A Prototype Finite Difference Model

Timothy H. Kaiser, Ph.D.
tkaiser2@nrel.gov

Slides at:

https://github.com/timkphd/slides

Examples at

To get examples:

```
mkdir examples
cd examples
git clone https://github.com/timkphd/examples.g
cd examples/mpi/mpi4py
```

A Prototype Model

- We will introduce a finite difference model that will serve to demonstrate what a computational scientist needs to do to take advantage of Distributed Memory computers using MPI
- The model we are using is a two dimensional solution to a model problem for Ocean Circulation, the Stommel Model

The Stommel Problem

- Wind-driven circulation in a homogeneous rectangular ocean under the influence of surface winds, linearized bottom friction, flat bottom and Coriolis force.

- Solution: intense crowding of streamlines towards the western boundary caused by the variation of the Coriolis parameter with latitude

Governing Equations Model Constants

$$\psi = 0$$

$$\psi = 0 \qquad \psi = 0$$

$$\gamma\left(\frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2}\right) + \beta\frac{\partial\psi}{\partial x} = f$$

$$f = -\alpha \sin\left(\frac{\pi y}{2L_y}\right)$$

$$L_x = L_y = 2000Km$$

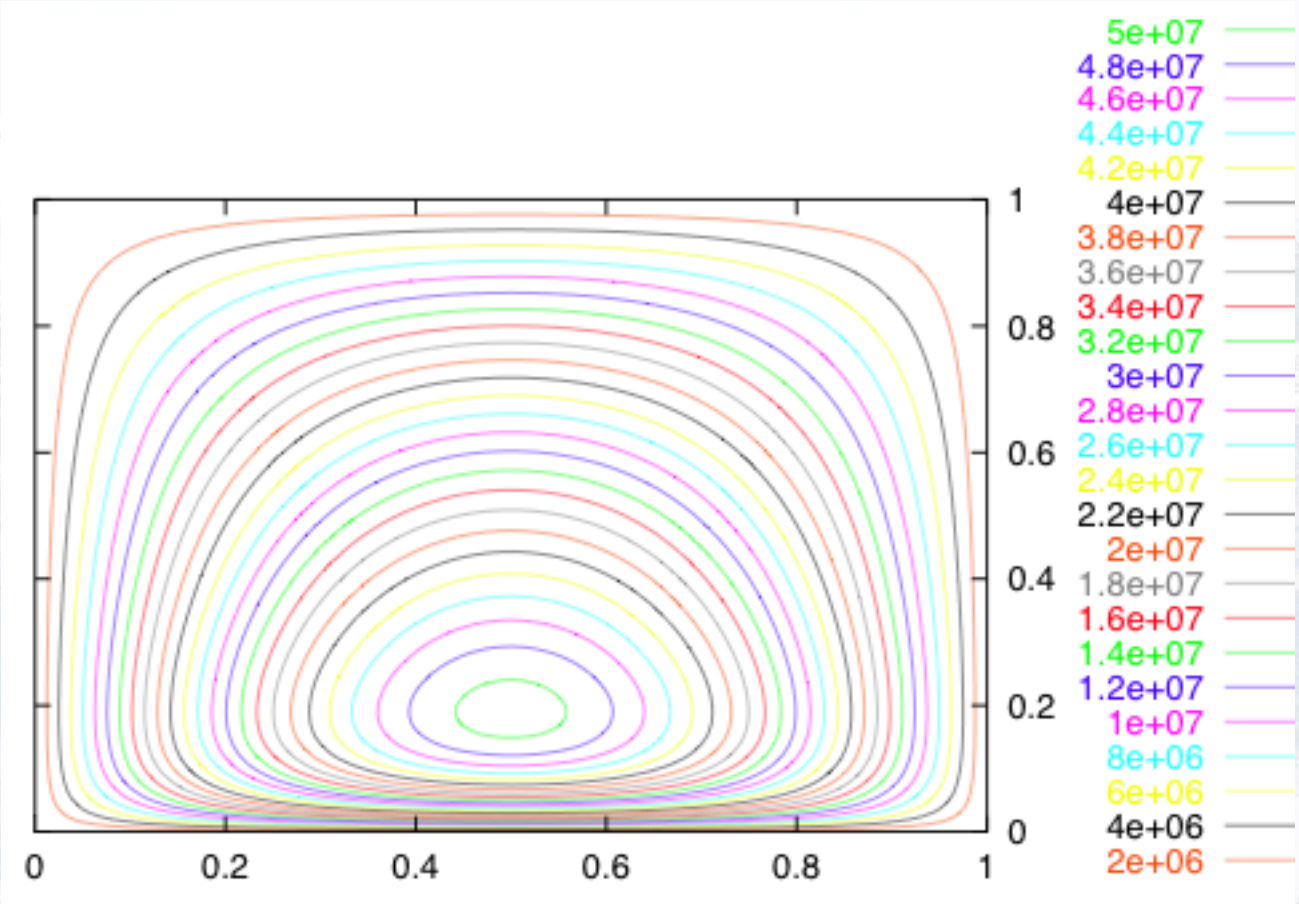
$$\gamma = 3 * 10^{(-6)}$$

$$\beta = 2.25 * 10^{(-11)}$$

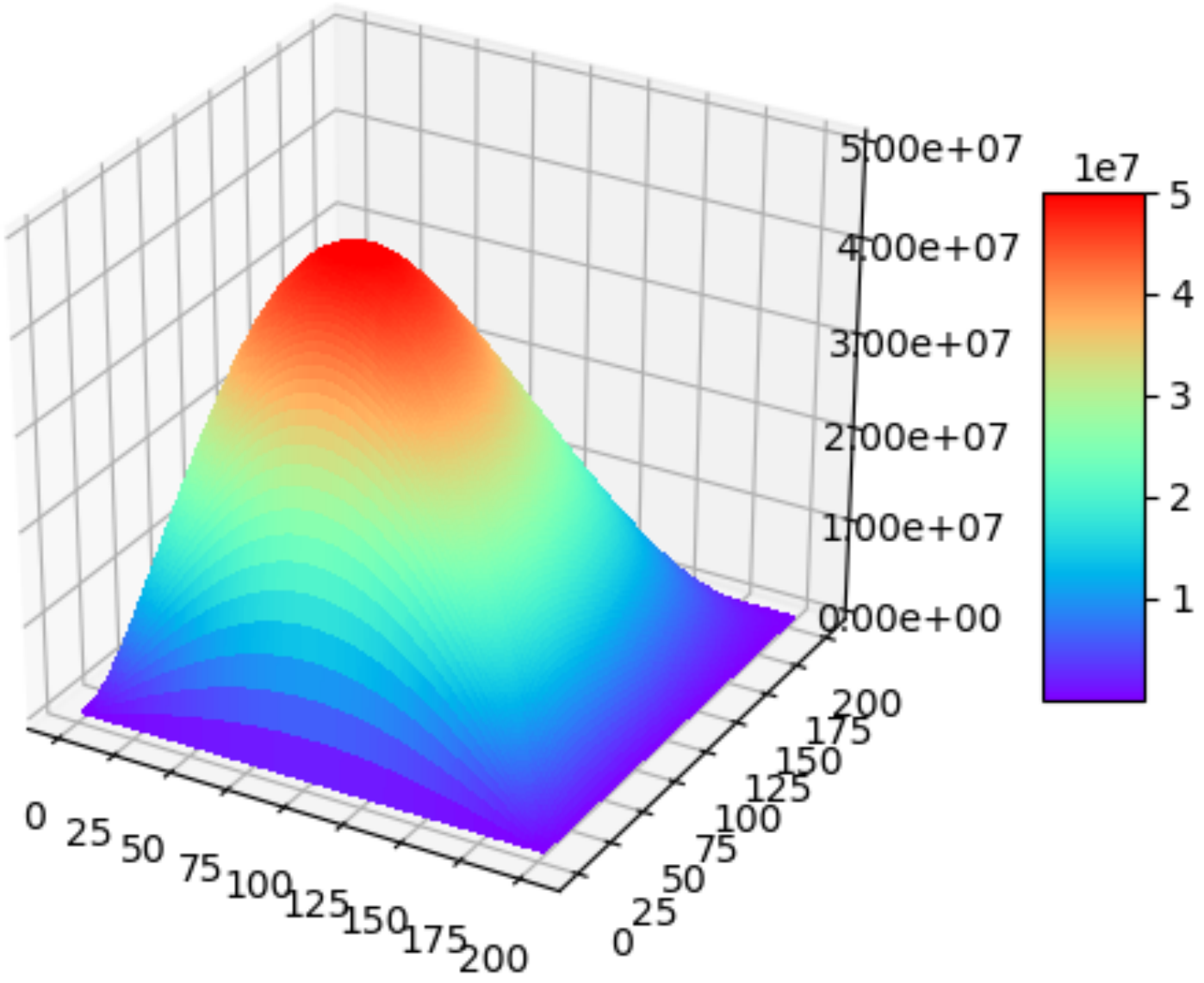
$$\alpha = 10^{(-9)}$$

$$\psi = 0$$

The steady state solution



The steady state solution



Domain Discretization

Define a grid consisting of points (x_i, y_j) given by

