Numerical Modelling in FORTAN
day 6
Paul Tackley, 2017
Today‘s Goals

1. Learn about **pointers, generic procedures and operators**
2. Learn about **iterative** solvers for boundary value problems, including the **multigrid** method
3. Practice, practice! This time programming an iterative solver for **Poisson’s equation**.

- Useful for e.g., gravitational potential, electromagnetism, convection (streamfunction-vorticity formulation)
Projects: start thinking about

Agree topic with me before final lecture
Pointers

• Point to either
  – (i) data stored by a variable declared as a target, or
  – (ii) another pointer, or
  – (iii) an area of allocated memory

• (i) and (ii) are mainly useful for creating interesting data structures (like linked lists); for us (iii) will be the most useful.

• “=>” is used to point a pointer at something

• See examples next slide

• Functions can also return pointers ("pointer functions")
Pointers to scalar variables

program pointless
  implicit none
  real,pointer:: p
  real,target:: a=3.1,b=4.7

  p=>a           ! point p to a
  print*,p       ! prints 3.1 (value of a)
  p=4.0          ! changes a to 4.0
  print*,a       ! prints 4.0
  p=>b           ! now points to b
  print*,b       ! prints 4.7
  print*,associated (p) ! test status: prints true
  nullify(p)     ! points p at nothing
  print*,associated (p) ! prints false
end program pointless
Pointers to array or array section

program pointarray
  implicit none
  integer, parameter:: n=7
  real, target:: T(n,n)
  real, pointer:: bot_boundary(:,), &
                 top_boundary(:,), center4(:, :)

  top_boundary  =>  T(:, 1)
  bot_boundary  =>  T(:, n)
  center4       =>  T(n/2:n/2+1, n/2:n/2+1)

  call random_number(T)
  top_boundary = 0.0 ! easy way to do
  bot_boundary = 1.0 ! boundary conditions
  center4 = 1.0

  print *, T
end program pointarray
Pointers in defined types
similar to allocatable array (i.e., not pointing to a variable)

```fortran
program pointertype
  implicit none

  type array2D
    real, pointer:: a(:, :) ! 2D array
  end type array2D

  type(array2D), allocatable:: T(:) ! 1D array of 2D arrays
  integer i, n, ng

  print '(a,$)', "Number of multigrid levels:
  read*, ng
  allocate (T(ng)) ! allocate number of grids
  do i = 1, ng
    n = 2**(i+1) ! #grid points in this grid
    print*, 'Allocating grid', n, n
    allocate (T(i)%a(n,n)) ! each grid
  end do

  !.... rest of program

end program pointertype```

Hint: arrays in subprograms

- Arrays that are declared using subroutine arguments have a limited size, e.g.

```fortran
Subroutine do_something (n,m)
  Integer,intent(in):: n,m
  Real:: local_array(n,m)
End Subroutine do_something
```

- Better to use allocatable arrays

