bmatrix}\phi = -h^2 \begin{bmatrix} g_{1,1} \\ g_{2,1} \\ \vdots \\ \vdots \\ \vdots \\ g_{n,n}\end{bmatrix} = b, \tag{20}$$

hvor

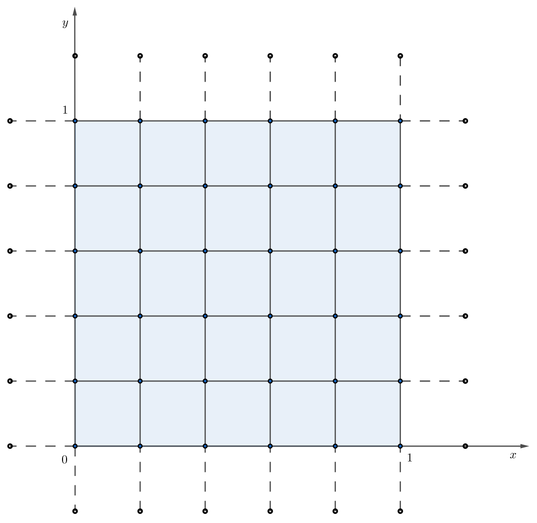
$$D = \begin{bmatrix} 4 & -1 & & & & & \\ -1 & 4 & -1 & & & & \\ & -1 & 4 & -1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & -1 & 4 & -1 & \\ & & & & -1 & 4 & -1 \\ & & & & & -1 & 4 \end{bmatrix} \in \mathbb{R}^{n\times n} \tag{21}$$

$I \in \mathbb{R}^{n\times n}$ er identitetsmatrisa og $A \in \mathbb{R}^{n^2\times n^2}$. Implementering av Dirichlets randbetingelse er uproblematisk i henhold til det ovenstående. I så tilfelle settes bare verdien i nodene på randen utifra

(3). Da har man et sett av $(n-2)\times(n-2)$ ligninger i like mange ukjente, og (17) kan brukes til å kalkulere verdien i de interne nodene direkte. Bruker man matriseformuleringen tas de kjente nodeverdiene med i vektoren b.

Skal vi bruke Neumanns randbetingelse blir det litt mer jobb for å håndtere den gitte normalderiverte på randen. Vi innfører fiktive noder (se Figur 3) utenfor domenet Ω, og approksimerer de ulike normalderiverte på $\partial\Omega$ ved sentral-differanse-formelen:

$$\frac{\partial}{\partial n}\phi_{i,j} \approx \frac{\phi_{i-1,j}-\phi_{i+1,j}}{2h} \implies \phi_{i-1,j}=\phi_{i+1,j}, \quad i=1 \tag{22a}$$
$$\frac{\partial}{\partial n}\phi_{i,j} \approx \frac{\phi_{i+1,j}-\phi_{i-1,j}}{2h} \implies \phi_{i+1,j}=\phi_{i-1,j}, \quad i=n \tag{22b}$$
$$\frac{\partial}{\partial n}\phi_{i,j} \approx \frac{\phi_{i,j-1}-\phi_{i,j+1}}{2h} \implies \phi_{i,j-1}=\phi_{i,j+1}, \quad j=1 \tag{22c}$$
$$\frac{\partial}{\partial n}\phi_{i,j} \approx \frac{\phi_{i,j+1}-\phi_{i,j-1}}{2h} \implies \phi_{i,j+1}=\phi_{i,j-1}, \quad j=n \tag{22d}$$



Figur 3: Gridet med fiktive noder utenfor domenet Ω.

Vi antar at (17) er gyldig på randen, inklusivt hjørnepunktene. Da ser vi at vi får tre mulige uttrykk for $\phi_{i,j}$ på den venstre vertikale grensen:

$$\phi_{i,j} \approx \frac{1}{4}\left[2\phi_{i,j+1}+\phi_{i+1,j}+\phi_{i-1,j}-h^2 g_{i,j}\right], \quad n<i<1 \tag{23}$$
$$\phi_{i,j} \approx \frac{1}{4}\left[2\phi_{i,j+1}+2\phi_{i-1,j}-h^2 g_{i,j}\right], \quad i=n \tag{24}$$
$$\phi_{i,j} \approx \frac{1}{4}\left[2\phi_{i+1,j}+2\phi_{i,j+1}-h^2 g_{i,j}\right], \quad i=1 \tag{25}$$

På samme måte finner man uttrykk for $\phi_{i,j}$ på de andre grensene. Dette medfører at A-matrisa i

(20) vil se litt annerledes ut med aktiv Neumannbetingelse:

$$A = \begin{bmatrix} D & -2I & & & & & \\ -I & D & -I & & & & \\ & -I & D & -I & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & -I & D & -I & \\ & & & & -I & D & -I \\ & & & & & -2I & D \end{bmatrix} \tag{26}$$

$$D = \begin{bmatrix} 4 & -2 & & & & & \\ -1 & 4 & -1 & & & & \\ & -1 & 4 & -1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & -1 & 4 & -1 & \\ & & & & -1 & 4 & -1 \\ & & & & & -2 & 4 \end{bmatrix} \in \mathbb{R}^{n\times n} \tag{27}$$

Men nå blir A singulær! Man kan f.eks se dette ved å merke seg at summen av elementene i hver rad blir 0. Dette medfører at $A\phi = 0$ for $\phi = 1 \neq 0$, altså er A singulær. Heldigvis har vi betingelsen (7), slik at vi kan nulle ut alle elementer til høyre for diagonalen i første rad av A (samt sette $b(1) = 0$). Da oppnår vi et system med en unik løsning.

2.4 Numerisk løsning

For å løse Poissonligningen numerisk, er to iterative algoritmer implementert i Fortran. Algoritmene er Jacobimetoden og Gauss-Seidel-metoden.

2.4.1 Jacobimetoden

Jacobimetoden kan sies å være den enkleste iterative algoritmen. Gitt et kvadratisk lineært system $Ax = b$ av størrelse n, ser vi ved å betrakte ligning i i systemet at

$$\sum_{j=1}^{n} a_{ij}x_j = b_j. \tag{28}$$

Løst for x_i får man den elementvise formuleringen av Jacobimetoden:

$$x_i^{(m+1)} = \frac{1}{a_{ii}}(b_j - \sum_{j\neq i} a_{ij}x^{(m)}) \tag{29}$$

Det er da lett å se at vi kan bruke de eksplisitte formlene for $\phi_{i,j}$ utledet ovenfor i Jacobiiterasjonen. Vi ser også at for hver iterasjon avhenger det nye elementet kun av de foregående. I Fortranprogrammet er løsning av Poissonligningen med Jacobimetoden implementert vha. de eksplisitte formlene. Man fortsetter å iterere inntil "root-mean-square" av differansen $x^{(m+1)} - x^{(m)}$ er mindre enn en gitt toleranse.

En tilstrekkelig betingelse for konvergens av Jacobimetoden er at matrisen A er strengt diagonaldominant, eller strengt diagonaldominant i minst en rad og diagonaldominant i de andre. Matrisen vi fant fram til i avsnitt 2.4.1 oppfyller den første av disse betingelsene, mens matrisen i 2.4.2 oppfyller den andre.

2.4.2 Gauss-Seidel-metoden

Gauss-Seidel-metoden ligner svært mye på Jacobimetoden, men skiller seg fra den ved at den benytter seg av nylig kalkulerte verdier i iterasjonen:

$$x_i^{(m+1)} = \frac{1}{a_{ii}}(b_j - \sum_{j<i} a_{ij} x_j^{(m+1)} - \sum_{j>i} a_{ij} x_j^{(m)}) \tag{30}$$

Gauss-Seidel-algoritmen er implementert i Fortranprogrammet ved å bruke både en generell matriseformulering, samt en rutine som anvender formlene funnet over. Også her fortsetter man å iterere inntil RMS av differansen $x^{(m+1)} - x^{(m)}$ er mindre enn en gitt toleranse.

Tilstrekkelige betingelser for konvergens av Gauss-Seidel-metoden er som for Jacobimetoden, så vi vet at den vil konvergere.