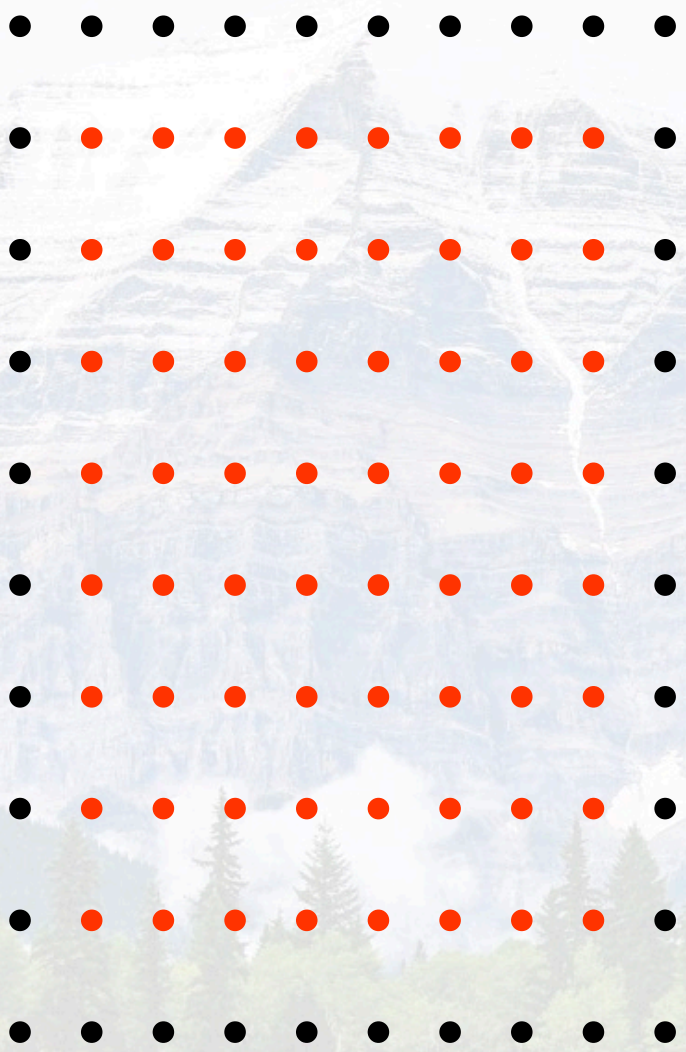
$$x_i = i\Delta x, i = 0, 1, \ldots\ldots, nx+1$$

$$y_j = j\Delta y, j = 0, 1, \ldots\ldots, ny+1$$

$$\Delta x = L_x / (nx + 1)$$

$$\Delta y = L_y / (ny + 1)$$

Domain Discretization



Seek to find an approximate solution

$\psi(x_i, y_j)$ at points (x_i, y_j):

$$\psi_{i,j} \approx \psi(x_i, y_j)$$

Centered Finite Difference Scheme for the Derivative Operators

$$\frac{\partial \psi}{\partial x} \approx \frac{\psi_{i+1,j} - \psi_{i-1,j}}{2\Delta x}$$

$$\frac{\partial^2 \psi}{\partial x^2} \approx \frac{\psi_{i+1,j} - 2\psi_{i,j} + \psi_{i-1,j}}{(\Delta x)^2}$$

$$\frac{\partial^2 \psi}{\partial y^2} \approx \frac{\psi_{i,j+1} - 2\psi_{i,j} + \psi_{i,j-1}}{(\Delta y)^2}$$