```fortran
Subroutine do_something (n,m)
  Integer,intent(in):: n,m
  Real,allocatable:: local_array(:,:)
  allocate(local_array(n,m))
End Subroutine do_something
```
Generic procedures  
(i.e., using a generic name to access different procedures)

• e.g., the intrinsic (built-in) \texttt{sqrt} function. There are several versions: \texttt{sqrt}(real), \texttt{dsqrt}(double precision), \texttt{csqrt}(complex). Use the generic name and the correct one will automatically be used.

• You can define the same thing. Define similar sets of procedures then define a generic \texttt{interface} to them (see example next slide).

• Easiest way: in a \texttt{module}
module goodstuff
  implicit none

  interface apbxc  ! define generic interface
    module procedure rapbxc,iapbxc
  end interface

contains

  real function rapbxc(a,b,c) ! actual function
    real,intent(in):: a,b,c
    rapbxc = a+b*c
  end function rapbxc

  integer function iapbxc(a,b,c)
    integer,intent(in):: a,b,c
    iapbxc = a+b*c
  end function iapbxc

end module goodstuff

!--------------------------------------------------
program generic
  use goodstuff
  implicit none
  real:: a=1.2,b=3.4,c=5.6
  integer:: i=2,j=3,k=4

  print*,apbxc(a,b,c)  ! real arguments
  print*,apbxc(i,j,k)  ! integer arguments

end program generic
Overloading

- **Overloading** means that one operator or procedure name is used to refer to several procedures: which one is used depends on the variable types.

- **Examples**
  - `apbxc` is overloaded with `rapbxc` and `iapbxc`
  - `*,+,-,/` are overloaded with integer, real, complex versions in different precisions

- You can overload existing operators, or define new overloaded operators or procedures
Overloading existing + and –

Useful for defined types

module coordstuff
    implicit none

    type point ! defined type
        real:: x,y,z
    end type point

    interface operator (+) ! new version of +
        module procedure pointplus
    end interface

    interface operator (-) ! new version of –
        module procedure pointminus
    end interface

contains

    function pointplus(a,b)
        type(point):: pointplus
        type(point),intent(in):: a,b
        pointplus%x = a%x + b%x
        pointplus%y = a%y + b%y
        pointplus%z = a%z + b%z
    end function pointplus

    function pointminus(a,b)
        type(point):: pointminus
        type(point),intent(in):: a,b
        pointminus%x = a%x - b%x
        pointminus%y = a%y - b%y
        pointminus%z = a%z - b%z
    end function pointminus
end module coordstuff

program test
    use coordstuff
    type(point):: p1,p2,p3
    p1%x=1.2; p1%y=0. ; p1%z=3.1
    p2%x=0. ; p2%y=1.2; p2%z=1.7

    p3 = p1 - p2 ! using overloaded –
    print*,p3
    p3 = p1 + p2 ! using overloaded +
    print*,p3
end program test
Defining new operator .distance.

```fortran
module coordsagain
  implicit none

  type point                ! defined type
    real:: x,y,z
  end type point

  interface operator (.distance.) ! new operator
    module procedure pointseparation
  end interface

  contains

  real function pointseparation(a,b)
    type(point),intent(in):: a,b
    pointseparation = sqrt( &
                  (a%x-b%x)**2+(a%y-b%y)**2+(a%z-b%z)**2)
  end function pointseparation

end module coordsagain

!-----------------------------------
program test
  use coordsagain
  type(point):: p1,p2
  real:: d
  p1%x=1.2; p1%y=0. ; p1%z=3.1
  p2%x=0. ; p2%y=1.2; p2%z=1.7

  d = p1.distance.p2       ! using new operator
  print*,d
end program test
```
Overloading "=". Useful for conversion between different types: in this case point to real.

Must use subroutine with 1st argument intent(out) and 2nd argument intent(in).

```fortran
module coords3
    implicit none

    type point           ! defined type
        real:: x,y,z
    end type point

    interface assignment (=) ! new "="
        module procedure absvec ! converts
    end interface           ! point to real

contains

    subroutine absvec(a,b) ! calculates distance
        real,intent(out):: a ! from origin
        type(point),intent(in):: b
        a = sqrt(b%x**2+b%y**2+b%z**2)
    end subroutine absvec

end module coords3

!--------------------------------------------------------
program test
    use coords3
    type(point):: p1
    real:: d
    p1%x=1.2; p1%y=0. ; p1%z=3.1

    d = p1        ! using new =,
    print*,d
end program test
```
Combine operators into 1 module

- Generic procedures
- New operators
- Overloaded operators (e.g., +, -, =)
The interfaces

module coords
  implicit none
  type point ! defined type
    real :: x, y, z
  end type point
end module

interface apbxc ! define generic interface
  module procedure rapbxc, iapbxc
end interface

interface operator (.distance.) ! new operator
  module procedure pointseparation
end interface

interface operator (+) ! new version of +
  module procedure pointplus
end interface

interface operator (-) ! new version of -
  module procedure pointminus
end interface

interface assignment (=) ! new "="
  module procedure absvec ! converts point to real
end interface

contains
The procedures