3 Visualisering av løsning

For å visualisere løsningen, er et program skrevet i C. Programmet benytter seg av API-et OpenGL samt FreeGLUT (et open-source alternativ til OpenGL Utility Toolkit). Måten man har valgt å visualisere løsningen på, er ved et tredimensjonalt plott. Løsningen vil beskrive en overflate i rommet, og denne er tegnet opp ved å bruke polygoner. Videre er gridet tegnet på overflaten. I tillegg er det implementert funksjonalitet for å rotere og skalere plottet.

4 Resultater

Vi tester løserne med Dirichlets randbetingelse og tilhørende g, og Neumanns randbetingelse med g som gitt i (8) og (9). Toleransen er satt til 10^{-6}. Resultater fra kjøringene er vist nedenfor, samt visualiseringer.

Tabell 1: Løsning av problemet med Dirichlets randbetingelse.

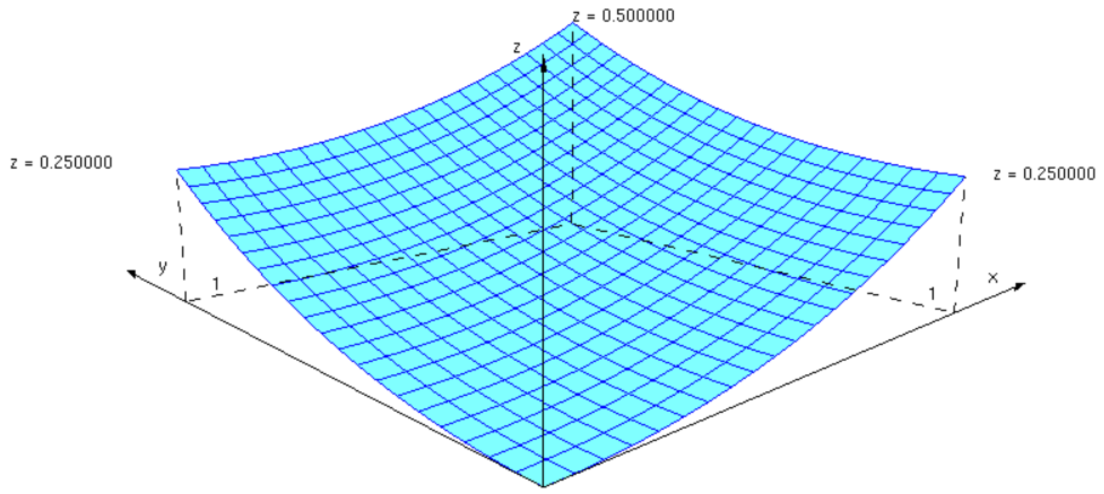
		Jacobi		Generell GS		Spesiell GS	
Gridstørrelse	Steglengde h	Iterasjoner	Tid [s]	Iterasjoner	Tid [s]	Iterasjoner	Tid [s]
10	0.1	175	0	99	0.000	98	0.000
20	0.05	589	0.016	332	0.422	329	0.000
40	0.025	1912	0.141	1087	57.031	1080	0.094
80	0.0125	5863	1.734	-	-	3402	1.031

Tabell 2: Løsning av problemet med Neumanns randbetingelse og g i (8)

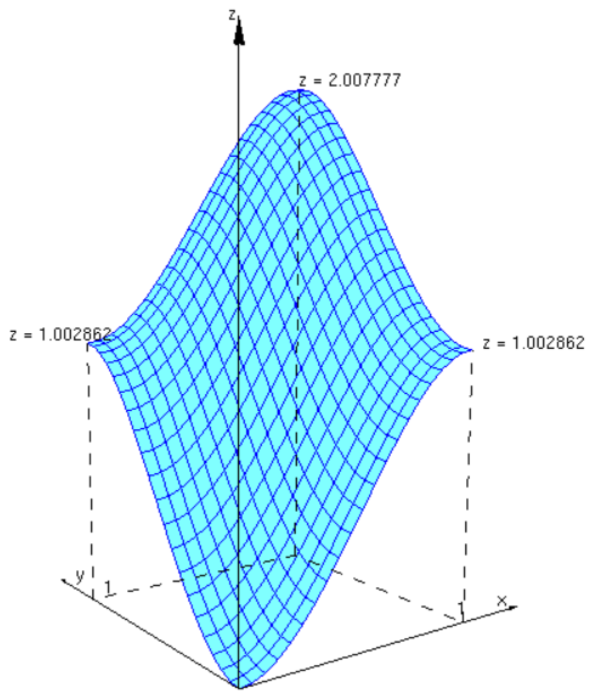
		Jacobi		Generell GS		Spesiell GS	
Gridstørrelse	Steglengde h	Iterasjoner	Tid [s]	Iterasjoner	Tid [s]	Iterasjoner	Tid [s]
10	0.1	6077	0.047	3353	0.391	3353	0.016
20	0.05	22735	0.453	12839	25.938	12839	0.267
40	0.025	76596	6.000	-	-	45075	3.625
80	0.0125	216480	69.094	-	-	139055	43.188

Tabell 3: Løsning av problemet med Neumanns randbetingelse og g i (9)

		Jacobi		Generell GS		Spesiell GS	
Gridstørrelse	Steglengde h	Iterasjoner	Tid [s]	Iterasjoner	Tid [s]	Iterasjoner	Tid [s]
10	0.1	4913	0.031	2793	0.329	2793	0.016
20	0.05	17118	0.359	10083	20.938	10083	0.203
40	0.025	50250	4.000	-	-	32027	2.547
80	0.0125	95541	30.531	-	-	78871	25.984

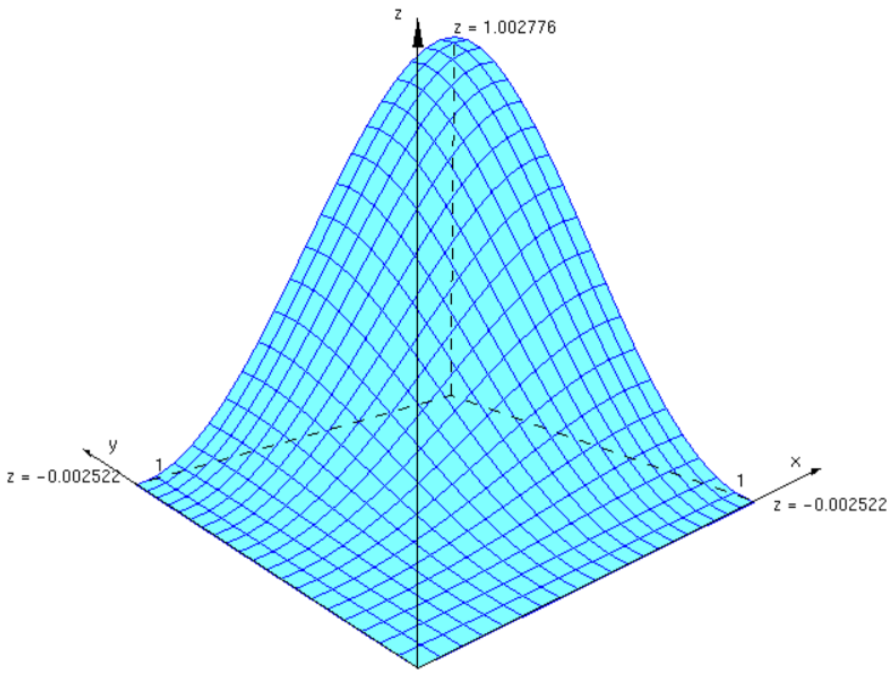


Figur 4: Visualisering av løsning med 20×20 grid, Dirichlets randbetingelse. Gauss-Seidel er brukt som løsningsalgoritme.



Figur 5: Visualisering av løsning med 20$\times$20 grid, Neumanns randbetingelse og g i (8). Gauss-Seidel er brukt som løsningsalgoritme.