Governing Equation
Finite Difference Form

$$\psi_{i,j} = a_1\psi_{i+1,j} + a_2\psi_{i-1,j} + a_3\psi_{i,j+1} + a_4\psi_{i,j-1} - a_5 f_{i,j}$$

$$a_1 = \frac{\Delta y^2}{2(\Delta x^2 + \Delta y^2)} + \frac{\beta \Delta x^2 \Delta y^2}{4\gamma \Delta x(\Delta x^2 + \Delta y^2)}$$

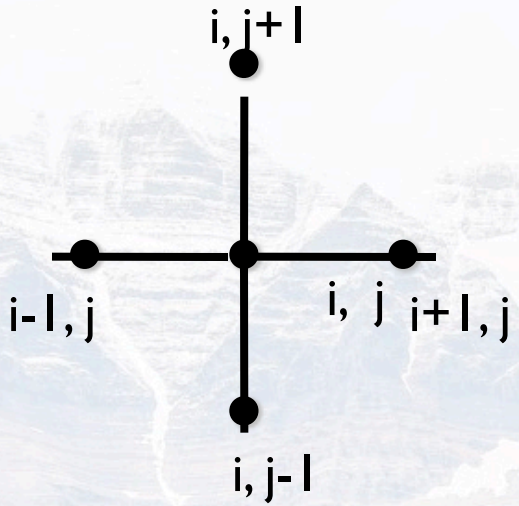
$$a_2 = \frac{\Delta y^2}{2(\Delta x^2 + \Delta y^2)} - \frac{\beta \Delta x^2 \Delta y^2}{4\gamma \Delta x(\Delta x^2 + \Delta y^2)}$$

$$a_3 = \frac{\Delta x^2}{2(\Delta x^2 + \Delta y^2)}$$

$$a_4 = \frac{\Delta x^2}{2(\Delta x^2 + \Delta y^2)}$$

$$a_5 = \frac{\Delta x^2 \Delta y^2}{2\gamma(\Delta x^2 + \Delta y^2)}$$

Five-point Stencil Approximation



interior grid points: i=1,nx;
j=1,ny

boundary points:

(i,0) & (i,ny+1) ; i=0,nx+1

(0,j) & (nx+1,j) ; j=0,ny+1

$$\psi_{i,j} = a_1\psi_{i+1,j} + a_2\psi_{i-1,j} + a_3\psi_{i,j+1} + a_4\psi_{i,j-1} - a_5 f_{i,j}$$

$$\psi_{i,0} = \psi_{i,ny+1} = 0; \quad \psi_{0,j} = \psi_{nx+1,j} = 0;$$

Jacobi Iteration

Start with an initial guess for $(\psi_{i,j})$

Repeat the process

do i = 1, nx; j = 1, ny

$$(\psi_{i,j})\_new = a_1(\psi_{i+1,j}) + a_2(\psi_{i-1,j}) + a_3(\psi_{i,j+1}) + a_4(\psi_{i,j-1}) - a_5 f_{i,j}$$

end do

$$(\psi_{i,j}) = (\psi_{i,j})\_new$$

Our Examples

We have a finite difference model that will serve to demonstrate what a computational scientist needs to do to take advantage of Distributed Memory computers using MPI.

The model we are using is a two dimensional solution to a model problem for Ocean Circulation, the Stommel Model. It has Wind-driven circulation in a homogeneous rectangular ocean under the influence of surface winds, linearized bottom friction, flat bottom and Coriolis force.

Solution: intense crowding of streamlines towards the western boundary caused by the variation of the Coriolis parameter with latitude.
For a description of the Fortran and C versions of this program see:
http://geco.mines.edu/prototype/Show_me_some_local_HPC_tutorials/stoma.pdf
http://geco.mines.edu/prototype/Show_me_some_local_HPC_tutorials/stomb.pdf

The python version, stp.py, follows this C version except it does a 1d decomposition. The C version is 1500x faster than the python version.

pcalc.py and ccalc.c are similar except they create a new communicator that contains N-1 tasks. These tasks do the calculation and pass data to the remaining task to be plotted. Thus we can have "C" do the heavy calculation and python do plotting.