```
real function rapbxc(a,b,c) ! real version of fn
  real, intent(in) :: a,b,c
  rapbxc = a+b*c
end function rapbxc

integer function iapbxc(a,b,c) ! int version
  integer, intent(in) :: a,b,c
  iapbxc = a+b*c
end function iapbxc

real function pointseparation(a,b) ! for .distance.
  type(point), intent(in) :: a,b
  pointseparation = sqrt((a%x-b%x)**2+(a%y-b%y)**2+(a%z-b%z)**2)
end function pointseparation

function pointplus(a,b) ! for +
  type(point), intent(in) :: a,b
  pointplus%x = a%x + b%x
  pointplus%y = a%y + b%y
  pointplus%z = a%z + b%z
end function pointplus

function pointminus(a,b) ! for -
  type(point), intent(in) :: a,b
  pointminus%x = a%x - b%x
  pointminus%y = a%y - b%y
  pointminus%z = a%z - b%z
end function pointminus

subroutine absvec(a,b) ! for = (distance
  real, intent(out) :: a ! from origin)
  type(point), intent(in) :: b
  a = sqrt(b%x**2+b%y**2+b%z**2)
end subroutine absvec
```

end module coords
program test
  use coords
  real:: a=1.2,b=3.4,c=5.6,d
  integer:: i=2,j=3,k=4
  type(point):: p1,p2,p3
  p1%x=1.2;  p1%y=0.  ;  p1%z=3.1
  p2%x=0. ;  p2%y=1.2;  p2%z=1.7

  print*,apbxc(a,b,c)  ! real arguments
  print*,apbxc(i,j,k)  ! integer arguments

  p3  = p1  -  p2   ! using overloaded -
  p3  = p1  +  p2   ! using overloaded +

  d  = p1           ! using overloaded =,
  d  = p1.distance.p2 ! using new operator
end program test
Review last week: Advection-diffusion for fixed flow field

(i) Calculate velocity at each point using centered derivatives

\[
(v_x, v_y) = \left( \frac{\partial \psi}{\partial y}, - \frac{\partial \psi}{\partial x} \right)
\]

(ii) Take timesteps to integrate the advection-diffusion equation for the specified length of time using UPWIND finite-differences for \( \frac{dT}{dT} \) and \( \frac{dT}{dy} \)

\[
\frac{\partial T}{\partial t} = -v_x \frac{\partial T}{\partial x} - v_y \frac{\partial T}{\partial y} + \nabla^2 T
\]
This is what it should look like

B=0.1

B=1

B=10

B=100
The next step: Calculate velocity field from temperature field (=> convection) e.g., for highly viscous flow (e.g., Earth’s mantle) with constant viscosity (P=pressure, Ra=Rayleigh number):

\[-\nabla P + \nabla^2 \vec{v} = -Ra \hat{y}\]

Substituting the streamfunction for velocity, we get:

\[\nabla^4 \psi = -Ra \frac{\partial T}{\partial x}\]

writing as 2 Poisson equations:

\[\nabla^2 \psi = -\omega\]

\[\nabla^2 \omega = Ra \frac{\partial T}{\partial x}\]

the streamfunction-vorticity formulation
we need a Poisson solver

• An example of a **boundary value problem** (uniquely determined by interior equations and values at boundaries), as compared to

• **initial value problems** (depend on initial conditions as well as boundary conditions, like the diffusion equation)

Siméon Denis Poisson (1781-1840)
Initial value vs boundary value problems...often, the problem is both

• (a) initial value problem
• (b) Boundary value problem
Example: 1D Poisson

Poisson: \( \nabla^2 u = f \)

Solve for \( u \) given \( f \) and \( u_0 = 0 \) for the boundary condition.

In 1-D: \( \frac{\partial^2 u}{\partial x^2} = f \)

Finite-difference form: \( \frac{1}{h^2} (u_{i-1} - 2u_i + u_{i+1}) = f_i \)

Example with 5 grid points:

\[
\begin{align*}
    u_0 &= 0 \\
    u_0 - 2u_1 + u_2 &= h^2 f_1 \\
    u_1 - 2u_2 + u_3 &= h^2 f_2 \\
    u_2 - 2u_3 + u_4 &= h^2 f_3 \\
    u_4 &= 0
\end{align*}
\]

Problem: simultaneous solution needed
Ways to solve Poisson’s equation

• **Problem**: A large number of finite-difference equations must be solved simultaneously

• **Method 1. Direct**
  – Put finite-difference equations into a matrix and call a subroutine to find the solution
  – **Pro**: get the answer in one step
  – **Cons**: for large problems
    • matrix very large \((nx*ny)^2\)
    • solution very slow: \(time \approx (nx*ny)^3\)

• **Method 2. Iterative**