Det første vi merker oss er at kjøringer med generell Gauss-Seidel for gridstørrelse>20 ikke er inkludert pga. lang kjøretid, sammenlignet med de andre løserne. Vi ser at antall iterasjoner er tilnærmet lik for generell og spesiell variant av Gauss-Seidel-metoden, så tidsdifferansen skyldes måten førstnevnte er implementert. For det første inngår overhead fra opprettelse av A og b i tidtakingen. Denne vil være $\mathcal{O}(n^2)$ pga. de doble løkkene som går over antall ukjente i hver rad/kolonne. For det andre er hver iterasjon mye mer tidkrevende. Den ytre løkken i hver iterasjon går fra 1 til n^2, i tillegg til to indre løkker som til sammen også går fra 1 til n^2; noe som gir $\mathcal{O}(n^4)$. Det er dette som dominerer kjøretiden. I den andre rutinen utnytter vi at vi kjenner formlene, slik at vi ikke trenger å gå gjennom en stor matrise for å hente ut riktige koeffisienter. Da trenger vi for hver iterasjon å gå gjennom hver rad og hver kolonne i gridet, hvilket gir $\mathcal{O}(n^2)$. Kjøretiden for Jacobis metode blir av samme grunn også $\mathcal{O}(n^2)$. Likevel ser vi at spesiell Gauss-Seidel gjennomgående trenger færre iterasjoner og mindre tid sammenlignet med Jacobi. Fra tabell 1 finner vi at Gauss-Seidel trenger i gjennomsnitt 56.6% færre iterasjoner enn Jacobi. Tilsvarende fra de andre tabellene får vi 58.7% og 65.5%. Bare i tabell 3, og kjøringen med gridstørrelse 80 er ikke Gauss-Seidel *mye* bedre enn Jacobi. Det er nokså uventet, med tanke på at koden er nærmest identisk. En annen fordel med Gauss-Seidel er at man strengt tatt ikke trenger å lagre foregående iterasjon, fordi man bruker de verdiene man har kalkulert i hver iterasjon. Da kan man skrive over elementene i arrayet som holder løsningen mens man itererer (dette er dog ikke implementert i koden).



Figur 6: Visualisering av løsning med 20×20 grid, Neumanns randbetingelse og g i (9). Gauss-Seidel er brukt som løsningsalgoritme.

Når det gjelder nøyaktighet, er et lite skript tillaget i MATLAB som finner det maksimale avviket (i absoluttverdi) mellom analytisk og numerisk løsning når gridstørrelsen er 20, og Jacobi eller spesiell Gauss-Seidel er anvendt. Problem 1, 2 og 3 svarer til rekkefølgen av de ulike problemene i denne rapporten. Avvikene er presentert i tabell 4.

Tabell 4: Største avvik med spesiell Gauss-Seidel

	Jacobi	Generell GS
Problem	Avvik	Avvik
1	0.000170	0.000084
2	0.005531	0.007777
3	0.004660	0.002778

Som vi ser er Gauss-Seidel litt bedre enn Jacobi, med unntak av i problem 2. Hvis vi husker tilbake til diskretiseringen av Poissonlikningen i avsnitt 2.3, så ignorerte vi ledd av høyere orden enn 2. Da vil feilen bli $\mathcal{O}(h^2)$. Med en steglengde $h = 0.05 \implies h^2 = 0.025$ ser vi at dette stemmer med avvikene over.

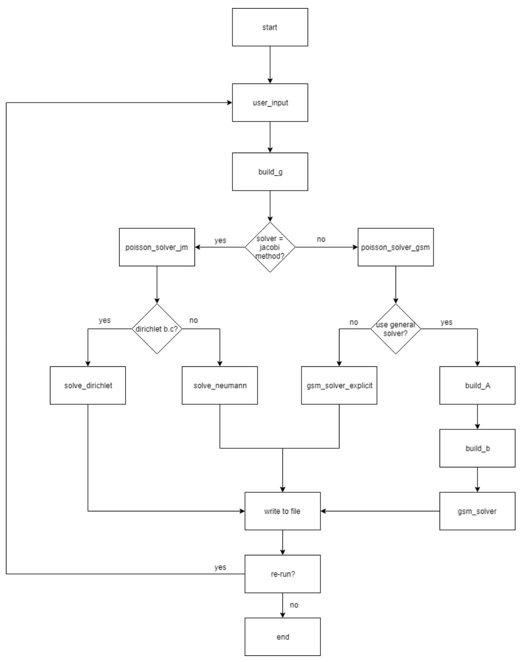
5 Konklusjon

Alle implementerte algoritmer fungerer slik de skal, og gir korrekt løsning innenfor forventet feilmargin. Gauss-Seidel-algoritmen med den mer generelle formuleringen er dog svært treg, og ble ikke brukt for gridstørrelser over 20 pga. dette. Varianten av Gauss-Seidel som utnytter kjennskap til ligningssystemet er nær identisk med Jacobimetoden, men var jevnt over mye raskere og krevde færre iterasjoner for å finne løsningen. Denne forutsetter dog som sagt en kjent struktur av matrisen, og for andre systemer uten den samme regelmessigheten vi finner her kan den ikke anvendes.

References

[1] Erwin Kreyszig, Herbert Kreyszig, and Edward J. Norminton. *Advanced Engineering Mathematics, 10th Edition.* John Wiley Sons, 2010.