File	Comment
ccalc.c	parallel
stc_03.c	parallel
pcalc.py	parallel
stp_00.py	serial
stp.py	parallel
tiny.in	tiny input file
small.in	small input file
st.in	regular input file

Python Version

```
#!/usr/bin/env python
from math import pi,sin
from math import fabs as abs
#from Numeric import empty
from numpy import empty
import numpy
from time import time as walltime
global vals,cons
global psi,new_psi,forf
import sys
global a1,a2,a3,a4,a5,a6,dx,dy
global r1,r2
global ttol
```

Python Version

```
class constants:
    a1,a2,a3,a4,a5,a6,dx,dy=(0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0)
    def __init__(this,inputs):
        dx=inputs.lx/(inputs.nx+1.0)
        dy=inputs.ly/(inputs.ny+1.0)
        dx2=dx*dx
        dy2=dy*dy
        bottom=2.0*(dx2+dy2)
        a1=(dy2/bottom)+(inputs.beta*dx2*dy2)/(2.0*inputs.gamma*dx*bottom)
        a2=(dy2/bottom)-(inputs.beta*dx2*dy2)/(2.0*inputs.gamma*dx*bottom)
        a3=dx2/bottom
        a4=dx2/bottom
        a5=dx2*dy2/(inputs.gamma*bottom)
        a6=pi/(inputs.ly)
        this.dx=dx
        this.dy=dy
        this.a1=a1
        this.a2=a2
        this.a3=a3
        this.a4=a4
        this.a5=a5
        this.a6=a6
    def getCons(this):
        return(this.a1,this.a2,this.a3,this.a4,this.a5,this.a6,this.dy,this.dx)
```

Python Version

```
class input:
   nx,ny=(50 , 50)
   lx,ly=(2000000 , 2000000)
   alpha,beta,gamma=(1.0e-9 , 2.25e-11 , 3.0e-6)
   steps=5000
   def __init__(this):
      pass
   def getInput(this):

return(this.nx,this.ny,this.lx,this.ly,this.alpha,this.bet
a,this.gamma,this.steps)
```

Python Version

```
def do_force(forf,i1,i2,j1,j2):
   def force(y):
      global cons,vals
      return(-vals.alpha*sin(y*cons.a6))
   for i in range(i1-1,i2+2):
      for j in range(j1-1,j2+2):
         y=j*cons.dy
         forf[i,j]=force(y)
```

```
def bc(psi,i1,i2,j1,j2):
   psi[i1-1,:]=0.0
   psi[i2+1,:]=0.0
   psi[:,j1-1]=0.0
   psi[:,j2+1]=0.0
```

Python Version

```
def do_jacobi(psi,new_psi,i1,i2,j1,j2):
# does a single Jacobi iteration step
# input is the grid and the indices for the interior cells
# new_psi is temp storage for the the updated grid
# output is the updated grid in psi and diff which is
# the sum of the differences between the old and new grids
   global cons
   global forf
   global ttot
   global a1,a2,a3,a4,a5,a6,dx,dy
   diff=0.0
   ts=walltime()
   for j in r2:
      for i in r1:
         new_psi[i,j]=a1*psi[i+1,j] + a2*psi[i-1,j] + \
         a3*psi[i,j+1] + a4*psi[i,j-1] - \
         a5*forf[i,j]
         diff=diff+abs(new_psi[i,j]-psi[i,j])
   psi[:,:]=new_psi[:,:]
   te=walltime()
   ttot=ttot+(te-ts)
   return (diff)
```

Python Version

```
#get the input.  see above for typical values
vals=input()

#set the indices for the interior of the grid
i1=1
i2=vals.nx
j1=1
j2=vals.ny
# allocate the grid to size nx * ny plus the boundary cells
t1=walltime()
psi=empty(((i2-i1)+3,(j2-j1)+3),"d")
new_psi=empty(((i2-i1)+3,(j2-j1)+3),"d")
forf=empty(((i2-i1)+3,(j2-j1)+3),"d")
#calculate the constants for the calculations
cons=constants(vals)
(a1,a2,a3,a4,a5,a6,dx,dy)=cons.getCons()

# set initial guess for the value of the grid
psi[:,:]=1.0
do_force(forf,i1,i2,j1,j2)
#set boundary conditions
bc(psi,i1,i2,j1,j2)
```

Python Version

```
new_psi[:,:]=psi[:,:]
iout=vals.steps//100
if(iout  == 0):
   iout=1
r1=range(i1,i2+1)
r2=range(j1,j2+1)
ttot=0
for i in range(0,vals.steps):
   diff=do_jacobi(psi,new_psi,i1,i2,j1,j2)
   if ((i+1) % iout) == 0 :
      print(i+1,diff)
t2=walltime()
write_it(psi,i1,i2,j1,j2,i2,j2)
print("total time=",t2-t1, "  time spent in do_jacobi=",ttot)
```

