BASIS

(the Corsica code's framework)

L. L. LoDestro

Corsica Winter School

Jan. 26—28, 2016

ackn: The authors of *The Basis System, part 1*; and Judy Harte, LLNL.
Work performed by LLNL under US DOE Contract DE-AC52-07NA27344

Preface: a bit about Basis

- What is it? -- both a code-development system and a compiled program.

- Capsule history:
  - First developed in the mid 1980's by Paul Dubois in MFE
  - Late 1980's: Paul moved to ICF, Lasnex brought under Basis; much further development through 1990's
  - As of 2012: small team, good support, modest development
  - Current status: authors are available if there are problems

Preface, cont.

- The Basis code-development system: tools to
  - provide run-time access to a code's database
  - connect independent codes under a common framework
  - relieve physics authors of many chores: i/o, graphics, history, portable data files, [dynamic dimensioning].
- The Basis executable (no user-attached "packages") includes:
  - an interactive language with interpreter and scripts, logs, …
  - mathematical functions
  - plotting package
  - history
  - saving and retrieving variables

Basis tutorial

- Before starting, set your path:

```
setenv  BASIS_ROOT  /project/caltrans/basis/vbasis

setenv  PACT  /project/caltrans/pact/pact04_05_11     # for savfiles

setenv  NCARG_ROOT  /project/caltrans/ncar       # for plot files

set path  = ($path  $BASIS_ROOT/bin  $PACT/bin  $NCARG_ROOT/bin)
```

- Start the code:

```
basis
```

You'll get the prompt:

```
Basis>
```

Basis tutorial

- Documentation:

https://wci.llnl.gov/codes/basis/documentation.html

part 1 # language tutorial, drawn upon here
2 # language reference manual
3 # graphics manual
6 # basis package library (PFB, RNG, SVD, …)

Basis> news

Basis> list (more on the list command later)

Basis language interpreter tutorial: language very similar to Fortran and Idl

- has all the Fortran operators and delimiters (and more)
- expressions look like Fortran
- has all the data types (and more)
- if ’s look just like Fortran
- do’s are similar to Fortran; but there are no labels
- has vector operations like F90 -- highly recommended

Differences between Basis and Fortran

- The Basis language is interpreted, not compiled.
- Comments start with #.
- No statement numbers or `goto`'s.
- Input is free form: no special columns; continue by ending with (, [, ,, +, etc. Statements stack with ;.
- Spaces are significant and act as delimiters.
- All variables must be declared.
- Function names and formal arguments must not be typed (`real`, etc.)
- Functions can return any entity; e.g., arrays or entity combinations.
- Parameters are passed by value (i.e., copies), not by reference (i.e., addresses).
- Case sensitive; but reserved words can be all lower or upper.

Expressions

- Expressions are combinations of operands, operators and delimiters
  - Operands: have a value; e.g., constants, variables, or functions that return a value
  - Operators: do something; e.g., +, -, *, /, **, .le., <, //, !
  - Delimiters: separate items; e.g., commas, parentheses
- Examples

444/5280	2**13	2. ** .5	(-2.) ** .5
444./5280	(1==3)	(1==3.)	1==3
exp(-700)	sqrt(-2)	(-2. + 0i)**.5	-2 + 3i

- Basis doesn't crash with 1./0; it returns something

Expressions, cont.

- Operators are applied in order of precedence and from left to right
  - Lowest precedence: + and − . (These are unary and binary)
  - Next highest: * and /
  - Next: **.   Note `a**b**c` is `a**(b**c)`
- Delimiters: (), { }, [], ,, and : . (more on last 4 later)
  - () raise precedence like Fortran
  - {} inhibit evaluation (more later)

Variables: pre-declared

- Remember: all variables must be declared
- Examples of pre-declared Basis variables
  - `debug`, `yes`, `true` (*not* `.true.`), `fuzz`, `pi`
  - `$a`, `$b`, …, `$z` # These are type "`chameleon`" (later)
- See "List of Parser Variables," Chap. 32 of the Basis Manual
- (Attached packages also have pre-declared variables.)