A Fortran-kode



Figur 7: Flowchart over Fortranprogrammet. I tillegg kalles r8mat_rms i blokkene rett før "write to file".

```fortran
program main
  ! Program for solving the 2-D Poisson equation
  !      d^2              d^2
  !    ----- u(x,y) + ----- u(x,y) = g(x,y)
  !   dx^2        dx^2
  ! on the square unit grid, by discretizing the equation and using the iterative Jacobi- or Gauss-Seidel
  ! methods. User may choose grid step size (identical in x- and y-direction) and
  ! boundary conditions through console I/O. Also allows a preset two-choice of g(x,y)
  ! for the Neumann b.c. Results are written to file.
  !
  ! For the Dirichlet b.c, g(x,y) = 1, while for Neumann b.c either
  ! g(x,y) = 12 - 12x - 12y', or
  ! g(x,y) = (6 - 12x) * (3y^2 - 2y^3) + (3x^2 - 2x^3)*(6-12y)'
  !
  ! In the case of Jacobi iteration, the equation is solved by explicitly applying the five-point
  ! formula.
  !
  ! If using Gauss-Seidel iteration, the equations is solved similarly to the Jacobi procedure, OR
  ! a system of equations Au=b is formulated and solved.
  !
  !
  ! Date/version: 30.04.2019/1.0

  implicit none
  double precision, parameter :: tolerance = 0.000001     ! error tolerance
  integer :: i,j                        ! loop variables
  integer :: iter                       ! number of iterations
  character(len=3) :: solver                  ! specifies solver algorithm
  integer numx, numy                      ! number of nodes is numx*numy
  double precision, dimension(:,:), allocatable :: u      ! matrix containing
    solution u
  double precision, dimension(:,:), allocatable :: g      ! matrix containing rhs of
    Poisson eq.
  logical :: dirichlet, neumann_g1, neumann_g2        ! boundary conditions
  logical :: explicit                     ! specifies variant of GS method
  real :: h                           ! grid step size
  character :: keypress                   ! user input
  double precision time_start, time_end           ! helper variables for measuring
    time

  write(*,*)'This program solves the 2D Poisson equation'
  write(*,*)
  write(*,*) '    d^2            d^2'
  write(*,*) '   ----- u(x,y) + ----- u(x,y) = g(x,y)'
  write(*,*) '    dx^2           dx^2'
  write(*,*)
  write(*,*)'for the unknown function u on the square unit grid.'
  write(*,*)

  ! prompt user for choosing tolerance, grid step size, boundary conditions and g(x,y
    ) (if choosing Neumann bc)
100 call user_input(h, dirichlet,neumann_g1,neumann_g2,solver,explicit)
  numx = 1.0/h + 1
  numy=numx

  allocate(u(numx,numy))

  allocate(g(numx,numy))
  call build_g(g,dirichlet,neumann_g1,neumann_g2, numx, numy, h)

  write(*,*)'Working...'

  call cpu_time(time_start)
  if (solver=='jm') then
    call poisson_solver_jm(u,g,numx,numy,tolerance,h,iter,dirichlet)
  else
    call poisson_solver_gsm(u,g,numx,numy,tolerance,iter,dirichlet,h,explicit)
  end if
  call cpu_time(time_end)

  write(*,*)'System was solved in:',(time_end - time_start),'seconds.'

!-------------print u to console if it is not too large
  if (numx<=11) then
    write(*,*)'Computed solution u is'
    write(*,*)
    do i=1,numx
        write(*,"(100g15.5)") ( u(i,j), j=1,numx )
    end do
    write(*,*)
  end if
```

```
!----------------------


!---------------print number of iterations
  write(*,*) 'Number of iterations:',(iter)
!----------------------

! create and open new .dat-file for storing solution
! the solution is stored in format
!--------------
! numx numy
! u(1,1)
! u(2,1)
! ...
! u(numx,1)
! u(1,2)
! u(2,2)
! ...
! ...
! u(numx, numy)
!----------------
! Write solution to "solution.dat". If file does not exist it is created.
! If it does exist, it is overwritten.
  open(100, file='solution.dat', status='replace')
  write(100,*)numx, numy
  do j=1,numy
    do i=1,numx
      write(100,*)u(i,j)
    end do
  end do
  close(100)
  write(*,*)'Results are written to file "solution.dat".'

  ! deallocate allocated variables
  deallocate(g)
  deallocate(u)

  write(*,*)
  write(*,'(A)', ADVANCE='NO')' Re-run program? (NOTE: this overwrites the "solution.dat"-file) y/n: '
  read(*,*) keypress
  if (keypress=='y') then
    write(*,*)
    go to 100
  end if

  write(*,*) 'Program terminating.'
  write(*,*)

end program

function r8mat_rms ( m, n, a )

!*****************************************************************************80
!
!! R8MAT_RMS returns the root mean square of data stored as an R8MAT.
!
!  Licensing:
!
!    This code is distributed under the GNU LGPL license.
```

```fortran
!
!  Modified:
!
!    21 August 2010
!
!  Author:
!
!    John Burkardt
!
!  Parameters:
!
!    Input, integer ( kind = 4 ) M, N, the number of rows and columns in A.
!
!    Input, real ( kind = 8 ) A(M,N), the data whose RMS is desired.
!
!    Output, real ( kind = 8 ) R8MAT_RMS, the root mean square of A.
!
  implicit none

  integer m
  integer n

  double precision a(m,n)
  double precision r8mat_rms

  r8mat_rms = sqrt ( sum ( a(1:m,1:n)**2 ) / real ( m * n) )

  return

end

subroutine user_input(h, dirichlet, neumann_g1, neumann_g2,solver,explicit)
! This routine prompts the user for choosing step size, boundary conditions and
   solver algorithm,
! and outputs variables corresponding to these
!
!
! Date/version: 30.04.2019/1.0

  implicit none
  real, intent(OUT) :: h                      ! step size
  logical, intent(OUT) :: dirichlet, neumann_g1, neumann_g2 ! boundary conditions
  logical, intent(OUT) :: explicit              ! specifies GS method variant
  character(len=3), intent(OUT) :: solver           ! solver algorithm
  character :: keypress1, keypress2, keypress3, keypress4   ! user input
                        !

101 write(*,'(A)',ADVANCE='NO')' Set the step size h (h must be in (0,1) and give
   equally spaced grid nodes): '
  read(*,'(f10.9)') h
  if (h>=1.0 .or. h<=0.0 .or. mod(1.0/h,1.0)/=0) then
    write(*,*)'Invalid input given...'
    go to 101
  else
    write(*,*)'Step size set to:',(h)
    write(*,*)
  end if

  write(*,*)'Boundary condition choices are either'
  write(*,*) ' 1) Dirichlet b.c (u is known on the boundary)'
  write(*,*) ' 2) Neumann b.c (du/dn is known on the boundary)'
   do while (.not. (keypress1 == '1' .or. keypress1 == '2'))
    write(*,'(A)',ADVANCE='NO')' Which boundary condition should be applied? 1/2: '
    read(*,*) keypress1
  end do


  if (keypress1 == '1') then
    dirichlet = .true.
    write(*,*)'Dirichlet boundary condition is applied.'
  else
    dirichlet = .false.
    write(*,*)'Neumann boundary condition is applied.'
    write(*,*)
    write(*,*)'Possible choices for function g(x,y) are'
    write(*,*) '1) g(x,y) = 12 - 12x - 12y'
    write(*,*) '2) g(x,y) = (6 - 12x) * (3y^2 - 2y^3) + (3x^2 - 2x^3)*(6-12y)'
    do while (.not. (keypress2 == '1' .or. keypress2 == '2'))
        write(*,'(A)',ADVANCE='NO')' Which function g(x,y) should be used? 1/2: '
        read(*,*) keypress2
    end do

    if (keypress2=='1') then
      neumann_g1 = .true.
      neumann_g2 = .false.
      write(*,*)'g(x,y) set to: g(x,y) = 12 - 12x - 12y.'
    else
      neumann_g2 = .true.
      neumann_g1 = .false.
      write(*,*)'g(x,y) set to: g(x,y) = (6 - 12x) * (3y^2 - 2y^3) + (3x^2 - 2x^3)
    *(6-12y).'
    end if

  end if

  write(*,*)

  write(*,*)'Choose solver algorithm:'
  write(*,*) '1) Jacobi method'
  write(*,*) '2) Gauss-Seidel method'
  do while (.not. (keypress3 == '1' .or. keypress3 == '2'))
    write(*,'(A)',ADVANCE='NO')' Which solver should be used? 1/2: '
    read(*,*) keypress3
  end do

  if (keypress3=='1') then
    solver = 'jm'
    write(*,*)'Jacobi method chosen as solver algorithm.'
    write(*,*)
  else
    solver = 'gsm'
    write(*,*)'Gauss-Seidel method chosen as solver algorithm.'
    write(*,*)
    write(*,*)'Use'
    write(*,*)' 1) general formulation of GS algorithm, or'
    write(*,*)' 2) special formulation.'
    do while (.not. (keypress4 == '1' .or. keypress4 == '2'))
    write(*,'(A)',ADVANCE='NO')' Which algorithm variant should be used? 1/2: '
    read(*,*) keypress4
    end do
    if (keypress4 == '2') then
      explicit = .true.
    else
      explicit = .false.
    end if
  end if

end subroutine

subroutine build_g(g, dirichlet, neumann_g1, neumann_g2, numx, numy, h)
! This routine creates a 2D matrix g that holds the values of the right hand side of
    the
! Poisson equation such that g(x,y)=g(i*h,j*h); where x,y are coordinates of the
    nodes,
! i,j are the indices of g and h is the grid step size.
!
! In addition, if the Dirichlet boundary condition is applied, the routine
! sets the points on the matrix's boundary to the known value of u there.
!
!
! Date/version: 30.04.2019/1.0

  implicit none
  logical, intent(IN) :: dirichlet, neumann_g1, neumann_g2  ! boundary conditions
  integer, intent(IN) :: numx, numy              ! numx*numy is number of nodes
  real, intent(IN) :: h                      ! step size
  double precision, dimension(numx,numy), intent(OUT) :: g  ! matrix g
  double precision :: x, y                   ! x- and y- coords
  integer i,j                          ! iteration variables

  if (dirichlet .eqv. .true.) then
    g = 1
    ! set the known boundary point values:
    do i=1,numx
      g(i,1)=0.25*((h*(i-1))**2)
      g(i,numy)=0.25*((h*(i-1))**2+(h*(numy-1))**2)
      g(1,i)=g(i,1)
      g(numx,i)=g(i,numy)
    end do
  else
    do j=1, numy
      x=(j-1)*h
      do i=1,numx
        y=(i-1)*h
        if (neumann_g1 .eqv. .true.) then
          g(i,j) = 12-12*x-12*y
        else if (neumann_g2 .eqv. .true.) then