  – Start with initial guess and keep improving it until it is “good enough”
  – **Pros**: for large problems
    • Minimal memory needed.
    • Fast if use multigrid method: \(time \approx (nx*ny)\)
  – **Cons**: Slow if don’t use multigrid method
Iterative (Relaxation) Methods

• An alternative to using a direct matrix solver for sets of coupled PDEs
• Start with ‘guess’, then iteratively improve it
• Approximate solution ‘relaxes’ to the correct numerical solution
• Stop iterating when the error (‘residue’) is small enough
Why?

• Storage:
  – Matrix method has large storage requirements: \((\#\text{points})^2\). For large problems, e.g., 1e6 grid points, this is impossible!
  – Iterative method just uses \#points

• Time:
  – Matrix method takes a long time for large \#points: scaling as \(N^3\) operations
  – The iterative \textbf{multigrid} method has \#operations scaling as \(N\)
Example: 1D Poisson

Poisson: \( \nabla^2 u = f \)

In 1-D: \( \frac{\partial^2 u}{\partial x^2} = f \)

Finite-difference form: \( \frac{1}{h^2} (u_{i-1} - 2u_i + u_{i+1}) = f_i \)

Assume we have an approximate solution \( \tilde{u}_i \)

The error or residue: \( R_i = \frac{1}{h^2} (\tilde{u}_{i-1} - 2\tilde{u}_i + \tilde{u}_{i+1}) - f_i \)

Now calculate correction to \( \tilde{u}_i \) to reduce residue
Correcting $\tilde{u}_i$

From the residue equation note that:

$$\frac{\partial R_i}{\partial \tilde{u}_i} = \frac{-2}{h^2}$$

So adding a correction $+\frac{1}{2}h^2R_i$ to $\tilde{u}_i$ should zero $R$

i.e.,

$$\tilde{u}_i^{n+1} = \tilde{u}_i^n + \alpha \frac{1}{2} h^2 R_i$$

Unfortunately it doesn’t zero $R$ because the surrounding points also change, but it does reduce $R$

$\alpha$ is a ‘relaxation parameter’ of around 1:

$\alpha > 1 \implies$ ‘overrelaxation’

$\alpha < 1 \implies$ ‘underrelaxation’
2 types of iterations: Jacobi & Gauss-Seidel

• Gauss-Seidel: update point at same time as residue calculation
  – do (all points)
  – residue=...
  – $u(i,j) = u(i,j) + f(residue)$
  – end do

• Jacobi: calculate all residues first then update all points
  – do (all points)
  – residue(i,j)=...
  – end do
  – do (all points)
  – $u(i,j) = u(i,j) + f(residue(i,j))$
  – end do
Use Gauss-Seidel or Jacobi?

- Gauss-Seidel converges faster and does not require storage of all points’ residue.
- $\alpha > 1$ (over-relaxation) can be used for GS, but $< 1$ required for J to be stable.
- For our multigrid program, optimal $\alpha$ about 1 for GS, 0.7 for Jacobi.
- Only problem: GS not possible on multiple CPUs. Solution: red-black iterations.
- Conclusion: use Gauss-Seidel iterations.
Demonstration of 1-D relaxation using Matlab:
Things to note

• The residue becomes smooth as well as smaller
  (=>short-wavelength solution converges fastest)

• #iterations increases with #grid points. How
  small must R be for the solution to be ‘good
  enough’ (visually)?

• Effect of $\alpha$:
  – smaller => slower convergence, smooth residue
  – larger => faster convergence
  – too large => unstable
Higher N => slower convergence
Now 2D Poisson’s eqn.

\[ \nabla^2 u = f \]

Finite-difference approximation:

\[ \frac{1}{h^2} \left( u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j} - 4u_{i,j} \right) = f_{ij} \]

Use iterative approach=>start with \( u=0 \), sweep through grid updating \( u \) values according to:

\[ \tilde{u}_{ij}^{n+1} = \tilde{u}_{ij}^{n} + \alpha R_{ij} \frac{h^2}{4} \]

Where \( R_{ij} \) is the residue ("error"):

\[ R = \nabla^2 \tilde{u} - f \]
Residue after repeated iterations

Start  
rms residue=0.5

5 iterations  
Rms residue=0.06

20 iterations  
Rms residue=0.025

Residue gets **smoother** => iterations are like a diffusion process
Iterations smooth the residue
=> solve $R$ on a coarser grid
=> faster convergence

Start
rms residue=0.5

5 iterations
Rms residue=0.06

