3-2

i. **Basics: accessing SAC functionality and data from external programs**
ii. Automating SAC processing with shell scripts
iii. Accessing SAC data from Fortran with the sacio library
iv. Accessing SAC data from MATLAB

SAC from external programs

- Even with macros, SAC inevitably will not be able (nor is designed) to implement every possible function or methodology one might wish to apply to seismic data

- SACs macro language has limits, and many things are not possible which are trivial in scripts or compiled languages

- External access to SAC is in two main ways; (a) running SAC itself from some sort of control environment, and (b) loading, modifying and writing SAC data in external programs unconnected to SAC

- We will look at each of these in turn.

3-2

i. Basics: accessing SAC functionality and data from external programs

ii. **Automating SAC processing with shell scripts**

iii. Accessing SAC data from Fortran with the sacio library

iv. Accessing SAC data from MATLAB

Calling SAC from shell scripts

- Shell scripts (similarly, batch files in Windows systems) are sets of commands interpreted by the operating system
- They can run both system commands and executable codes (like SAC)
- Shell scripts also enable basic programming constructs like conditionals and loops
- They can be used to provide a higher level of SAC automation than is possible from within SAC using macros

SAC default macro

- SAC automation is made possible by the ability for the user to specify a macro which SAC will execute when it starts:

```
$ sac mymacro
```

starts SAC, which then runs the command(s) in the macro mymacro

- A shell script can construct these macros on the fly, so each time SAC is executed (say in a loop) a new macro can be used

Example SAC automation script

```
#!/bin/csh
# Script to demonstrate automated running of SAC.
#
unalias sac # prevent any aliases colliding

# set loop length
set n = 5

# run loop
@ i = 1
while ($i <= $n) # start of loop

# build up a SAC macro
echo "funcgen seismogram" > /tmp/mymacro
echo "mul " $i >> /tmp/mymacro
echo "w myfile."$i >> /tmp/mymacro
echo "quit" >> /tmp/mymacro

# run SAC with the macro
sac /tmp/mymacro

@ i = $i + 1
end # end of loop

# end of script
```

SAC automation

- Automation in this fashion is a handy way of accessing SAC functionality from other programs
- Set up large SAC batch jobs; possible to use for parallel processing in SAC
- Not very efficient – requires a lot of extra processing effort starting and quitting SAC. Shell scripts are much (much!) slower than compiled code, and only provided fairly limited programming functionality
- For a lot of applications this does not matter!

3-2

i. Basics: accessing SAC functionality and data from external programs
ii. Automating SAC processing with shell scripts
iii. Accessing SAC data from Fortran with the sacio library
iv. Accessing SAC data from MATLAB

Externally processing SAC data

- Other model is to process SAC data inside an external compiled program
- This enables the addition of processing not available in SAC – obviously in research seismology this is not an uncommon occurrence!
- (Probably) the most commonly used programming language in Science is Fortran. Also, a huge amount of ‘legacy’ code exists.
- A library of data access functions for Fortran (and C) are distributed with SAC
- Enables easy reading and writing of SAC files, allows access to headers

Linking FORTRAN to the sacio library

- Fortran codes using the sacio library routines must be linked to the sacio.a library, e.g.:

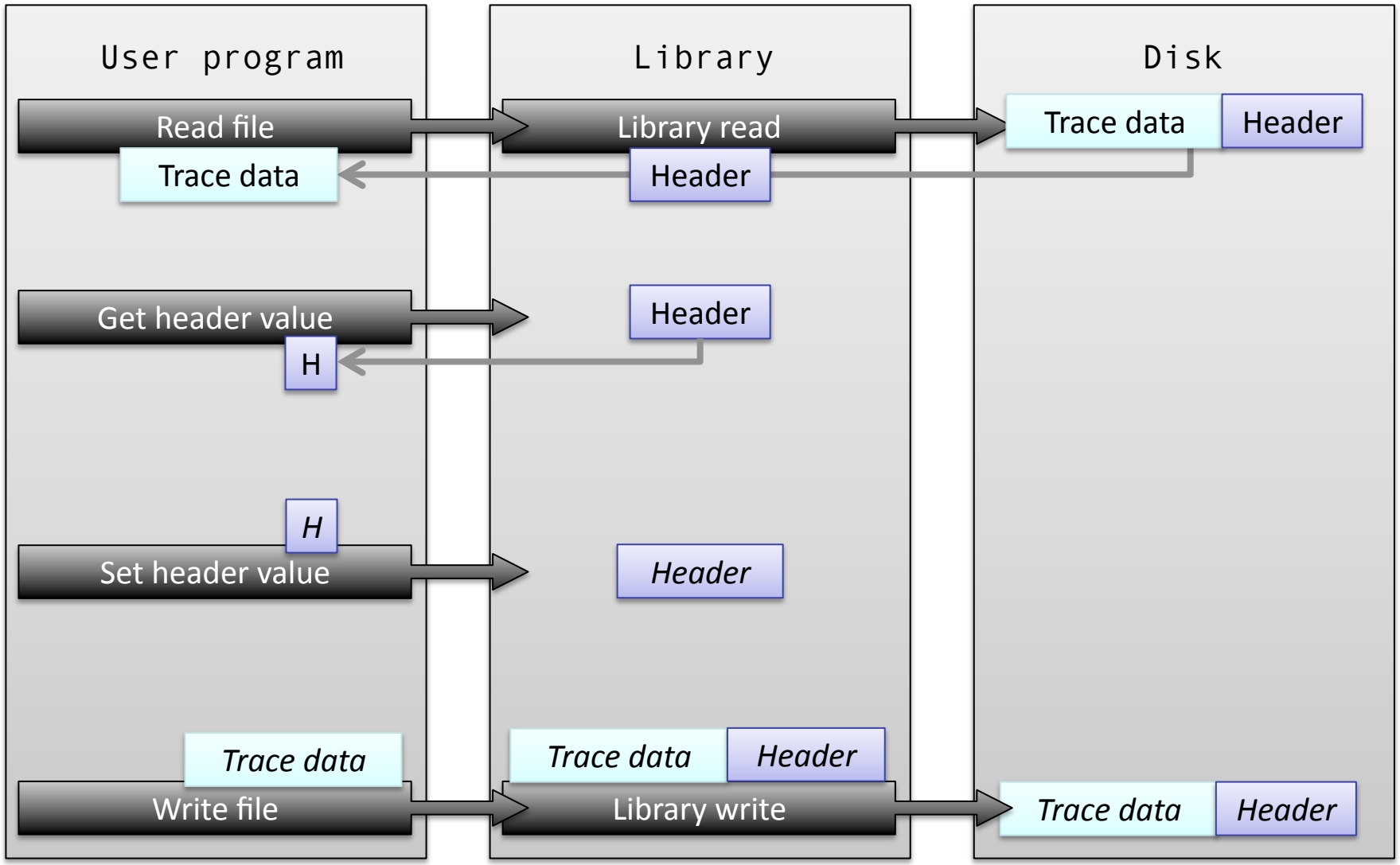
```
$ gfortran -o myprog myprog.f90 /usr/local/lib/sacio.a
```

- Problems can arise when trying to link 64-bit programs to a 32-bit library (for example, that distributed with MacSAC)

- To circumvent this, compile your program as 32-bit:

```
$ gfortran -m32 -o myprog myprog.f90 /usr/local/lib/sacio.a
```

sacio library interaction model



Reading SAC files

- Subroutine rsac1 reads SAC time-series files:

```
*=====================================================================
      subroutine rsac1(kname,yarray,nlen,beg,del,max,nerr)
*=====================================================================
* PURPOSE: To read an evenly spaced SAC file.
*=====================================================================
* INPUT ARGUMENTS:
*    KNAME:   Name of disk file to read. [c]
*             The name should be blank filled.
*    MAX:     Size of YARRAY. [i]
*=====================================================================
* OUTPUT ARGUMENTS:
*    YARRAY:  Contains the data from the file. [fa]
*    NLEN:    Number of data points read. [i]
*             Will be less than or equal to MAX.
*    BEG:     Beginning value of independent variable. [f]
*    DEL:     Sampling interval of the independent variable. [f]
*    NERR:    Error return flag 0 if no error occurred. [i]
*             Possible values for this subroutine are:
*             =  801 if file is not evenly spaced.
*             = -803 if number of points in file is greater than MAX.
*             In this case, the first MAX points are read.
```

- Only minimal header information is returned – header of the last trace read is stored in the library

Getting headers

- Subroutine getfhv fetches (floating point) header values for the last trace read by the library

```
*=====================================================================
      subroutine getfhv(kname,fvalue,nerr)
*=====================================================================
* PURPOSE: To get a floating point header value from the current SAC
file.
*=====================================================================
* INPUT ARGUMENTS:
*    KNAME:   Name of header field to get. [c]
*=====================================================================
* OUTPUT ARGUMENTS:
*    FVALUE:  Value of header field from current SAC data file. [f]
*    NERR:    Error flag. Set to 0 if no error occurred. [i]
*             = 1336 Header variable is undefined.
*             = 1337 Header variable does not exist.
*=====================================================================
```

- Equivalent routines fetch integer (getihv), character (getkhv) and logical (getlhv) headers.