          g(i,j) = (6-12*x)*(3*y**2-2*y**3) + (3*x**2-2*x**3)*(6-12*y)
        end if
      end do
    end do
  end if

end subroutine

subroutine poisson_solver_jm(u,g,numx,numy,tolerance,h,iter,dirichlet)
! This routine solves the 2D Poisson equation by making the call
! to the appropriate subroutine, according to which boundary condition
! is active.
!
```

```fortran
!
! Date/version: 01.05.2019/1.0

  implicit none
  integer, intent(IN) :: numx,numy              ! number of nodes is numx*numy
  real, intent(IN) :: h                     ! grid step size
  double precision, dimension(numx,numy), intent(IN) :: g   ! matrix containing rhs
    of Poisson eq.
  double precision, intent(IN) :: tolerance         ! error tolerance
  double precision, dimension(numx,numy), intent(OUT) :: u  ! matrix containing
    solution u
  integer, intent(OUT) :: iter                ! number of iterations
  logical, intent(IN) :: dirichlet

  if (dirichlet .eqv. .true.) then
    call solve_dirichlet(u,g,numx,numy,tolerance,h,iter)
  else
    call solve_neumann(u,g,numx,numy,tolerance,h,iter)
  end if

end subroutine


subroutine solve_dirichlet(u, g, numx, numy, tolerance, h,iter)
! Solves the 2D Poisson equation with Dirichlet boundary condition by
! using the Jacobi iteration. Solution is returned in matrix u, and number of
! iterations in variable iter
!
!
! Date/version: 30.04.2019/1.0

  implicit none
  integer, intent(IN) :: numx, numy             ! number of nodes is numx*numy
  real, intent(IN) :: h                     ! step size
  double precision, dimension(numx,numy), intent(IN) :: g   ! rhs of Poisson eq
  double precision, intent(IN) :: tolerance         ! error tolerance
  double precision, dimension(numx,numy), intent(OUT) :: u  ! solution matrix
  integer, intent(OUT) :: iter                ! number of iterations
  integer :: i, j                        ! loop variables
  double precision, dimension(numx,numy) :: udiff       ! differences between current
     and prev. iteration
  double precision, dimension(numx,numy) :: u_new       ! vector with new iterate
  double precision :: diff                  ! difference between rms of current and
    prev. iteration
  logical :: done                        ! controls if were done iterating
  double precision r8mat_rms                  ! function for calculating rms

  u = 0     ! initial guess for solution

  iter = 1
  done = .false.
  do while (.not. done)
    do j=1, numy
      do i=1, numx
      if (i==1 .or. j==1 .or. i==numx .or. j==numy) then
        ! on boundary, so value is known
        u_new(i,j)=g(i,j)
      else
        ! use five-point formula for unknown values on interior
        u_new(i,j) = 0.25*(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1)-g(i,j)*h**2)
      end if
      end do
    end do
    udiff = u_new-u
    diff = r8mat_rms(numx,numy,udiff)
    if (diff <= tolerance) then
      done = .true.
    else
      u=u_new
    end if
    iter=iter+1
  end do

end subroutine

subroutine solve_neumann(u,g,numx,numy,tolerance,h,iter)
! Solves the Poisson equation on the square unit grid,
! with Neumann boundary condition du/dn=0, using the Jacobi method.
! Returns the solution in the matrix u and number of iterations performed in iter.
!
!
! Date/version: 30.04.2019/1.0

  implicit none
  integer, intent(IN) :: numx, numy              ! number of nodes is numx*numy
  double precision, dimension(numx,numy), intent(IN) :: g   ! rhs of Poisson eq
  double precision, intent(IN) :: tolerance          ! error tolerance
  real, intent(IN) :: h                     ! step size
  integer, intent(OUT) :: iter                 ! number of iterations
  double precision, dimension(numx,numy), intent(OUT) :: u  ! solution matrix
  double precision :: diff                    ! difference between rms of current and
    prev. iteration
  double precision, dimension(numx,numy) :: udiff       ! differences between current
     and prev. iteration
  double precision, dimension(numx,numy) :: u_new       ! matrix with most recent
    iterates
  integer :: i, j                         ! loop variables
  logical :: done                         ! controls whether we're done iterating
  double precision r8mat_rms                   ! function for returning rms of data

  u = 0   ! initial guess for solution

  done = .false.
  iter = 1
  do while (.not. done)
    do j=1,numy
      do i=1,numx
      ! formula for u(i,j) is different for
      ! 1) the 4 corners
      ! 2) the 4 boundaries excluding corner points
      ! 3) interior
      ! => 9 different formulas for u(i,j):
        if (j==1) then
          if (i==1) then                 ! lower left corner
            u_new(i,j)=0
          else if (i==numx) then             ! upper left corner
            u_new(i,j)=0.25*(2*u(i-1,j)+2*u(i,j+1)-g(i,j)*h**2)
          else                      ! left boundary excluding corners
            u_new(i,j)=0.25*(u(i+1,j)+u(i-1,j)+2*u(i,j+1)-g(i,j)*h**2)
          end if
        else if (j==numy) then
          if (i==1) then                  ! lower right corner
            u_new(i,j)=0.25*(2*u(i+1,j)+2*u(i,j-1)-g(i,j)*h**2)
          else if (i==numx) then              ! upper right corner
            u_new(i,j)=0.25*(2*u(i-1,j)+2*u(i,j-1)-g(i,j)*h**2)
          else                        ! right boundary excluding corners
            u_new(i,j)=0.25*(u(i+1,j)+u(i-1,j)+2*u(i,j-1)-g(i,j)*h**2)
          end if
        else if (i==1 .and. j>1 .and. j<numy) then     ! lower boundary excluding corners
          u_new(i,j)=0.25*(2*u(i+1,j)+u(i,j-1)+u(i,j+1)-g(i,j)*h**2)
        else if (i==numx .and. j>1 .and. j<numy) then ! upper boundary excluding corners
          u_new(i,j)=0.25*(2*u(i-1,j)+u(i,j-1)+u(i,j+1)-g(i,j)*h**2)
        else                          ! interior
          u_new(i,j)=0.25*(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1)-g(i,j)*h**2)
        end if
      end do
    end do
    udiff = u_new-u
    diff = r8mat_rms(numx,numy,udiff)
    if (diff <= tolerance) then
      done = .true.
    else
      u=u_new
    end if
    iter = iter+1
  end do

end subroutine

subroutine poisson_solver_gsm( u,g,numx,numy,tolerance,iter,dirichlet,h,explicit )
! This routine solves the 2D Poisson equation using the Gauss-Seidel method. It
! makes the appropriate call to a subroutine depending on which variant of the GS method
! should be used. The algorithm used is either one using the explicit formulas for u(i,j) or
! a more general matrix formulation. In the latter case, this routine must also
! set up the system Au=b by making calls to subroutine.
!
!
! Date/version 01.05.2019/1.0

  implicit none
  integer, intent(IN) :: numx, numy              ! number of nodes is numx*numy
  logical, intent(IN) :: explicit                ! specifies variant of GS method
  double precision, dimension(numx,numy), intent(OUT) :: u  ! matrix containing solution u
  double precision, dimension(numx,numy), intent(IN) :: g   ! matrix containing rhs of Poisson eq.
  double precision, dimension(:,:), allocatable :: A      ! matrix A in Au=b
  double precision, dimension(:), allocatable :: b      ! vector b in Au=b
  integer :: n                         ! number of unknown variables
  double precision, intent(IN) :: tolerance
  logical, intent(IN) :: dirichlet
  integer, intent(OUT) :: iter
  real, intent(IN) :: h

  if (explicit .eqv. .true.) then
    call gsm_solver_explicit(u, g, numx, numy, tolerance, h,iter,dirichlet)
```

```fortran
  else
    if (dirichlet .eqv. .true.) then
      n = numx - 2
    else
      n = numx
    end if

    allocate(A(n**2,n**2))
    call build_A(A,n,dirichlet)

    allocate(b(n**2))
    call build_b(b,g,n,h,dirichlet,numx)

    call gsm_solver(u,A,b,g,numx,numy,tolerance,iter,n,dirichlet)

    ! deallocate allocated variables
    deallocate(A)
    deallocate(b)
  end if

end subroutine

subroutine gsm_solver_explicit(u, g, numx, numy, tolerance, h,iter, dirichlet)
! This routine solves the 2D Poisson equation by the iterative Gauss-Seidel method,
! and returns the solution in the matrix u and the number of iterations performed in
    the variable iter.
! It uses the explicit formulas for u(i,j), and essentially copies the code in
    solve_dirichlet and solve_neumann,
! replacing u with u_new at the appropriate places in each formula.
!
!
! Date/version: 02.05.2019/1.0


  implicit none
  integer, intent(IN) :: numx, numy              ! number of nodes is numx*numy
  real, intent(IN) :: h                    ! step size
  double precision, dimension(numx,numy), intent(IN) :: g   ! rhs of Poisson eq
  double precision, intent(IN) :: tolerance         ! error tolerance
  logical, intent(IN) :: dirichlet              ! boundary condition
  double precision, dimension(numx,numy), intent(OUT) :: u  ! solution matrix
  integer, intent(OUT) :: iter                 ! number of iterations
  integer :: i, j                         ! loop variables
  double precision, dimension(numx,numy) :: udiff       ! differences between current
     and prev. iteration
  double precision, dimension(numx,numy) :: u_new       ! vector with new iterate
  double precision :: diff                   ! difference between rms of current and
    prev. iteration
  logical :: done                         ! controls if were done iterating
  double precision r8mat_rms                   ! function for calculating rms

  u = 0     ! initial guess for solution
  iter = 1
  done = .false.

  if (dirichlet .eqv. .true.) then
    do while (.not. done)