Declaring variables

- Variables must begin with a lower-case letter.
- Basis has all the usual Fortran types: integer, real, real8, double (not double precision), complex, logical, character*n
- Variables can be initialized in the declaration statement:

```
real x, y, z=44.
logical v1=true, v2=false
```

- Notation for complex constants: 3. + 3i (no space before i)
- 10b for octal; 010x for hex (toggle output: oct, hex, dec)
- Variables not explicitly initialized are set to zero.

Try it

Expressions (not statements) given to the Basis interpreter (a.k.a. "the parser") are evaluated and printed

- Try some expressions
- Check me out on precedence
- Generate a syntax error
- See what value debug has; toggle it; redo the error
- Cause an overflow
- Cause an underflow
- What is fuzz? Change it and recalculate 2*pi
- Play around. $b=2; $c=3; $b; $c; ($b==$c); ($b=$c); $b

A couple handy scripts

- Basis script read upon start-up: .basis
  - In my ~/.basis:

```
integer logunit = basopen("bassession.log", "w")  # gets a unit number
baspecho(logunit)                                 # turns on logging
```

- csh alias to extract log-file commands: xbascom
  - alias xbascom grep --text '^> ' !* | sed s/'^> '//

- Basis script to control precision: dbprec

```
define complex double {complex}
    # To restore: define complex {complex}
define real {double}
    # To restore to single precision: define real {real}
```

Declaring variables, cont.: Arrays

- Array variables can have up to seven dimensions

  `real x(100),  y(-3:5, 7:10)`
  - Lowest subscript defaults to 1 (like Fortran)
  - Arrays are stored in column-major order (like Fortran)
- Square brackets [] build arrays

  `real xx=[ [pi, pi**2] ], [3,9] ], yy(3:4)=6`
  - Dimensions can be declared via initialization
- Constants are broadcast
- Useful pre-declared arrays: `ones()`, `iota()`, `spanl()`

  and array operators: `transpose()`, `fromone()`, `:=`, `!`, `outer()`, `shape()`, … (more later)
- The delimiters [], ,, and : are hereby illustrated.

More about variables

- Array indices:
  - default low, high like this: `arr1d(:)`, `arr2d( :4, 3)`, `arr2d( , 3: )`
  - can have increments: `x(::10)` (More re :: later)
- More types:
  - `chameleon`: assume type and shape of the RHS expression
  - `range`: indicate subscripts; e.g., `range x = 3:5`, `rr = ::-2`
  - `indirect`: useful with function arguments (call-by-value)
  - `structure`: composed of variously declared variables

Variables, cont.

- Variables are grouped in "packages"
  - e.g., packages `par`, `fft`, `svd`, `fit`, `ezc` in Basis itself

and, within packages, "groups" (more later)

- Variables have scope: global (at the prompt), local (inside a function)
  - `global real vv` # inside a function, this overrides local
  - `<pkg> real vv` # assigns `vv` to "package" `pkg`. All package-variable scopes are global (more later)
  - If there is more than one `vv` in your code –
    - `list vv` shows a table of their packages and precedence
    - `vv` can be prefixed with a pkg-name: `<pkg1>.vv`

Variables, cont.: character variables

- Syntax is character *n; e.g., character*2 c1= "hi".
- Concatenate with //; e.g., $c = c1 // ". How are you?"
- Basis has many built-in string-related functions.

Variables, about done

- To undeclare them (and release storage): `forget v1`, …
- Any questions?
- The requirement that variables must be declared can be turned off; but in general it is a bad idea to do so
- In general, Basis is highly customizable

Diversion: the LIST command

- The LIST command is very useful

Use it to find out about:

- individual variables and functions (list <var>)
- what packages are in a Basis-built code (list packages)
- what variables have the same name (list <var>)
- other variables in a particular package or group

- list by itself documents the list command

Variables: Try them

- Declare some arrays and initialize them using

  <a constant>, ones(), iota(), spanl() *

- Declare a real array z; then try real y = [z, z+1]
- Operate on your arrays with

  shape(), transpose(), fromone(), !, outer() *

  (more array operators later)

- * Hint: use the list command. ("!" is in the manual index)
- What do you think (iota(6:10)-ones(5))(7:8) produces?
- Find the type structure in the manual; declare a structure and extract an element
- Declare a variable autovar; then list autovar