Setting headers

- Subroutine setfhv sets (floating point) header values for the current header stored in the library

```
*=====================================================================
      subroutine setfhv(kname,fvalue,nerr)
*=====================================================================
* PURPOSE: To set a floating point header value in the current SAC file.
*=====================================================================
* INPUT ARGUMENTS:
*    KNAME:   Name of header field to set. [c]
*    FVALUE:  New value of header field. [f]
*=====================================================================
* OUTPUT ARGUMENTS:
*    NERR:    Error flag. Set to 0 if no error occurred. [i]
*             = 1337 Header field does not exist.
*=====================================================================
```

- Equivalent routines set integer (setihv), character (setkhv) and logical (setlhv) headers.

Writing SAC files

- Two different approaches to writing files:
  1. Output a file with minimal header information
     - Requires no set up
     - Any previous trace information is lost
     - Useful for temporary or QC files, files to which the header information no longer applies
  2. Output a file with a complete header
     - Need to either set up header, or have one from a previously read trace
     - Useful for when program is a processing step

Writing SAC files: method 1

- Command wsac1 writes out a file with a minimal header:

```
*=====================================================================
      subroutine wsac1(kname,yarray,nlen,beg,del,nerr)
*=====================================================================
* PURPOSE: To write an evenly spaced SAC file.
*=====================================================================
* INPUT ARGUMENTS:
*     KNAME:  Name of disk file to write. [c]
*             The name should be blank filled.
*    YARRAY:  Array containing the dependent variable. [fa]
*      NLEN:  Length of YARRAY. [i]
*       BEG:  Beginning value of the independent variable. [f]
*       DEL:  Sampling interval of the independent variable. [f]
*=====================================================================
* OUTPUT ARGUMENTS:
*      NERR:  Error return flag, 0 if no error occurred. [i]
*=====================================================================
```

Writing SAC files: method 2

- Command wsac0 writes out a file with *the current* header:

```
*=====================================================================
      subroutine wsac0(kname,xarray,yarray,nerr)
*=====================================================================
* PURPOSE: To write a SAC file to disk using current header values.
*=====================================================================
* INPUT ARGUMENTS:
*     KNAME:  Name of disk file to write. [c]
*             The name should be blank filled.
*    XARRAY:  Array containing independent variable. [fa]
*             This is not used if the data is evenly spaced.
*    YARRAY:  Array containing dependent variable. [fa]
*=====================================================================
* OUTPUT ARGUMENTS:
*      NERR:  Error return flag 0 if no error occurred. [i]
*=====================================================================
```

- If there is no current header (for example, if the trace is being generated by the program) then a new (empty) header must be generated with a call to newhdr, and the required variables set with set*hv calls

Example

```
!  program to double amplitudes in files recorded before 2005
   program correct_pre2005
      implicit none
      integer, parameter :: nmax = 10000
      real data(nmax),beg,dt
      integer nzyear,npts,nerr,ip

      character fn*80,stnm*8

!     read trace
      fn = 'SWAV.BHN'
      call rsac1(fn,data,npts,beg,dt,nmax,nerr)

!     if seismogram is pre-2005, double amplitudes
      call getnhv('NZYEAR',nzyear,nerr)
      if (nzyear < 2005) then
         do ip=1,npts
            data(ip) = data(ip) * 2.0
         enddo
      endif

!     flag fixed in a header
      call setkhv('KUSER0','FIXED   ',nerr)

!     write back the file, using the header currently in memory
      call wsac0(fn,data,data,nerr)
      stop
   end program correct_pre2005
```

FOTRAN90 alternative

- The F90sac library is an alternative to the standard sacio library which uses FORTRAN90 constructs to provide a more object-oriented approach to handling SAC files
- Traces are treated as structures, with header and trace being fields
- This makes multiple traces much easier to handle

```
program simplef90
use f90sac  ! use the f90sac module
implicit none
type (SACTrace) :: tr

fname = 'SWAV.BHE'
call f90sac_readtrace(fname,tr)

tr % t0 = 275. ;
tr % kt0 = 'NewPick' ;
tr % trace(:) = 0.0

call f90sac_writetrace(fname,tr)

end program simple
```

3-2

i. Basics: accessing SAC functionality and data from external programs
ii. Automating SAC processing with shell scripts
iii. Accessing SAC data from Fortran with the sacio library
iv. Accessing SAC data from MATLAB and beyond

Using SAC data in other languages

- Read and write routines also exist in other languages
- MATLAB and Python routines are included in the API subdirectory in the software distribution
- Enables the use of, e.g., graphical/processing capabilities:

```
% read traces
tr = msac_mread('*.BHE') ;
A = [tr(:).trace] ;
T = [tr(:).time] ;
for itr = 1:length(tr), D(:,itr) = tr(itr).gcarc; , end

% sort matrices by epicentral distance
[d,ind] = sort([tr(:).gcarc]) ;
AS = A(:,ind) ;
TS = T(:,ind) ;
DS = D(:,ind) ;

% plot surface
h=surf(DS,TS,AS,'LineStyle','none');
```

Using SAC data in other languages