      do j=1, numy
        do i=1, numx
        if (i==1 .or. j==1 .or. i==numx .or. j==numy) then
```

```
          ! on boundary, so value is known
          u_new(i,j)=g(i,j)
        else
          ! use five-point formula for unknown values on interior
          u_new(i,j) = 0.25*(u_new(i-1,j)+u(i+1,j)+u_new(i,j-1)+u(i,j+1)-g(i,j)*h**2)
        end if
        end do
      end do
      udiff = u_new-u
      diff = r8mat_rms(numx,numy,udiff)
      if (diff <= tolerance) then
        done = .true.
      else
        u=u_new
      end if
      iter=iter+1
    end do
  else
    do while (.not. done)
      do j=1,numy
        do i=1,numx
        ! formula for u(i,j) is different for
        ! 1) the 4 corners
        ! 2) the 4 boundaries excluding corner points
        ! 3) interior
        ! => 9 different formulas for u(i,j):
          if (j==1) then
            if (i==1) then                    ! lower left corner
              u_new(i,j)=0
            else if (i==numx) then                ! upper left corner
              u_new(i,j)=0.25*(2*u_new(i-1,j)+2*u(i,j+1)-g(i,j)*h**2)
            else                        ! left boundary excluding corners
              u_new(i,j)=0.25*(u(i+1,j)+u_new(i-1,j)+2*u(i,j+1)-g(i,j)*h**2)
            end if
          else if (j==numy) then
            if (i==1) then                    ! lower right corner
              u_new(i,j)=0.25*(2*u(i+1,j)+2*u_new(i,j-1)-g(i,j)*h**2)
            else if (i==numx) then                ! upper right corner
              u_new(i,j)=0.25*(2*u_new(i-1,j)+2*u_new(i,j-1)-g(i,j)*h**2)
            else                        ! right boundary excluding corners
              u_new(i,j)=0.25*(u(i+1,j)+u_new(i-1,j)+2*u_new(i,j-1)-g(i,j)*h**2)
            end if
          else if (i==1 .and. j>1 .and. j<numy) then    ! lower boundary excluding corners
            u_new(i,j)=0.25*(2*u(i+1,j)+u_new(i,j-1)+u(i,j+1)-g(i,j)*h**2)
          else if (i==numx .and. j>1 .and. j<numy) then ! upper boundary excluding corners
            u_new(i,j)=0.25*(2*u_new(i-1,j)+u_new(i,j-1)+u(i,j+1)-g(i,j)*h**2)
          else                          ! interior
            u_new(i,j)=0.25*(u_new(i-1,j)+u(i+1,j)+u_new(i,j-1)+u(i,j+1)-g(i,j)*h**2)
          end if
        end do
      end do
      udiff = u_new-u
      diff = r8mat_rms(numx,numy,udiff)
      if (diff <= tolerance) then
        done = .true.
      else
        u=u_new
      end if
```

```fortran
      iter = iter+1
    end do
  end if

end subroutine


subroutine build_A(A, n, dirichlet)
! This routine builds the matrix A in Au=b. The A matrix is formed from matrices D
    and the identity matrix.
! The size and structure of A and D depends on whether the Dirichlet or Neumann b.c
    is active.
!
!
! Date/version: 30.04.2019/1.0

  integer, intent(in) :: n                    ! number of unknowns in system
  logical, intent(in) :: dirichlet               ! Dirichlet boundary condition
  double precision, dimension(n**2,n**2), intent(out) :: A  ! system matrix A
  double precision, dimension(n,n) :: eye, D           ! helper matrices for
    constructing A

  ! build identity matrix eye
  do j=1,n
    do i=1,n
      if (i==j) then
        eye(i,j)=1
      else
        eye(i,j)=0
      end if
    end do
  end do

  ! build D matrix
  do j=1,n
    do i=1,n
      if (i==j) then
        D(i,j)=4
        if (i==1) then
          if (dirichlet .eqv. .false.) then
            D(i,j+1) = -2
          else
            D(i,j+1) = -1
          end if
          D(i+1,j) = -1
        else if (i==n) then
          if (dirichlet .eqv. .false.) then
            D(i,j-1)=-2
          else
            D(i,j-1)=-1
          end if
          D(i-1,j)=-1
        else
          D(i+1,j)=-1
          D(i,j+1)=-1
        end if
      end if
    end do
  end do

  ! build A by inserting D and eye at the appropriate
  ! indices.
  do i=0,n*(n-1),n
    A(i+1:i+n,i+1:i+n)=D
    if (i==0) then
      if (dirichlet .eqv. .false.) then
        A(i+1:i+n,i+n+1:i+2*n)=-2*eye
      else
        A(i+1:i+n,i+n+1:i+2*n)=-eye
      end if
    else if (i==n*(n-1)) then
      if (dirichlet .eqv. .false.) then
        A(i+1:i+n,i-n+1:i)=-2*eye
      else
        A(i+1:i+n,i-n+1:i)=-eye
      end if
    else
      A(i+1:i+n,i-n+1:i)=-eye
      A(i+1:i+n,i+n+1:i+2*n)=-eye
    end if
  end do

  ! apply Neumann b.c to A if it is active
  if (dirichlet .eqv. .false.) then
    do i=2,n*n
      A(1,i)=0
    end do
  end if

end subroutine

subroutine build_b(b,g,n,h,dirichlet,numx)
! This routine creates the b vector in Au=b. If the Dirichlet b.c is applied, the
  known boundary values of u
! must be included in b. The other values in b are simply retrieved from g
!
!
! Date/version: 30.04.2019/1.0

  implicit none
  integer, intent(in) :: n                ! number of unknowns in each row
  integer, intent(in) :: numx             ! grid size is numx*numx
  double precision, dimension(numx,numx), intent(in) :: g ! rhs of Poisson eq. and
    known boundary points if Dirichlet b.c
  real, intent(in) :: h                   ! step size
  logical, intent(in) :: dirichlet        ! Dirichlet b.c
  double precision, dimension(n**2), intent(out) :: b   ! vector b in Au=b
  integer :: counter, k, l, i, j          ! helper variables for counting,
    looping/indexing

  counter=1

  if (dirichlet .eqv. .true.) then
  ! must remember to include known values at the boundaries in b.
  ! These values are stored in g.
    k=2
    l=numx-1
    ! loop over interior points as these are the unknowns:
    do j=k,l
      do i=k,l
```

```
        if (i==k) then
          if (j==k) then
            b(counter)=g(i-1,j)+g(i,j-1)-g(i,j)*h**2
          else if (j==l) then
            b(counter)=g(i-1,j)+g(i,j+1)-g(i,j)*h**2
          else
            b(counter)=g(i-1,j)-g(i,j)*h**2
          end if
        else if (i==l) then
          if (j==k) then
            b(counter)=g(i+1,j)+g(i,j-1)-g(i,j)*h**2
          else if (j==l) then
            b(counter)=g(i+1,j)+g(i,j+1)-g(i,j)*h**2
          else
            b(counter)=g(i+1,j)-g(i,j)*h**2
          end if
        else if (j==k .and. i>k .and. i<l) then
          b(counter)=g(i,j-1)-g(i,j)*h**2
        else if (j==l .and. i>k .and. i<l) then
          b(counter)=g(i,j+1)-g(i,j)*h**2
        else
          b(counter)=-g(i,j)*h**2
        end if
        counter=counter+1
      end do
    end do
  else
  ! values in b are found from g
    k=1
    l=numx
  ! loop over all points on grid:
    do j=k,l
      do i=k,l
        b(counter)=-g(i,j)*h**2
        counter=counter+1
      end do
    end do
  end if

  ! apply Neumann b.c to b if it is active
  if (dirichlet .eqv. .false.) then
    b(1)=0
  end if

end subroutine

subroutine gsm_solver(u,A,b,g,numx,numy, tolerance, iter,n, dirichlet)
! This routine solves the 2D Poisson equation by the iterative Gauss-Seidel method,
! and returns the solution in the matrix u and the number of iterations performed in
  the variable iter.
! It uses the more general formulation of the algorithm:
! https://www.cfd-online.com/Wiki/Gauss-Seidel_method
!
!
! Date/version: 30.04.2019/1.0

  implicit none
  integer, intent(in) :: numx, numy              ! number of nodes is numx*numy
  integer, intent(in) :: n                   ! number of unknowns
  double precision, dimension (n**2,n**2), intent(in) :: A  ! matrix A in Au=b
  double precision, dimension(numx,numy), intent(in) :: g   ! matrix g with rhs of
    Poisson equation
  double precision, dimension(n**2), intent(in) :: b       ! vector b in Au=b
  double precision, intent(in) :: tolerance         ! tolerance used for terminating
    iterations
  logical, intent(in) :: dirichlet              ! dirichlet boundary condition
  double precision, dimension(numx,numy), intent(out) :: u  ! solution matrix
  integer, intent(out) :: iter                  ! number of iterations
  double precision :: left_sum, right_sum     ! left_sum/right_sum is sum of A(i,j)*x(
    j) to the LEFT/RIGHT  of diagonal of A
  logical :: done                           ! logical for checking convergence
  double precision :: diff                    ! difference between rms of new and old
    iterate
  double precision, dimension(n*n) :: x, x_new         ! vectors containing old and
    new iterate
  double precision, dimension(n*n) :: x_diff           ! vector containing difference
    between iterates
  double precision r8mat_rms                    ! function for returning rms of data
  integer :: i,j,counter                      ! iteration and counter variables

  x = 0
  x_new = 0
  done = .false.
  iter = 1

  do while (.not. done)
    do i = 1,n*n
      left_sum = 0
      right_sum = 0
      do j = 1,i-1
        left_sum = left_sum+A(i,j)*x_new(j)
      end do
      do j = i+1,n*n
        right_sum = right_sum+A(i,j)*x(j)
      end do
      x_new(i) = (1.0/A(i,i))*(b(i)-left_sum-right_sum)
    end do
    x_diff = x_new-x
    diff = r8mat_rms(n*n,1,x_diff)
    if (diff <= tolerance) then
      done = .true.
    else
      x=x_new
    end if
    iter = iter+1
  end do

  ! output the solution as a matrix with correct boundary values
  counter = 1
  do j=1,numx
    do i=1,numx
      if (dirichlet .eqv. .true.) then
        if (i==1 .or. i==numx .or. j==1 .or. j==numx) then
          u(i,j)=g(i,j)
        else
          u(i,j) = x_new(counter)
          counter = counter+1
        end if
      else
        u(i,j) = x_new(counter)
```

```fortran
        counter = counter+1
      end if
    end do
  end do

end subroutine
```