20 iterations
Rms residue=0.025
2-grid Cycle

• Several iterations on the fine grid
• Approximate ("restrict") R on coarse grid
• Find coarse-grid solution to R (=correction to u)
• Interpolate ("prolongate") correction=>fine grid and add to u
• Repeat until low enough R is obtained
We will use the grid on the left. Number of points has to be a power-of-two plus 1, e.g., 5, 9, 17, 33, 65, 129, 257, 513, 1025, 2049, 4097, ...

Figure 5.1.2 Vertex-centred and cell-centred coarsening in two dimensions. (● grid points.)
Multigrid cycle

• Start as 2-grid cycle, but keep going to coarser and coarser grids, on each one calculating the correction to the residue on the previous level
• **Exact solution** on coarsest grid (~ few points in each direction)
• Go from coarsest to finest level, at each step interpolating the correction from the next coarsest level and taking a few iterations to smooth this correction
• **All lengthscales are relaxed @ the same rate!**
Figure 19.6.1. Structure of multigrid cycles. S denotes smoothing, while E denotes exact solution on the coarsest grid. Each descending line \( \downarrow \) denotes restriction (\( R \)) and each ascending line \( \uparrow \) denotes prolongation (\( P \)). The finest grid is at the top level of each diagram. For the V-cycles (\( \gamma = 1 \)) the E step is replaced by one 2-grid iteration each time the number of grid levels is increased by one. For the W-cycles (\( \gamma = 2 \)), each E step gets replaced by two 2-grid iterations.
• Convergence rate independent of grid size
• =>#operations scales as #grid points
Programming multigrid V-cycles

- You are **given a function** that does the steps in the V-cycle
- The function is **recursive**, i.e., it **calls itself**. It calls itself to find the correction at the next coarser level.
- It calls **various functions** that you need to **write**: doing an iteration, calculating the residue, restrict or prolongate a field
- Add these functions and make it into a **module**
- Boundary conditions: zero
recursive function Vcycle_2D_Poisson(u_f, rhs, h) result (resV)
implicit none
real resV
real, intent(inout):: u_f(:, :) ! arguments
real, intent(in) :: rhs(:, :), h
integer :: nx, ny, nxc, nyc, i, j ! local variables
real, allocatable:: res_c(:, :, :), corr_c(:, :, :), res_f(:, :, :), corr_f(:, :, :)
real :: alpha = 1.0, res_rms

nx = size(u_f, 1); ny = size(u_f, 2) ! must be power of 2 plus 1
nxc = 1 + (nx - 1) / 2; nyc = 1 + (ny - 1) / 2 ! coarse grid size

if (min(nx, ny) > 5) then ! not the coarsest level
    !---------- take 2 iterations on the fine grid----------
    allocate(res_f(nx, ny), corr_f(nx, ny), &
             corr_c(nxc, nyc), res_c(nxc, nyc))

    !---------- restrict the residue to the coarse grid -------
    call residue_2D_Poisson(u_f, rhs, h, res_f)
    call restrict(res_f, res_c)

    !-------- solve for the coarse grid correction --------
    corr_c = 0.
    res_rms = Vcycle_2D_Poisson(corr_c, res_c, h*2) ! *RECURSIVE CALL*

    !----- prolongate (interpolate) the correction to the fine grid
    call prolongate(corr_c, corr_f)

    !--------- correct the fine-grid solution -----------
    u_f = u_f - corr_f

    !-------- two more smoothing iterations on the fine grid---
    res_rms = iteration_2D_Poisson(u_f, rhs, h, alpha)
    res_rms = iteration_2D_Poisson(u_f, rhs, h, alpha)

deadallocate(res_f, corr_f, res_c, corr_c)
else
solve for the coarse grid correction

\begin{verbatim}
! corr_c = 0.
res_rms = Vcycle_2DPosisson(corr_c, res_c, h*2) ! *RECURSIVE CALL*
\end{verbatim}

prolongate (interpolate) the correction to the fine grid

\begin{verbatim}
call prolongate(corr_c, corr_f)
\end{verbatim}

correct the fine-grid solution

\begin{verbatim}
! two more smoothing iterations on the fine grid---
res_rms = iteration_2DPosisson(u_f, rhs, h, alpha)