Python Version

```
def write_it(psi,i1, i2, j1, j2,nx,ny):
   from numpy import empty
   myid=100
   if(i1==1):
      i0=0
   else :
      i0=i1
   if(i2==nx):
      i3=nx+1
   else :
      i3=i2
   if(j1==1):
      j0=0
   else :
      j0=j1
   if(j2==ny):
      j3=ny+1
   else :
      j3=j2
                  fname="out"+str(myid)
                  eighteen=open(fname,"w")
                  aline=(str(i3-i0+1)+" , "+str(j3-j0+1)+"
               "+str(psi.shape))
                  eighteen.write(aline+"\n")
                  for i in range(i0,i3+1) :
                     for j in range(j0,j3+1) :
                        xout=str(psi[i][j])
                        eighteen.write(xout)
                        if(j != j3):
                           eighteen.write(" ")
                     eighteen.write("\n")
                  eighteen.close()
```

A Prototype Finite Difference Model (Philosophy)

Overview

- Model written in Fortran 90
- Uses many new features of F90
  - Free format
  - Modules instead of commons
  - Module with kind precision facility
  - Interfaces
  - Allocatable arrays
  - Array syntax

http://inside.mines.edu/~tkaiser/fortran//

http://inside.mines.edu/~tkaiser/fortran/new/

Free Format

- Statements can begin in any column
- ! Starts a comment
- To continue a line use a “&” on the line to be continued

Modules instead of commons

- Modules have a name and can be used in place of named commons
- Modules are defined outside of other subroutines
- To "include" the variables from a module in a routine you "use" it
- The main routine stommel and subroutine jacobi share the variables in module "constants"

```
module constants
    real dx,dy,a1,a2,a3,a4,a5,a6
end module
…
program stommel
    use constants
…
end program
```

```
subroutine jacobi
    use constants
…
end subroutine jacobi
```

Kind precision facility

Instead of declaring variables

```
real*8 x,y
```

We use

```
real(b8) x,y
```

Where b8 is a constant defined within a module

```
module numz
 integer,parameter::b8=selected_real_kind(14)
end module
program stommel
    use numz
    real(b8) x,y
    x=1.0_b8
...
```

Kind precision facility Why?

Legality
Portability
Reproducibility
Modifiability

is not legal syntax in Fortran 90

Declaring variables "double precision" will give us 16 byte reals on some machines

```
integer,parameter::b8=selected_real_kind(14)
real(b8) x,y
x=1.0_b8
```

Allocatable arrays

- We can declare arrays to be allocatable
- Allows dynamic memory allocation
- Define the size of arrays at run time

```
real(b8),allocatable::psi(:,:)     ! our calculation grid
real(b8),allocatable::new_psi(:,:) ! temp storage for the grid
```

```
! allocate the grid to size nx * ny plus the boundary cells
allocate(psi(0:nx+1,0:ny+1))
allocate(new_psi(0:nx+1,0:ny+1))
```

Interfaces

- Similar to C prototypes
- Can be part of the routines or put in a module
- Provides information to the compiler for optimization
- Allows type checking

```
module face
  interface bc
    subroutine bc (psi,i1,i2,j1,j2)
      use numz
      real(b8),dimension(i1:i2,j1:j2):: psi
      integer,intent(in):: i1,i2,j1,j2
    end subroutine
  end interface
end module
program stommel
    use face
...
```

Array Syntax

Allows assignments of arrays without do loops

```
! allocate the grid to size nx * ny plus the boundary cells
    allocate(psi(0:nx+1,0:ny+1))
    allocate(new_psi(0:nx+1,0:ny+1))


! set initial guess for the value of the grid
    psi=1.0_b8


! copy from temp to main grid
    psi(i1:i2,j1:j2)=new_psi(i1:i2,j1:j2)
```

Program Outline (1)

- Module NUMZ - defines the basic real type as 8 bytes
- Module INPUT - contains the inputs
  - nx,ny (Number of cells in the grid)
  - lx,ly (Physical size of the grid)
  - alpha,beta,gamma (Input calculation constants)
  - steps (Number of Jacobi iterations)
- Module Constants - contains the invariants of the calculation

Program Outline (2)