B C-kode

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "GL/glut.h"

static int numx, numy;          /// numx*numy = number of nodes in equation system
static float **fvals;           /// solution of equation
static float min, max;          /// min and max of fvals
static float h;                 /// grid step size
static GLuint display_list;     /// display list for scene

GLfloat rotV=0.0f;              /// defines rotation angle about vertical axis
GLfloat rotH=0.0f;              /// defines rotation angle about a horizontal axis
GLfloat vspeed=0.0f;            /// vertical rotation speed
GLfloat hspeed=0.0f;            /// horizontal rotation speed
GLfloat scale=1.0f;             /// scaling variable

void readFile(char* fileName) {
    /*
    ***************************************************************************************

    * PURPOSE:
    * Read file to initialize global variables 'numx', 'numy' and 'h', and
    * read function values into global matrix 'fvals'
    *
    * Date/version: 03.05.2019/1.0

    ***************************************************************************************
    */
  FILE* file;
  file = fopen(fileName, "r");
  if (!file) printf("Data file not found: %s", fileName);

  fscanf(file," %i %i", &numx, &numy); ///read numx and numy in from first line of file
    h = 1.0/(numx-1);                       ///set grid step-size h

  ///Allocate memory to matrix 'fvals':
  ///RETRIVED FROM: http://pleasemakeanote.blogspot.com/2008/06/2d-arrays-in-c-using-malloc.html
  fvals = (float**) malloc((numx)*sizeof(float*));
    for (int i = 0; i < numx; i++){
        fvals[i] = (float*) malloc((numx)*sizeof(float));
    }

  /**Read function values into matrix.
  * Function values are stored in file as a list.
  * Each line contains the calculated function value at point (x,y), starting at (0,0).
  * Function values are organized in COLUMN-MAJOR order in file.
  **/
  for(int n=0; n < numx; n++) {                /// Loop over columns first,
        for(int m=0; m < numy; m++) {             /// then loop over rows.
            fscanf(file,"%f", &fvals[n][m]);        /// Read values directly into matrix
                                                  /// by exploiting ordering of file
            ///find max and min function values:
            float z=fvals[n][m];
            if ((n==0) & (m==0)) {
                max=z;
                min=z;
            }
            else if (z>max){
                max=z;
            }
            else if (z<min){
                min=z;
            }

        }
  }

  fclose(file);

}

void renderBitmapString(
    /******************
    * PURPOSE:
    * Render a string starting at specified raster position
    *
    * RETRIEVED FROM: https://www.lighthouse3d.com/tutorials/glut-tutorial/bitmap-
    fonts/
    *
    ******************/
  float x,              /// x position
  float y,              /// y position
  float z,              /// z position
  void *font,           /// 'font' is chosen font
  char *string) {       /// 'string' string to render
  char *c;
  glRasterPos3f(x, y,z);
  for (c=string; *c != '\0'; c++) {
    glutBitmapCharacter(font, *c);
  }
}

void init() {
    /***********
    * PURPOSE:
    * Setup for OpenGL.
    * Compile surface plot, grid and x-, y-, z-axes for later execution.
    *
    * Date/version: 03.05.2019/1.0
    ************/
    glClearColor(1.0, 1.0, 1.0, 0);                       /// set background to white
    glEnable(GL_BLEND);                                   /// enables blending
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);    /// alpha blending for
    transparency

    ///set up axes and scene, and store result in display list:
    glColor3f(1.0,1.0,1.0);
    display_list = glGenLists(1);
    glNewList(display_list, GL_COMPILE);
        ///make surface:
        glColor4f(0,255,255,0.5);             ///cyan, somewhat transparent
        for (int i=0; i<numx-1; i++)    {
            for (int j=0; j<numy-1; j++)    {
                glBegin(GL_POLYGON);
                float x = i*h;              ///define proper x-,
                float y = j*h;              ///y-,
                float z = fvals[i][j];      ///and z-coordinates
                glVertex3f(x,y,z);
                x = i*h;
                y = (j+1)*h;
                z = fvals[i][j+1];
                glVertex3f(x,y,z);
                x = (i+1)*h;
                y = (j+1)*h;
                z = fvals[i+1][j+1];
                glVertex3f(x,y,z);
                x = (i+1)*h;
                y = (j)*h;
                z = fvals[i+1][j];
                glVertex3f(x,y,z);
            glEnd();
        }
    }

        ///make grid on surface:
        glColor3f(0.0f,0.0f,1.0f);              ///set grid color to pure blue
        for (int i=0; i<numx-1; i++)
    {
            for (int j=0;j<numy-1; j++)
        {
                glBegin(GL_LINE_LOOP);
                    float x = i*h;
                    float y = j*h;
                    float z = fvals[i][j];
                    glVertex3f(x,y,z);
                    x = i*h;
                    y = (j+1)*h;
                    z = fvals[i][j+1];
                    glVertex3f(x,y,z);
                    x = (i+1)*h;
                    y = (j+1)*h;
                    z = fvals[i+1][j+1];
                    glVertex3f(x,y,z);
                    x = (i+1)*h;
                    y = (j)*h;
                    z = fvals[i+1][j];
                    glVertex3f(x,y,z);
                glEnd();
        }
    }