Variables: pop quiz

- Declare a complex number and fill it with 2i; take the square root
- Declare a 3 x 4 array; fill the 2^{nd} row with all 5's
- real z=[[1,2,3],[4,5,6]]. What is [z,z+1] cf. z//(z+1) ?
- Declare an array with indices from -200 to 200
- Declare a range variable, rng1, to access every 5^{th} element of that array starting with -200 and another to access every 5^{th} element starting with -197
- Declare two character variables and concatenate them
- Use the list command on your variables

Manipulating arrays

- Especially because Basis is an interpreter, use array syntax-- a(...) = b(...)*c(...)**2 --as opposed to do loops
- Double colon notation--a:b:c--where a, b, or c is real:
  - If c is real, a:b:c is a vector of values spaced c apart

    chameleon time = 0. : 100. : 1.e-6

    (How would you fill time using iota() ? using spanl() ?) paws
  - If c is integer, a:b:c is a vector of length c (or c+1)
- The := operator appends the RHS to the LHS
  - real c=iota(3)+12; do $k=15,18; c := $k**2; c; list c; enddo paws
- The concatenation (//) operator appends one array to the end of another

Diversion: try some graphics

- win on   # puts up a graphics window
- plot y, x # plots y vs. x; the comma is optional
  - plot iota(20)**3 iota(20) # expressions are ok for y, x
  - nf # clear the frame.   sf resends a blanked-out plot
  - nf; plot iota(20)**3 color red # x-axis defaults to integers
  - nf; plot spanl(10,100,20) scale=linlog # log scale, '=' is optional
  - attr scale=loglog # replots with the new scale
  - attr color=cyan   # this one works on the next curve
  - plot iota(100)**3 iota(100) # no nf: adds the curve to plot
  - undo   # removes last plot command

Graphics diversion, cont.

- Set up some arrays for a few more plot examples:

```
real r(10,20), z=r;  integer ireg=z+1
r=outer(iota(0,9),ones(20));  z=outer(ones(10),spanl(0,100,20))
r(:5,:4);  z(:5,:4)  #  see how r and z came out
```

- Try these: `nf;  plot r(,4)`

```
nf; plot r(,4) z(,2) color green # why did the slope change?
plot r(,4) z(,6) color magenta;  plot r(,4) z(,6)*1.1
nf;  plot transpose(r)  color rainbow
```

- 'mesh' plots: `plotm z r ireg;    plot r z mark = circle`

```
attr scale loglin
ireg(:5,:8) = 0; nf; plotm z r ireg scale loglin
```

  – I have ignored plotm until now. Could be useful for core/sol plots.

Array manipulations, cont.

- Most Basis functions (sqrt, sin, exp, etc.) accept arrays as arguments and return objects with the argument's shape
  - E.g.: real xx = 0:2*pi:100; plot sin(xx) xx paws
- Find more such functions:
  - list sqrt # to find what group it is in paws
  - list <sqrt's group> # oops. list par.groups and take a good guess
- There are several ways to get information on an array. After real x(2,5,8), try
  - shape(x) # shape can reshape too. Try $x=shape(x, 2*5, 8); list $x
  - length(x) paws
  - list x

Array subscripts

- If subscripts are missing, the entire array is used
- Subscripts, e.g., on real x(-2:3, 4:8, 9:20), can be
  - the obvious: such as (0,4,9) or (,,11) or (0: , 5:6, :16)
  - but also: ([-1, 1, 3], 4, 10) # []s' for one-d only
  - or: (: , :) # the missing argument is taken to be the minimum value of the last (x's 3rd) dimension
- Of course, integer variables in place of integers or range variables equal to any of these subscript expressions can also be used.

Array subscripts, cont.