- Module face - contains the interfaces for the subroutines
  - bc - boundary conditions
  - do_jacobi - Jacobi iterations
  - force - right hand side of the differential equation
  - Write_grid - writes the grid

Program Outline (3)

- Main Program
  - Get the input
  - Allocate the grid to size nx * ny plus the boundary cells
  - Calculate the constants for the calculations
  - Set initial guess for the value of the grid
  - Set boundary conditions using
  - Do the jacobi iterations
  - Write out the final grid

C version considerations

- To simulate the F90 numerical precision facility we:
  - #define FLT double
  - And use FLT as our real data type throughout the rest of the program
- We desire flexibility in defining our arrays and matrices
  - Arbitrary starting indices
  - Contiguous blocks of memory for 2d arrays
  - Use routines based on *Numerical Recipes in C*

Vector allocation routine

```
FLT *vector(INT nl, INT nh)
{
/* creates a vector with bounds vector[nl:nh]] */
        FLT *v;
/* allocate the space */
        v=(FLT *)malloc((unsigned) (nh-nl+1)*sizeof(FLT));
        if (!v) {
            printf("allocation failure in vector()\n");
                exit(1);
    }
/* return a value offset by nl */
        return v-nl;
}
```

Matrix allocation routine

```
FLT **matrix(INT nrl,INT nrh,INT ncl,INT nch)
/* creates a matrix with bounds matrix[nrl:nrh][ncl:nch] */
/* modified from the book version to return contiguous space */
{
 INT i;
 FLT **m;
/* allocate an array of pointers */
 m=(FLT **) malloc((unsigned) (nrh-nrl+1)*sizeof(FLT*));
 if (!m){
  printf("allocation failure 1 in matrix()\n"); exit(1);}
 m -= nrl; /* offset the array of pointers by nrl */
 for(i=nrl;i<=nrh;i++) {
  if(i == nrl){
/* allocate a contiguous block of memroy*/
   m[i]=(FLT *) malloc((unsigned) (nrh-nrl+1)*(nch-ncl+1)*sizeof(FLT));
   if (!m[i]){
     printf("allocation failure 2 in matrix()\n");exit(1); }
   m[i] -= ncl; /* first pointer points to beginning of the block */
  }
  else {
    m[i]=m[i-1]+(nch-ncl+1); /* rest of pointers are offset by stride */
  }
 }
 return m;
}
```

Digression... a 3d Volume allocation routine

Same idea but we allocate an array of slices

```
FLT ***cube(INT nslice1,INT nslice2,INT nrow1,INT nrow2,INT ncol1,INT ncol2) {
FLT **slice(INT nrow1,INT nrow2,INT ncol1,INT ncol2,FLT **temp);
     FLT *temp,***mcube;
     INT i;
     mcube=(FLT ***) malloc((unsigned) (nslice2-nslice1+1)*sizeof(FLT*));
     if (!mcube){
          printf("allocation failure at 1 in cube()\n");
          return NULL;
     }
     mcube -= nslice1;
     temp=(FLT*) malloc((unsigned) (nslice2-nslice1+1)*
                                   (nrow2-nrow1+1)*
                                   (ncol2-ncol1+1)*sizeof(FLT));
     if (*temp){
          printf("allocation failure at 2 in cube()\n");
          return NULL;
     }
     for(i=nslice1;i<=nslice2;i++) {
          mcube[i]=slice(nrow1, nrow2, ncol1, ncol2,&temp);
          if(!mcube[i]) return NULL;
          temp += (nrow2-nrow1+1)*(ncol2-ncol1+1);
     }
     return mcube;
}
```

Digression... a 3d version of this routine

Our slice allocation routine

```
FLT **slice(INT nrow1,INT nrow2,INT ncol1,INT ncol2,FLT **temp) {
    INT i;
     FLT **mslice;
     mslice=(FLT **) malloc((unsigned) (nrow2-nrow1+1)*sizeof(FLT*));
     if (!mslice){
          printf("allocation falure at 3 in slice()\n");
          return NULL;
     }
     mslice -= nrow1;
     for(i=nrow1;i<=nrow2;i++) {
         if(i == nrow1){
              mslice[i]=*temp;
              mslice[i] -= ncol1;
         }
         else {
             mslice[i]=mslice[i-1]+(ncol2-ncol1+1);
         }
     }
     return mslice;
}
```