res_rms = iteration_2DPosisson(u_f, rhs, h, alpha)
\end{verbatim}

deallocate(res_f, corr_f, res_c, corr_c)

else

\begin{verbatim}
! coarest level (ny=5): iterate to get 'exact' solution

do i = 1, 1100
    res_rms = iteration_2DPosisson(u_f, rhs, h, alpha)
end do
\end{verbatim}

deallocate(res_f, corr_f, res_c, corr_c)

end if

\begin{verbatim}
resV = res_rms ! returns the rms. residue
\end{verbatim}

end function Vcycle_2DPosisson
The use of **result** in functions

- Avoids use of the function name in the code. Instead, another variable name is used to set the result
- Required by some compilers for recursive functions
- Example see this code
recursive function Vcycle_2DPoisson(u_f,rhs,h) result (resV)
  implicit none
  real resV
  real, intent(inout) :: u_f(:, :) ! arguments
  real, intent(in) :: rhs(:, :) , h
  integer :: nx, ny, nxc, nyc, i, j ! local variables
  real, allocatable :: res_c(:, :), corr_c(:, :), res_f(:, :), corr_f(:, :)
  real :: alpha=1.0, res_rms

  nx=size(u_f,1); ny=size(u_f,2) ! must be power of 2 plus 1
  nxc=1+(nx-1)/2; nyc=1+(ny-1)/2 ! coarse grid size

  if (min(nx,ny)>5) then  ! not the coarsest level
    allocate(res_f(nx,ny),corr_f(nx,ny), &
      corr_c(nxc,nyc),res_c(nxc,nyc))

    !------ coarsest level (ny=5): iterate to get 'exact' solution
    do i = 1,100
      res_rms = iteration_2DPoisson(u_f,rhs,h,alpha)
    end do

    resV = res_rms ! returns the rms. residue
  end if

end function Vcycle_2DPoisson
Exercise

• Make a Poisson solver module by adding necessary functions to the provided V-cycle function

• Write a main program that tests this, taking multiple iterations until the residue is less than about 1e-5 of the right-hand side f
  – NOTE: may need 64-bit precision to obtain this accuracy

• The program should have the option of calling the iteration function directly, or the V-cycle function, so that you can compare their performance
Functions to add

- function `iteration_2DPoisson(u,f,h,alpha)`
  - does one iteration on u field as detailed earlier
  - is a function that returns the rms. residue

- subroutine `residue_2DPoisson(u,f,h,res)`
  - calculates the residue in array res

- subroutine `restrict(fine,coarse)`
  - Copies every other point in fine into coarse

- subroutine `prolongate(coarse,fine)`
  - Copies coarse into every other point in fine
  - Does linear interpolation to fill the other points
Convergence criterion

Equation to solve: \[ \nabla^2 u = f \]

Residue: \[ R = \nabla^2 \tilde{u} - f \]

Iterate until \[ \frac{R_{rms}}{f_{rms}} < \text{err} \] \quad \text{Where err is e.g. } 10^{-5}

Rms=root-mean-square: \[ f_{rms} = \sqrt{\frac{\sum(A_{ij})^2}{n_x n_y}} \]
Test program details

• Using namelist input, read in nx, ny, flags to indicate what type of source and iterations, and alpha

• Initialise:
  – $h=1/(ny-1)$
  – source field $f$ to random, spike, etc.
  – $u(:,:,)=0$

• Repeatedly call either iteration_2DPoisson or Vcycle_2DPoisson until “converged” (according to rms. residue compared to rms. source field $f$)

• Write $f$ and $u$ to files for visualisation!
Tests

• Record number of iterations needed for different grid sizes for each scheme, from 17x17 to at least 257x257

• What is the effect of alpha on iterations? (for multigrid it is hard-wired)

• For multigrid, what is the maximum number of grid points that fit in your computer, and how long does it take to solve the equations?
Example solutions

- For a delta function (spike) in the center of the grid, i.e.,
  \[ f(nx/2+1,ny/2+1)=1/dy^{**2}, \text{ otherwise } 0 \]
- \((1/dy^{**2} \text{ is so that the integral of this}=1)\)
Hand in by email

- Your complete program (.f90/5 files) and parameter file necessary to run it
- Results of your tests
  - #iterations vs. grid size
  - effect of alpha
  - maximum #points
- Plots of your solutions for a delta-function source