- All operands in an array expression must be the same size and shape. (Scalars are broadcast.)
- squeeze() gets rid of dimensions of length one
  - example:

```
integer ii(2,3,4)
shape(ii(1,,))              # = 1,3,4
shape(squeeze(ii(1,,)))     # = 3,4
```

- This is useful when errors would otherwise occur due to mismatched shapes

Array operators

- + - * /; **; & |; ==, etc., are component by component
- Recall shape(), fromone(), from previous slides
- Array arithmetic operators (*not* comp.-by-comp.) are
  - transpose() # transpose
  - *! # matrix multiply
  - /! # matrix divide (really)
  - ! (or .dot.) # inner product (operands need not be 1d)
  - outer() # outer product
- Set up a 2x2 matrix problem: M x = S. paws
  - Declare real x=s /! m Did it work? Print m*!x; m*!m; m!m

Pop quiz #2

- What is the difference between 1:100:1 and 1:100:.1?
- Declare two arrays and concatenate them. Print them out to see the result
- Fill a 3 x 4 array by several methods: brackets, colon notation; Basis functions iota, ones, outer, spanl
- Invoke a Basis function with a scalar; e.g., sin, sqrt
- Invoke the same function with a complex scalar, a vector, and a matrix
- Create a logical array by comparing two real arrays with a logical operator
- Declare a 3d array and then reshape it into a 1d array and also a 4d array (Corsica's poloidal flux is a 1d array)

if's and structured statements I

- Use the where() statement instead, where you can:

  where(logical expression, result if true [, result if false])

  – Works component by component.
  – Returns an object of the same type as 2nd and 3rd arguments
  – If the 3rd argument is missing, the returned size is the number of true elements; otherwise, it is the size of 1st arg
  – Try it: declare an array real x(30); x=ranf(x) paws
    - Now construct an array equal to one where x>.5, and -3 otherwise
    - Construct an array equal to x, but replace elements < .2 with 1.e-6
    - What is the number of elements i of x with x(i) < .25?
    - Is where(a>b, a, b) ⇔ max(a, b) ?
- See also gather(), not(), ior(), ...

if's and structured statements II

- Structured statements are pre-"compiled"
  - During input, Basis changes the prompt to >, >>, >>>, etc. (one > for each level of nesting).
  - If there is a syntax error, this is aborted. If there is no error--
  - do's, etc., are then executed; functions are merely defined.
- if's: Basis has the usual
  - if(<logical expression>) <single statement>
  - if(<*lexp*>) then; ...; [elseif() then; ... ;] [else; ...;] endif

if's and structured statements III

- do loops:
  - do $k=1, 10, 2; $k; if($k=3) break; enddo
  - do; <*slist*> ; enddo # goes forever (not a problem)
  - do; <*slist*> ; until (<*slist*>) # executes at least once
- while (<*lexp*>) ; <*slist*> ; endwhile
- for (<*forinit*>, <*lexp*>, <*slist1*>) ; <*slist*> ; endfor qqq
  - for ($j=1, $j<=10, $j=$j+1) ; <*slist*> ; endfor
- break, next [<*integer*>], and return provide jumps out of these blocks. (I have to test every time.)

Interrupts

- The parser can be interrupted with `^c` or the command `debugger`
- Try it: paws
  - `$k =0 ;  do ;  $k = $k + 1;  $k;  enddo`
  - `$k =0 ;  do ;  $k = $k + 1;  $k;  debugger;  enddo`

Timing

- list second, timer, partime paws

- Fill 1d arrays theta, from 0 to 2 pi; and phi, from 0 to pi, both of length 1001; and a 2d array

  y(theta, phi) = - (3/8 pi) **.5 sin(theta) cos (phi)

- Do this two ways: (1) a single double do loop; (2) array syntax; and time each way.

Graphics diversion II

- Contour plot: plotz y theta phi color rainbow # y from prev page
- Documentation for graphics
  - https://wci.llnl.gov/codes/basis/pdf/ezn.pdf ← excellent
  - http://www.ncarg.ucar.edu/supplements (autograph and conpack)
  - Use the list command
    - list # remind yourself of the list-command syntax
    - list packages # look for graphics things; notice "ezc"
    - list ezc.groups
    - list Key # an ezn groupname, shortened.
    - list conkey # charcter var, holds keywords for the plotc command
    - conkey # scale, style, thick, lev, etc.
    - [output <*filename*> # after this, output goes only there]
    - Key, # outputs the whole group. "," continues the line
    - Ezcurve
    - [output tty]

Graphics: contour plots

- plotz f, x, y, <keylist>   # ff
  - rectangular gridded data: f is 2d, x and y are 1d, and   f(i,j) = f(x(i),y(j))
  - mesh data: f, x, and y are 2d. f(i,j) = f(x(i,j),y(i,j))
    - So this is logically rectangular. Can also plot this data with plotc
  - scattered data: f, x, and y are 1d. f(i) = f(x(i),y(i))
    - Basis creates a rectangular mesh and interpolates
    - Keyword rsquared is required
- Keywords:
  - grid, scale, thick, style, font, mark, marksize, lev, color, rsquared, legend

Graphics, cont.