        /// display values at (1,1,z), (0,1,z), (1,0,z)
        glColor3f(0.0f,0.0f,0.0f);
        char buffer[10]={'\0'};
        sprintf(buffer, "z = %f", fvals[numx-1][numy-1]);
        renderBitmapString(1.0f, 1.0f , fvals[numx-1][numy-1]*(1.0f+0.01f),
    GLUT_BITMAP_HELVETICA_12, buffer);
        sprintf(buffer, "z = %f", fvals[0][numy-1]);
        renderBitmapString(-0.22f, 1.22f , fvals[0][numy-1]*(1.0f+0.01f),
    GLUT_BITMAP_HELVETICA_12, buffer);
        sprintf(buffer, "z = %f", fvals[numx-1][0]);
        renderBitmapString(1.0f, -0.05f , fvals[numx-1][0]*(1.0f+0.01f),
    GLUT_BITMAP_HELVETICA_12, buffer);
```

```

        /// make vertical dashed lines from floor to displayed values:
        glLineStipple(1, 0x00FF);
        glEnable(GL_LINE_STIPPLE);
        glBegin(GL_LINE_STRIP);
        glVertex3f(0.0f,1.0f,0.0f);
        glVertex3f(0.0f,1.0f,fvals[0][numy-1]);
        glEnd();

        glBegin(GL_LINE_STRIP);
        glVertex3f(1.0f,0.0f,0.0f);
        glVertex3f(1.0f,0.0f,fvals[numx-1][0]);
        glEnd();

        glBegin(GL_LINE_STRIP);
        glVertex3f(1.0f,1.0f,0.0f);
        glVertex3f(1.0f,1.0f,fvals[numx-1][numy-1]);
        glEnd();

        glBegin(GL_LINE_LOOP);
        glVertex3f(0.0f,0.0f,0.0f);
        glVertex3f(0.0f,1.0f,0.0f);
        glVertex3f(1.0f,1.0f,0.0f);
        glVertex3f(1.0f,0.0f,0.0f);
        glEnd();

        glDisable(GL_LINE_STIPPLE);

        /// x-axis:
        glColor3f(0, 0, 0);
        glBegin(GL_LINE_STRIP);
            glVertex3f(0.0f, 0.0f, 0.0f);
            glVertex3f(1.3f, 0.0f, 0.0f);
        glEnd();

        glBegin(GL_TRIANGLES);
            glVertex3f(1.25f, 0.0f, 0.01f);
            glVertex3f(1.25f, 0.0f, -0.01f);
            glVertex3f(1.3f, 0.0f, 0.0f);
        glEnd();

        /// y-axis:
        glBegin(GL_LINE_STRIP);
            glVertex3f(0.0f, 0.0f, 0.0f);
            glVertex3f(0.0f, 1.3f, 0.0f);
        glEnd();

        glBegin(GL_TRIANGLES);
            glVertex3f(0.0f, 1.3f, 0.0f);
            glVertex3f(0.0f, 1.25f, 0.01f);
            glVertex3f(0.0f, 1.25f, -0.01f);
        glEnd();

        /// z-axis:
        glBegin(GL_LINE_STRIP);
            glVertex3f(0.0f, 0.0f, 0.0f);
            glVertex3f(0.0f, 0.0f, max);
        glEnd();

        glBegin(GL_TRIANGLES);
```

```
            glVertex3f(0.0f, 0.0f, max);
            glVertex3f(0.01f*max, 0.0f, 1.25f/1.3f*max);
            glVertex3f(-0.01f*max, 0.0f, 1.25f/1.3f*max);
        glEnd();

        glBegin(GL_TRIANGLES);
            glVertex3f(0.0f, 0.00f, max);
            glVertex3f(0.0f, -0.01f*max, 1.25f/1.3f*max);
            glVertex3f(0.0f, 0.01f*max, 1.25f/1.3f*max);
        glEnd();

        ///name x-, y- and z-axis:
        glColor3f(0, 0, 0);
        glRasterPos3f(1.25f, 0.05f, 0.0f);
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, 'x');
        glRasterPos3f(1.05f, 0.075f, 0.0f);
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, '1');
        glRasterPos3f(0.05f, 1.25f, 0.0f);
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, 'y');
        glRasterPos3f(0.075f, 1.05f, 0.0f);
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, '1');
        glRasterPos3f(0.0f, 0.05f, max);
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, 'z');


    glEndList();

}

void display()  {
    /************
    *PURPOSE:
    * Draws 3D-plot of solution as surface with grid
    *
    * Date/version: 03.05.2019/1.0
    *************/

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); /// clear screen and buffer
    glMatrixMode(GL_MODELVIEW);                         /// use modelview matrix
    glLoadIdentity();                                   /// reset matrix
    gluLookAt(-max, -max, max,                          /// eye position
              0.0f, 0.0f, (max-min)*1.0f/2,             /// reference point
              0.0f, 0.0f, 1.0f);                        /// up vector
    glScalef(scale,scale,scale);                        /// initial scale=1
    glRotatef(rotV,1,-1,0);                             /// initial v. rot.=0
    glRotatef(rotH,0,0,1);                              /// initial h. rot.=0


    ///draw scene:
    glPushMatrix();
    glCallList(display_list);
    glPopMatrix();

    ///increment horizontal/vertical rot. speed (user controllable):
    rotV+=vspeed;
    rotH+=hspeed;

    glutSwapBuffers();
}

```

```
void reshape(int w, int h){
    /***************
    * PURPOSE:
    * In case of change of window height/width, this function maintains the correct
    perspective
    *
    * Date/version: 03.05.2019/1.0
    ****************/

    if (h==0) h=1;                  /// prevents division by zero in next line
    float ratio = w*1.0 / h;

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glViewport(0, 0, w, h);
    gluPerspective(45,ratio,0,1000);
    glMatrixMode(GL_MODELVIEW);

}

void keyPress(int key, int xx, int yy) {
    /********************
    * PURPOSE:
    * Process keypresses to rotate and scale figure
    *
    * Date/version: 03.05.2019/1.0
    *********************/

  switch (key) {
      case GLUT_KEY_UP:   /// rotate fig. "towards" camera
            vspeed+=0.005f;
            break;
      case GLUT_KEY_DOWN: /// rotate fig. "away" from camera
            vspeed-=0.005f;
            break;
        case GLUT_KEY_LEFT: /// rotate clockwise about vertical axiz
            hspeed-=0.005f;
            break;
        case GLUT_KEY_RIGHT:/// rotate counterclockwise about vertical axis
            hspeed+=0.005f;
            break;
        case GLUT_KEY_F1:   /// scale down
            scale-=0.05;
            break;
        case GLUT_KEY_F2:   /// scale up
            scale+=0.05;
            break;
  }
}

int main(int argc, char** argv)
{   /*************************
    * PURPOSE:
    * The main function initializes and calls
    * functions in the right order for program to run
    * successfully
    *
    * Date/version: 03.05.2019/1.0
    ******************************/
    readFile("solution.DAT");
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
    glutInitWindowSize(800,600);
    glutInitWindowPosition(100,100);
    glutCreateWindow("Visualization of Poisson equation solution");
    glutSpecialFunc(keyPress);       /// process keypresses
    glutDisplayFunc(display);        /// callback function
    glutIdleFunc(display);           /// callback function
    glutReshapeFunc(reshape);        /// callback function

    init();
    glutMainLoop();                  ///event processing cycle

    return 0;
}
```

C MATLAB-kode

```matlab
clear all
% import solutions:
A1 = importdata('solution1.dat');
vec1 = A1(2:length(A1),1);
A1 = vec2mat(vec1,21);
A2 = importdata('solution2.dat');
vec2 = A2(2:length(A2),1);
A2 = vec2mat(vec2,21);
A3 = importdata('solution3.dat');
vec3 = A3(2:length(A3),1);
A3 = vec2mat(vec3,21);

% create analytical solution:
syms x y
f1(x,y)=0.25*(x^2+y^2);
f2(x,y)=3*(x^2+y^2)-2*(x^3+y^3);
f3(x,y)=(3*x^2-2*x^3)*(3*y^2-2*y^3);
g(x,y)=12-12*x-12*y;
F1=zeros(10,10);
F2=zeros(10,10);
F3=zeros(10,10);
h=0.05;
for i = 0:20
    for j = 0:20
        F1(i+1,j+1)=f1(h*i,h*j);
        F2(i+1,j+1)=f2(h*i,h*j);
        F3(i+1,j+1)=f3(h*i,h*j);
    end
end
% calculate largest difference between analytical and numerical solution:
max_diff1=max(max(abs(F1-A1)));
max_diff2=max(max(abs(F2-A2)));
max_diff3=max(max(abs(F3-A3)));
```