- There are many controls to customize plots
  - quadrant control, titles, text, Greek (`ezcstxqu`), …
  - well explained, with examples, in ezn.pdf and in the ncar doc's
  - e.g.'s also in corsica/scripts/graphics.bas
- Control the keyword values cf. Basis's defaults:
  - `ezcreset=true` # (default) the default scale, color, style, title… are restored on the next frame
  - `ezcreset=false` # changes made with `attr` are retained for next frames
  - The defaults themselves can be changed: `list EzcurveDefaults`

Graphics: more kinds of plots

- plotm x2d y2d ireg <keylist> # plot mesh (see earlier)
- plotb # plot mesh boundaries (= plotm … bnd=1)
- plotc f2d x2d y2d ireg <klist> # mesh-based plotz with ireg options
- plotf f2d x2d y2d ireg <klist> # fillmesh plot
- plotv x2d y2d dx2d dy2d ireg <klist> # plot vector field
- plotp x2d y2d <klist> # plot a polygonal mesh
- plotpf f2d x2d y2d <klist> # polygonal fillmesh plot
- srfplot(x1d, y1d, z2d, nx, ny, view) # wire-frame surface plot
- isoplot(t3d, nx, ny, nz, c0, view) # wire-frame isosurface plot

Graphics, cont.

- To save your plots

  cgm on # Basis leaves ...001.ncgm in your filespace

  [i]ctrans # NCAR - [interactively views] translates ncgm to ps

  -- has line-thickness control

  ncgm2cgm, cgm2ncgm < *<filein>* > *<fileout>* # NCAR

  ps2pdf # Adobe

  ~bulmer1/bin/ncgm2pdf –help # ncgm to pdf in one step

Functions I

- Syntax is:

```
function(arg1, ..., argn; opt_arg1, ...)
<slist>
[return <expr>;]
endf
```

E.g.:

```
function diff(x;msg)
   default(msg)="no";   if(msg ~= "no") << msg
   chameleon z=shape(x, length(x);  return z(2:) - z(1:length(z)-1)
endf
```

Functions II

- Neither the function nor its formal parameters are typed
  - Anything can be returned, including structures
- Optional arguments can be defaulted with `default()`

Functions III

- Remember scope (earlier):
  - local variables---declared inside functions---do not exist after the return
  - global variables (as at the top-level parser) can be defined inside functions
  - Experiment:

paws

```
real b(3,2)=0;   list b;   function f;   real b;   list b;   endf
                 list b;   f;   list b
list bbb;   function f;   real b;   global real bbb;   list local.b;   endf
            f;   list b, bbb
function f(b);   list local.b;   b=1;   endf;   f(b);   b
```

Functions IV

- Call by value: arguments are not changed upon return
  - Can switch to call-by-address with an & --- f(&b) --- but only for calling *compiled* functions from the parser:

```
real tt=0; tt; second(&tt); tt
```

paws

  - But there is type indirect:

paws

    - function w(nam); indirect y=nam; y(3)=7; endf
    - real x(10); call w("x") # sets x(3) to 7

Macros - Doc'd in Chap 24 of part 2

- Basis has two forms for macro definition
- `define` *<macro name>* *<macro definition>*
  - No arguments
- `mdef` *<macro name>*[()] = <*slist*> `mend`
  - Arguments $1, … $9, $*, $- can be used in <*slist*>
- {} suppress macro expansion and protect delimiters
- Macros are not pre-"compiled"
- Macros cf. functions have plusses and minuses
  - The `read` command is executed when the macro is invoked *in the order it is encountered in <slist>* (not so for functions—more later)
  - Everything is global—no encapsulation

The command command

- Basis's command command allows any function to be invoked with a command-line type of syntax
- Commands defined at run-time are defined as macros
  - E.g., with real x(10) and function w_() as w was before,

```
w_ command "x"   #  does same thing as call w_("x")
w_ command_s x  #  ditto. (  "x" here is OK too.)
define w  w_  command_s $1 # After this, then
w x                        #  ditto
```

- We include this here because plot commands are implemented with command
  - If you have trouble and, e.g., try list plotz, you'll get information about the syntax of the arguments in this form

The read command

- Input to the parser can be put in text files ("scripts") and passed to the parser with the read statement
  - read <*fname*> # "*fname*" or *fname* (not '*fname*') OK
  - After the last line is read, control returns to the level above
  - The parser acts on file contents just as at the prompt, executing as it goes, except --
  - Warning: read statements in structured blocks are not executed until the block has finished executing
- Where does Basis look for files?
  - list Path

paws

- echo = no # Turns off output to the terminal (and so the logfile) when reading from files.

The PFB package & PDB files

- Saves and restores data in a binary, portable form (PDB)
  - Data includes variables, functions, macros
  - To save: create mysavfile
    write <*item1, item2,…*>
    write functions | macros | variables | all
    close
  - To restore: restore mysavfile
    or
    real x = pfb.x # to copy in just x
- Compare two PDB files with pdbdiff
- Documentation:

  basis> list pfb

  pdbdiff –help qqq

  https://wci.llnl.gov/codes/basis/pdf/lib.pdf # Chap 8

paws

The ^

- ^ before a word toggles Basis from treating it as a string to an expression:

```
character filen="bas.in"
read ^filen
```

  – Useful for reading, opening or restoring files inside macros or functions

Stream i/o

Read (>>) and write (<<) to and from text files

- Stream output

```
integer ioun = basopen("myfile","w") # "r" to read
ioun << "Header"
ioun << var1   qqq << var   cf format
ioun  << var2 << return << var3  # return inserts a line break
call basclose(ioun)
```

  - list format # converts numbers to strings – next page
- [stdout] <<   # outputs to the terminal
  - Remember also output tstfile; ...; output tty

The format function

- list format # converts numbers to strings
- For integers:

    format(<*integer expr*>, <*field width*>)

    format(22,4) # prints 22

    paws
- For reals:

    format(<*real expr*>, <*field width*>, <*digits to right of decimal point*>, <*E or F format*>)

    format(2*pi, 10, 6, 1) # try it

    paws

Stream i/o, cont.

- Stream input

```
ioun >> var
```

Try it:

paws

```
cat >> tstfile
      c  comments here
          time = 2.56,  fac = 13.5e-2
          1.2  2.3  3.4  4.5
# back at the Basis prompt
      real x, y, d(2,2)
      integer ioun = basopen("tstfile", "r")
      ioun >> x; ioun >> y; ioun >> d   #  note that non-numbers are skipped
      call basclose(ioun)
```

- See manual for noisy, eof, autocr, …

Recursive parsing: execuser and parsestr

- Two routines are provided to call the parser:
  - execuser(<*script-function name*>)
  - parsestr(<*Basis commands*>)
- These make possible the insertion of scripts into compiled code (which must ultimately be called with a Basis command), enabling the user to alter or add models---develop new code---without recompiling the physics or waiting for code authors to do it.
  - Try it. paws
  - Corsica's makes extensive use of this; e.g., with its "hooks"

Miscellaneous features

- Command-line editing: have you used already? Try it
  - Use arrow keys or type ^r paws
- Execute Bourne shell commands: try it
  - !ls or basisexe("ls") paws
  - Set debug=yes; generate an error; type !cat <*tracefile name*>
- Error trapping: errortrp(off) # default is on. Try sqrt(-1)

  Type flush(<your log-file unit number>) first
- Guess how to insert a pause in a script. Try one. paws
- Stack control: set which package's variables have priority:

  list packages; parpush <*pkgname*>; list packages; parpop paws
- Script-file names, save-file names, or commands can be put on the execute line; but this can be customized

Final Exam

- Create and fill a 1D array, say te(100), with values from 0 to 1. Find the index at which te is closest to 0.2 (hint: the answer is a single expression)
- Open the file /projects/caltrans/IPPWinterSchool. Declare and fill a character array holding the names of the coils. Accomplish this in four lines, two of which are loops.
- Recall the difference between `debugger` and `paws`. What use can you think of for the former?